Item 293
git_comments:

git_commits:

1. **summary:** AVRO-1821 ADDENDUM: Fix checkstyle violations.
   message: AVRO-1821 ADDENDUM: Fix checkstyle violations.
   label: code-design

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

1. **summary:** Avro (Java) Memory Leak in ReflectData Caching
   description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a

{{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.
label: code-design

2. **summary:** Avro (Java) Memory Leak in ReflectData Caching
description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections. <Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import

org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.

3. **summary:** Avro (Java) Memory Leak in ReflectData Caching
   description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections. <Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to

use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.

4. **summary:** Avro (Java) Memory Leak in ReflectData Caching
   description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema =

Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.
label: code-design

5. **summary:** Avro (Java) Memory Leak in ReflectData Caching
description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic

classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.

6. **summary:** Avro (Java) Memory Leak in ReflectData Caching

description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.

7. **summary:** Avro (Java) Memory Leak in ReflectData Caching
description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true);

Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.

8. **summary:** Avro (Java) Memory Leak in ReflectData Caching
description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import

static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.

9. **summary:** Avro (Java) Memory Leak in ReflectData Caching
description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections. <Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other

reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.

10. **summary:** Avro (Java) Memory Leak in ReflectData Caching
    description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections. <Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new

HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.

11. **summary:** Avro (Java) Memory Leak in ReflectData Caching
    description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections. <Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} ---

a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.

12. **summary:** Avro (Java) Memory Leak in ReflectData Caching
    description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations: {code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections. <Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new

ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.
label: code-design

13. **summary:** Avro (Java) Memory Leak in ReflectData Caching
description: I think I have encountered one of the memory leaks described by AVRO-1283 in the way Java Avro implements field accessor caching in {{ReflectData}}. When a reflected object is serialized, the key of {{ClassAccessorData.bySchema}} (as retained by {{ReflectData.ACCESSOR_CACHE}}) retains a strong reference to the schema that was used to serialize the object, but there exists no code path for clearing these references after a schema will no longer be used. While in most cases, a class will probably only have one schema associated with it (created and cached by {{ReflectData.getSchema(Type)}}), I experienced {{OutOfMemoryError}} when serializing generic classes with dynamically-generated schemas. The following is a minimal example which will exhaust a 50MiB heap ({{-Xmx50m}}) after about 190K iterations:

