Introduction

Team: Solution Accepted;
Members: Junxian Chen, Wen Chia Yang, Zihua Weng;
Realm is a mobile database that runs directly inside phones, tablets or wearables.

Feature 1: Adding more query methods

Background

Queries are the most common functions of a database.
Realm's query engine uses a Fluent interface to construct multi-clause queries.

```
public class User extends RealmObject {

    @PrimaryKey
    private String          name;
    private int             age;

    @Ignore
    private int             sessionId;

    // Standard getters & setters generated by your IDE...
    public String getName() { return name; }
    public void   setName(String name) { this.name = name; }
    public int    getAge() { return age; }
    public void   setAge(int age) { this.age = age; }
    public int    getSessionId() { return sessionId; }
    public void   setSessionId(int sessionId) { this.sessionId = sessionId;
}
}
```

Let say if we want to find all users named John or Peter, we would write:

```
// Build the query looking at all users:
RealmQuery<User> query = realm.where(User.class);

// Add query conditions:
query.equalTo("name", "John");
query.or().equalTo("name", "Peter");

// Execute the query:
RealmResults<User> result1 = query.findAll();

// Or alternatively do the same all at once (the "Fluent interface"):
RealmResults<User> result2 = realm.where(User.class)
                                  .equalTo("name", "John")
                                  .or()
                                  .equalTo("name", "Peter")
                                  .findAll();
```

Here Realm has already implemented a bundle of query functions for us, such as where(), equalTo(), or(), greaterThan(), count(), etc. Say, we would like to add these useful functions into the Realm Query module, then the following research will be helpful to the developers who intend to realize the features.

Research

As we can see from the above example, Realm first creates a ReamlQuery object and then adds query conditions on it.

Take equalTo() as an example, when equalTo() in io/realm/RealmQuery.java is executed, ReamlQuery calls equalTo() function from io/realm/internal/TableQuery.java.

```
public RealmQuery<E> equalTo(String fieldName, @Nullable String value) {
   return this.equalTo(fieldName, value, Case.SENSITIVE);
}

public RealmQuery<E> equalTo(String fieldName, @Nullable String value, Case casing) {
   realm.checkIfValid();

   return equalToWithoutThreadValidation(fieldName, value, casing);
}

private RealmQuery<E> equalToWithoutThreadValidation(String fieldName, @Nullable String
value, Case casing) {
   FieldDescriptor fd = schema.getColumnIndices(fieldName, RealmFieldType.STRING);
   this.query.equalTo(fd.getColumnIndices(), fd.getNativeTablePointers(), value, casing);
   return this;
}
```

Then equalTo() will call nativeEqual which is a Realm C++ library method and return a TableQuery object itself.

TableQuery class supports all low level methods (equalTo/greaterThan/lessThan/count) a table has. All mentioned native communications to the Realm C++ library are also handled by this class.

