Project Part 4: Realm-Java

Authors: Wen-Chia Yang, Junxian Chen, Zihua Weng

February 26, 2020

ABSTRACTION

Objectives

In this report, we want to research Realm Database from the following aspects:

1. The architecture of the realm-core
2. Different aspects of social context in realm-java
3. Interesting pull requests and issues in realm-java

Source Code

Repository: https://github.com/solution-accepted/realm-java

Branch: master

RESEARCH

Overview

System architecture is the conceptual model that defines the structure, behavior, and more views of a system[1]. In this section, we will focus on a higher level of how a database is implemented in the realm-core project.

Since the realm-core project only documents a limit parts of its entire database design, to understand the Realm Database architecture, we started from assuming it shares a similar structure of a Database Management System which has a layered structure and is composed of at least five components(or sub-systems)[2]. They are query processor, relation manager, file and access methods, buffer manager and disk space manager, and transaction manager[2]. By thinking about this prerequisite domain knowledge, we try to use a top-down strategy to analyze the realm-core project. We read the project source code to evaluate our assumption and kept modify our model according to how Realm works and simulated continually. Figure 1 shows the final structure that we extracted from realm-core and the relationship between realm-core and realm-java.

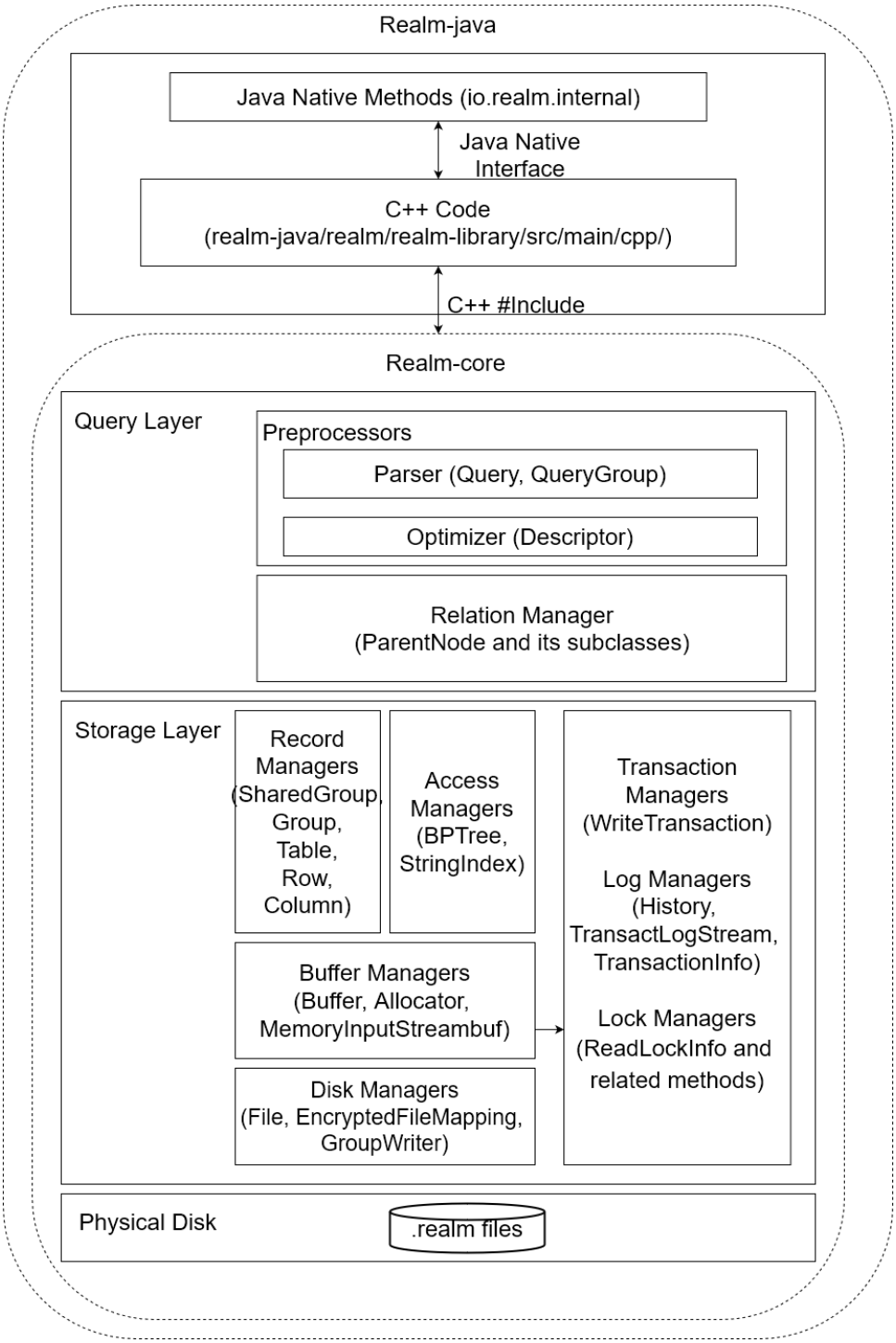

Figure 1: System Architecture of Realm-core

The following paragraphs are detailed explanations for each component.