{code:title=AvroMemoryLeakMinimal.java|borderStyle=solid} import java.io.ByteArrayOutputStream; import java.io.IOException; import java.util.Collections; import org.apache.avro.Schema; import org.apache.avro.io.BinaryEncoder; import org.apache.avro.io.EncoderFactory; import org.apache.avro.reflect.ReflectDatumWriter; public class AvroMemoryLeakMinimal { public static void main(String[] args) throws IOException { long count = 0; EncoderFactory encFactory = EncoderFactory.get(); try { while (true) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); // serialize ByteArrayOutputStream baos = new ByteArrayOutputStream(1024); BinaryEncoder encoder = encFactory.binaryEncoder(baos, null); (new ReflectDatumWriter<Object>(schema)).write(new Object(), encoder); byte[] result = baos.toByteArray(); count++; } } catch (OutOfMemoryError e) { System.out.print("Memory exhausted after "); System.out.print(count); System.out.println(" schemas"); throw e; } } } {code} I was able to fix the bug in the latest 1.9.0-SNAPSHOT from git with the following patch to {{ClassAccessorData.bySchema}} to use weak keys so that it properly released the {{Schema}} objects if no other threads are still referencing them: {code:title=ReflectData.java.patch|borderStyle=solid} --- a/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java +++ b/lang/java/avro/src/main/java/org/apache/avro/reflect/ReflectData.java @@ -57,6 +57,7 @@ import org.apache.avro.io.DatumWriter; import org.apache.avro.specific.FixedSize; import org.apache.avro.specific.SpecificData; import org.apache.avro.SchemaNormalization; +import org.apache.avro.util.WeakIdentityHashMap; import org.codehaus.jackson.JsonNode; import org.codehaus.jackson.node.NullNode; @@ -234,8 +235,8 @@ public class ReflectData extends SpecificData { private final Class<?> clazz; private final Map<String, FieldAccessor> byName = new HashMap<String, FieldAccessor>(); - private final IdentityHashMap<Schema, FieldAccessor[]> bySchema = - new IdentityHashMap<Schema, FieldAccessor[]>(); + private final WeakIdentityHashMap<Schema, FieldAccessor[]> bySchema = + new WeakIdentityHashMap<Schema, FieldAccessor[]>(); private ClassAccessorData(Class<?> c) { clazz = c; {code} Additionally, I'm not sure why an {{IdentityHashMap}} was used instead of a standard {{HashMap}}, since two equivalent schemas have the same set of {{FieldAccessor}}. Everything appears to work and all tests pass if I use a {{WeakHashMap}} instead of an {{WeakIdentityHashMap}}, but I don't know if there was some other reason object identity was important for this map. If a non-identity map can be used, this will help reduce memory/CPU usage further by not regenerating all the field accessors for equivalent schemas. The following unit test appears to reliably catch this bug, but is non-deterministic due to the nature of garbage collection (and I'm not sure there's a way around that): {code:title=TestReflectData.java|borderStyle=solid} package org.apache.avro.reflect; import org.apache.avro.Schema; import org.junit.Test; import java.io.IOException; import java.lang.reflect.Field; import java.util.Collections; import java.util.Map; import static org.hamcrest.Matchers.lessThan; import static org.junit.Assert.assertThat; public class TestReflectData { /** * Test if ReflectData is leaking {@link Schema} references */ @SuppressWarnings("unchecked") @Test public void testWeakSchemaCaching() throws IOException, NoSuchFieldException, IllegalAccessException { for (int i = 0; i < 1000; i++) { // Create schema Schema schema = Schema.createRecord("schema", null, null, false); schema.setFields(Collections.<Schema.Field>emptyList()); ReflectData.get().getRecordState(new Object(), schema); } // Reflect the number of schemas currently in the cache Field cacheField = ReflectData.class.getDeclaredField("ACCESSOR_CACHE"); cacheField.setAccessible(true); Map<Class<?>, ?> ACCESSOR_CACHE = (Map) cacheField.get(null); Object classData = ACCESSOR_CACHE.get(Object.class); Field bySchemaField = classData.getClass().getDeclaredField("bySchema"); bySchemaField.setAccessible(true); Map<Schema, FieldAccessor[]> accessors = (Map) bySchemaField.get(classData); System.gc(); // Not guaranteed reliable, but seems to be reliable enough for our purposes // See if the number of schemas in the cache is less than the number we generated - if so, then they are being released. assertThat("ReflectData cache should release references", accessors.size(), lessThan(1000)); } } {code} (Added {{org.hamcrest:hamcrest-all}} dependency to test scope for the built-in {{lessThan()}} matcher) ---- The current workaround that I'm using to mitigate the leak is to cache schemas and re-use older instances when I'm about to serialize an equivalent schema. Since most of the generated schemas are equivalent, this limits the number of leaked schemas to a handful. A more permanent workaround would be to switch to using a {{GenericRecord}} instead of a generic java class for the object that is being serialized, since this cuts out the use of {{ReflectData}} entirely.
label: code-design

jira_issues_comments:

1. **body:** Thanks for tracking this down, [~baharclerode]. I think you're right about the memory leak. It looks like you've done a great job putting together a test case and the fix. Could you put together a patch or pull request with those and we'll get it committed? For your question about the IdentityHashMap vs regular HashMap, I think the main idea is that because these lookups are in very tight loops, we want to avoid unnecessary operations. It's cheap to keep a copy per schema because there aren't typically a huge number of schemas in an app. But, we do like to use weak maps to avoid problems like this. Thanks for working on this!
   label: code-design
2. Attaching changes and unit tests as a patch from {{git format-patch}}
3. Commit 58daaf08a2637e0976cc124571200ce198b3143d in avro's branch refs/heads/master from [~rdblue] [https://git-wip-us.apache.org/repos/asf?p=avro.git;h=58daaf0] AVRO-1821: Fix possible memory leak of Schemas in ReflectData. Contributed by Byran Harclerode.
4. **body:** I committed the fix. Thanks for your contribution, [~baharclerode]! I updated it slightly to avoid the need for reflection in the test (used package-private instead) and I used a Guava weak identity map instead of the one we're trying to move away from.
   label: code-design
5. Commit 54eefb8d780237a7108c2a0c91b12282686426ad in avro's branch refs/heads/master from [~rdblue] [https://git-wip-us.apache.org/repos/asf?p=avro.git;h=54eefb8] AVRO-1821 ADDENDUM: Fix checkstyle violations.
6. [~rdblue] I was just verifying AVRO-1826 (the build rat problem) and it failed ... The file lang/java/avro/src/test/java/org/apache/avro/reflect/TestReflectData.java is missing the appropriate copyright message.
7. Commit b30b9e7a3365f50aa6f4481705937c462914764d in avro's branch refs/heads/master from [~rdblue] [https://git-wip-us.apache.org/repos/asf?p=avro.git;h=b30b9e7] AVRO-1821: Add license header to TestReflectData.
8. Fixed. Thanks for catching that, [~nielsbasjes]!
9. Any ETA when this patch will be released?
10. [~alunarbeach] it looks like [this|https://github.com/apache/avro/commit/ec8a091819a25bccf03adc868449f57f9c076d19] is already committed and released in 1.8.1.
11. Thanks [~nkollar]