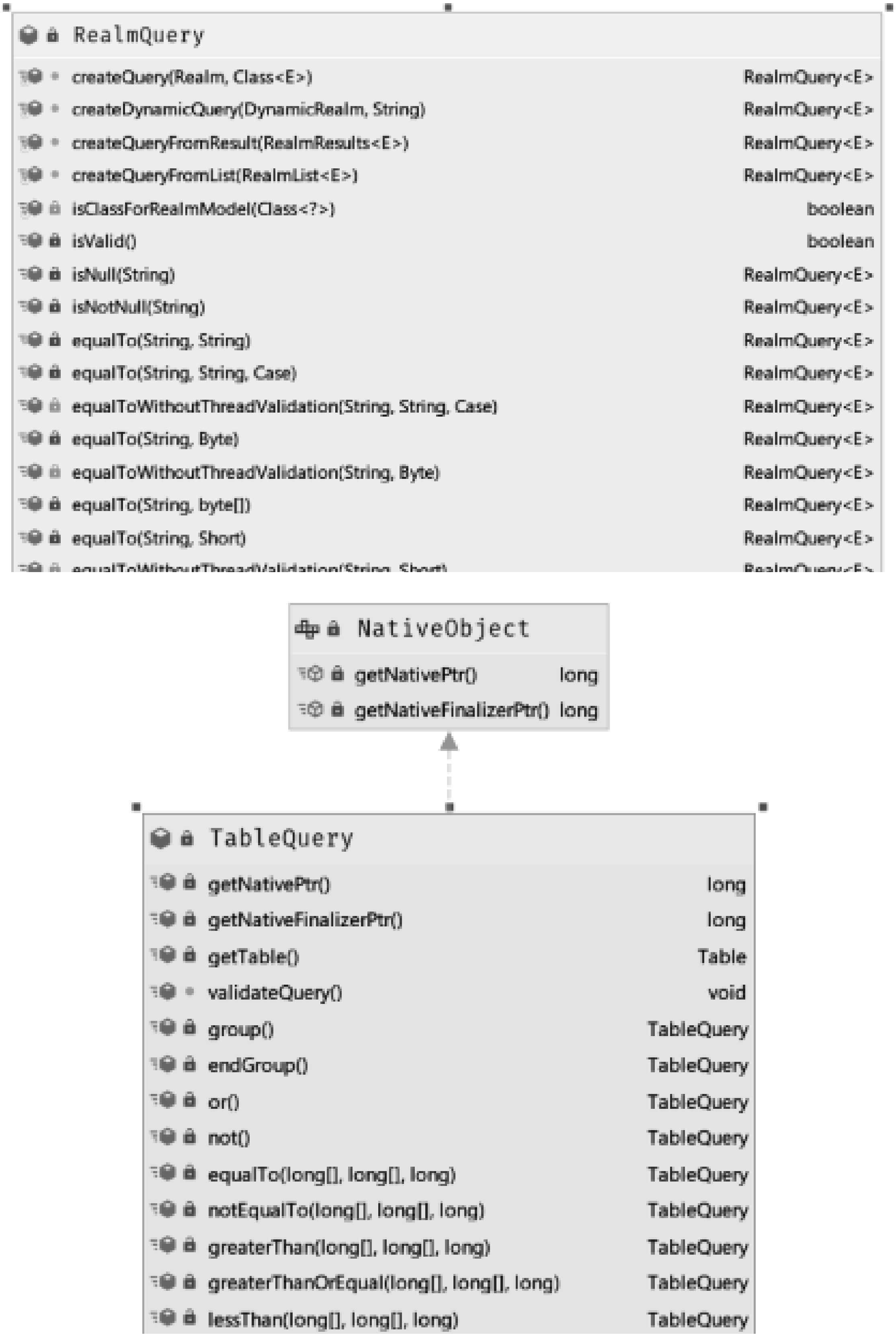
```
// Equals
public TableQuery equalTo(long[] columnIndexes, long[] tablePtrs, @Nullable String value,
Case caseSensitive) {
   nativeEqual(nativePtr, columnIndexes, tablePtrs, value, caseSensitive.getValue());
   queryValidated = false;
   return this;
}
```

```
private native void nativeEqual(long nativeQueryPtr, long[] columnIndex,
long[] tablePtrs, long value);
```

Going back to the equalTo() function in ReamlQuery, the query field in ReamlQuery is assigned to the return TableQuery object. Finally, the function returns a RealmQuery object.

Conclusion

In conclusion, for developers who want to add more query function to Realm, one should add customized method to io/realm/RealmQuery.java and the corresponding method in io/realm/internal/TableQuery.java which calls a defined C++ library function.



RealmQuery

createQuery(Realm, Class<E>)	RealmQuery<E>
createDynamicQuery(DynamicRealm, String)	RealmQuery<E>
createQueryFromResult(RealmResults<E>)	RealmQuery<E>
createQueryFromList(RealmList<E>)	RealmQuery<E>
isClassForRealmModel(Class<?>)	boolean
isValid()	boolean
isNull(String)	RealmQuery<E>
isNotNull(String)	RealmQuery<E>
equalTo(String, String)	RealmQuery<E>
equalTo(String, String, Case)	RealmQuery<E>
equalToWithoutThreadValidation(String, String, Case)	RealmQuery<E>
equalTo(String, Byte)	RealmQuery<E>
equalToWithoutThreadValidation(String, Byte)	RealmQuery<E>
equalTo(String, byte[])	RealmQuery<E>
equalTo(String, Short)	RealmQuery<E>
equalToWithoutThreadValidation(String, Short)	RealmQuery<E>

NativeObject

getNativePtr()	long
getNativeFinalizerPtr()	long

TableQuery

getNativePtr()	long
getNativeFinalizerPtr()	long
getTable()	Table
validateQuery()	void
group()	TableQuery
endGroup()	TableQuery
or()	TableQuery
not()	TableQuery
equalTo(long[], long[], long)	TableQuery
notEqualTo(long[], long[], long)	TableQuery
greaterThan(long[], long[], long)	TableQuery
greaterThanOrEqual(long[], long[], long)	TableQuery
lessThan(long[], long[], long)	TableQuery

Feature 2: Adding more encryption methods

Background

From the official documentation we know that the Realm database file can be encrypted with a 512-bit key (i.e. 64 bytes), using standard AES-256 encryption algorithm. In the future we may want to add more encryption methods like Triple DES, RSA, ChaCha20 or TwoFish to enhance security of the database. To do so, we have to find out where these encryption methods are adopted.

Research

First let’s start with a code snippet which is likely to appear in our Android project:

```
byte[] key = new byte[64];
new SecureRandom().nextBytes(key);
RealmConfiguration config = new RealmConfiguration.Builder()
  .encryptionKey(key)
  .build();

Realm realm = Realm.getInstance(config);
```

The snippet above randomly generated a 64-byte key which served as the encryption key of our Realm database. Here a new RealmConfiguration.Builder was created and the method encryptionKey(key) was called. RealmConfiguration.Builder is an inner class of RealmConfiguration so we should check inside RealmConfiguration.java:

RealmConfiguration.java:548

```
public Builder encryptionKey(byte[] key) {
   //noinspection ConstantConditions
   if (key == null) {
       throw new IllegalArgumentException("A non-null key must be provided");
   }
   if (key.length != KEY_LENGTH) {
       throw new IllegalArgumentException(String.format(Locale.US,
               "The provided key must be %s bytes. Yours was: %s",
               KEY_LENGTH, key.length));
   }
   this.key = Arrays.copyOf(key, key.length);
   return this;
}
```

We navigated to RealmConfiguration.java, and inside the RealmConfiguration.Builder class we located the encryptionKey(key) method. Basically what it did was assigning the key value to the Builder and then return the Builder itself for chaining.

RealmConfiguration.java:794

```
public RealmConfiguration build() {
   // Check that readOnly() was applied to legal configuration. Right now it should only be allowed if
   // an assetFile is configured
   if (readOnly) {
      // ...

   return new RealmConfiguration(directory,
           fileName,
           getCanonicalPath(new File(directory, fileName)),
           assetFilePath,
           key,
           schemaVersion,
           migration,
           deleteRealmIfMigrationNeeded,
           durability,
           createSchemaMediator(modules, debugSchema),
           rxFactory,
           initialDataTransaction,
           readOnly,
           compactOnLaunch,
           false
   );
}
```

Finally by calling the build() method, a RealmConfiguration was created. Please note that the key is now passed to the new RealmConfiguration. Now we are wondering how the key was used across the Realm project. Inside RealmConfiguration, the only way to access the key is calling getEncryptionKey():

RealmConfiguration.java:152

```
public byte[] getEncryptionKey() {
    return key == null ? null : Arrays.copyOf(key, key.length);
}
```

However, there are no usages of getEncryptionKey() across the project. We should look somewhere else.

Then we noticed “`Realm realm = Realm.getInstance(config)`”. The configuration was passed to a static method inside the Realm class called “getInstance()”. We traced the passing route of the configuration:

Realm.java:410

```
public static Realm getInstance(RealmConfiguration configuration) {
    //noinspection ConstantConditions
    if (configuration == null) {
        throw new IllegalArgumentException(NULL_CONFIG_MSG);
    }
    return RealmCache.createRealmOrGetFromCache(configuration, Realm.class);
}
```

RealmCache:286

```
static <E extends BaseRealm> E createRealmOrGetFromCache(RealmConfiguration
configuration,
        Class<E> realmClass) {
    RealmCache cache = getCache(configuration.getPath(), true);

    return cache.doCreateRealmOrGetFromCache(configuration, realmClass);
}
```

RealmCache:293

```
private synchronized <E extends BaseRealm> E
doCreateRealmOrGetFromCache(RealmConfiguration configuration,
        Class<E> realmClass)
```

RealmCache:312 (Inside RealmCache.`doCreateRealmOrGetFromCache()`)

```
OsRealmConfig osConfig = new OsRealmConfig.Builder(configuration).build();
```

OsRealmConfig.java:87 (The declaration of OsRealmConfig.Builder)

```
public static class Builder
```

OsRealmConfig.java:98 (The constructor of OsRealmConfig.Builder)

```
public Builder(RealmConfiguration configuration) {
   this.configuration = configuration;
}
```

OsRealmConfig.java:150 (The build() method of OsRealmConfig.Builder)

```
public OsRealmConfig build() {
   return new OsRealmConfig(configuration, fifoFallbackDir,
autoUpdateNotification, schemaInfo,
           migrationCallback, initializationCallback);
}
```

OsRealmConfig.java:198 (The constructor of `OsRealmConfig`)

```
private OsRealmConfig(final RealmConfiguration config,
                      String fifoFallbackDir,
                      boolean autoUpdateNotification,
                      @Nullable OsSchemaInfo schemaInfo,
                      @Nullable OsSharedRealm.MigrationCallback
migrationCallback,
                      @Nullable OsSharedRealm.InitializationCallback
initializationCallback)
```

OsRealmConfig.java:237 (Inside the constructor of `OsRealmConfig`)

```
// Set encryption key
byte[] key = config.getEncryptionKey();
if (key != null) {
   nativeSetEncryptionKey(nativePtr, key);
}
```

OsRealmConfig.java:368 (The declaration of `nativeSetEncryptionKey`)

```
private static native void nativeSetEncryptionKey(long nativePtr, byte[] key);
```

io_realm_internal_OsRealmConfig.cpp:97

```
JNIEXPORT void JNICALL
Java_io_realm_internal_OsRealmConfig_nativeSetEncryptionKey(JNIEnv* env,
jclass,
jlong native_ptr,
jbyteArray j_key_array)
{
   TR_ENTER_PTR(native_ptr)
   try {
       JByteArrayAccessor jarray_accessor(env, j_key_array);
       auto& config = *reinterpret_cast<Realm::Config*>(native_ptr);
       // Encryption key should be set before creating sync_config.
       REALM_ASSERT(!config.sync_config);
       config.encryption_key = jarray_accessor.transform<std::vector<char>>();
   }
   CATCH_STD()
}
```

Here, we have found that the encryption key setter is a native C++ function. We continued our search but did not find anything related to encryption in our **realm-java** project. Later we searched the Internet and found a page on StackOverflow, which said the encryption module was embedded in the **realm-core** project. The default encryption algorithm in Realm is AES-256 which is declared in aes_cryptor.hpp. Realm handles different encryption algorithms by realm::util::encryption_read_barrier and realm::util::encryption_write_barrier using different file mapping (alloc_slab.cpp).

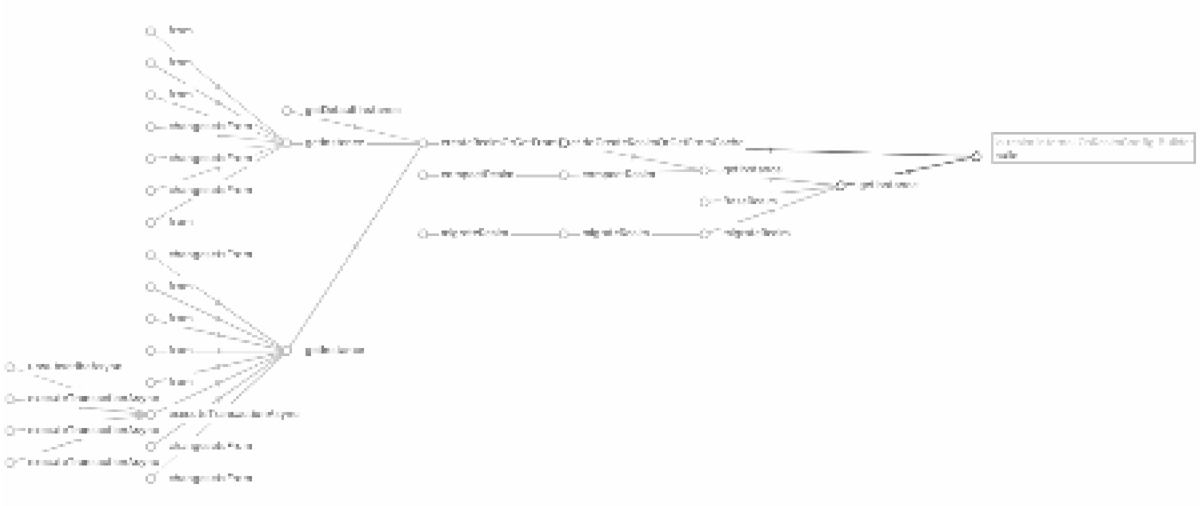
```
realm::util::encryption_read_barrier(addr, Array::header_size,
map->get_encrypted_mapping(),Array::get_byte_size_from_header);
```

Here the encrypted_mapping is retrieved from encrypted_file_mapping.cpp.

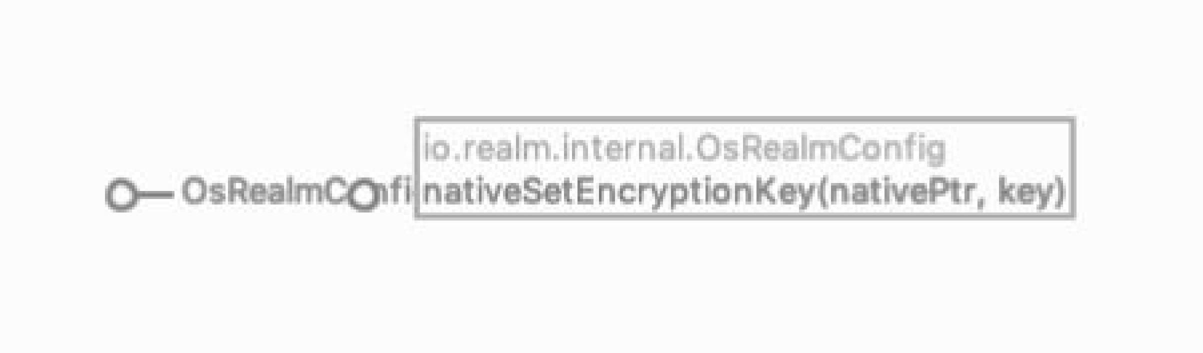
Theoretically, in encrypted_file_mapping.cpp, we could add a new Cryptor class here and implement different encryption methods.

Conclusion

OsRealmConfig.build() is called by multiple places. Here in our feature 2, we only focus on the path: build() -> doCreateRealmOrGetFromCache -> createRealmOrGetFromCache -> getInstance.



nativeSetSchemaConfig is the only one method called in OsRealmConfig.


OsRealmConfi
io.realm.internal.OsRealmConfig
nativeSetEncryptionKey(nativePtr, key)

By tracing the config we located the feature can be added by modifying the realm-core project.

Junxian Chen, Wen Chia Yang, Zihua Weng
Feb. 5, 2020