Realm-java

Realm-java is a wrapper of realm-core. Through Java Native Interface, Java code in realm-java calls realm-java C++ code, which directly includes realm-core header. Most essential features are implemented within realm-core.

Query Processor

Since the architecture of Realm is layered, the query processor is the first place that processes commands from outside(other applications). Basically, the main role of the query processor is to analyze what the query is and to parse the query into chunks that can be used for the query engine. In Realm, a query can be viewed as a combination of conditions and actions. A condition for a column might be “contains”, “like”, “equal”, etc (learned more conditions here: https://github.com/realm/realm-core/blob/master/src/realm/query_conditions.hpp). The action can be denoted as actions in an aggregated column. Some examples of actions are sum, maximum, minimum, etc (learn more actions here: https://github.com/realm/realm-core/blob/master/src/realm/query.hpp).

Relation Manager

In the realm-core project, it implements these components by using a Query Engine. Basically, the query engine serves as a role to execute a query within its different combinations. For example, if a query like one.less(9.99).two.equal(“price”) on double and string columns, this will be processed as two nodes with the condition and data type which is defined in the query engine (Less-FloatDoubleNode and Equal-StringNode). These nodes are relational operators that are inherited from ParentNode and are created with different generic data types inside the query engine[3].

Files and Access Methods

Files here refer to “Files of Records”. These files form the basis of the record-based (or row-based) APIs which are provided for the upper layer. The common concepts related here are tables, rows, and columns. As described in an official document[4], Realm also follows the Table-Row-Column pattern. What makes Realm different is that the core entity in Realm is the row instead of the table. That is to say, there are no real tables in Realm because they are generated in run time rather than being stored in files. What is actually stored are columns. We have found relevant classes like Group, Table, Row, Column implemented in their source code files.

As for access methods, an important topic here is indexing. This design is intended to accelerate the speed of query execution. In the realm-core, a specialized BPTree class is implemented for the indexing purpose.

Buffer Managers

Database Buffer is a special area in memory for caching pages of data files and index files. Realm-core has a utility named Buffer, which is a class that represents the buffer concept that owns a region of memory and knows its size. This utility Buffer is seen in TransactLog related classes such as TransactLogBufferStream, TransactLogConvenientEncoder, TransactLogParser. Other important classes are Allocator and SlabAlloc, which are responsible for allocating memory for certain operations. SlabAlloc, which inherits Allocator, is the allocator that is used to manage the memory of a Realm group, i.e., a Realm database.

Disk Managers

Disk Managers have direct access to the physical disk and are responsible for disk space management and I/O operation management. Involving classes include File, EncryptedFileMapping, and GroupWriter.

File is a utility class that defines common behaviors of a file on disk, with some customizations meeting the special needs of the realm-core project, such as encryption and lock mechanism. EncryptedFileMapping is another utility class that handles file encryption.

GroupWriter is a class controlling a memory-mapped window into a file. It has two important public methods named write_group() and commit(). The method write_group() writes all changed array nodes into free space and returns the top ref (reference), which can be passed as a parameter to the commit() method. The method commit() receives top ref and then flushes changes to a physical medium, then writes the new top ref to the file header, then flushes again[5].

Transaction Manager

A transaction manager is part of an application that is responsible for coordinating transactions across one or more resources[6]. The transaction manager will ask resource managers to prepare and notify them if the transaction is committed or aborted. In realm-core, TransactLogStream and related objects maintain logs to record all events, and information for each transaction is stored in TransactionInfo. They are the log manager in Realm. ReadLockInfo and related lock methods in ShareGroup are executed as the lock manager. If the transaction is successful, WriteTransaction will write the result into ShareGroup (database) and update TransactionInfo, otherwise, WriteTransaction will trigger SharedGroup::rollback() which calls Replication::abort_transact() to abort the transaction. Since in Realm, the transaction is operated on the replicated table, rollback does not need to restore the data but delete the replicates.

Social Context

Realm is a complex system. A Realm family is composed of the realm-core, realm-java, realm-cocoa, realm-js, and realm-dotnet. Here, we only focus on realm-java.

State of realm-java

Realm's development began at the end of 2010 under the name TightDB[7]. The realm-java project was initiated on Apr. 20, 2012, when they had their first commit[8]. Its first stable version was released in June 2016. From Figure 2, we assume realm-java was in the implementation phase starting from mid-2014 to mid-2016. Since then it has entered the phase of maintenance and improvement. As the system becomes more stable, the amount of commits shows a decreasing trend.

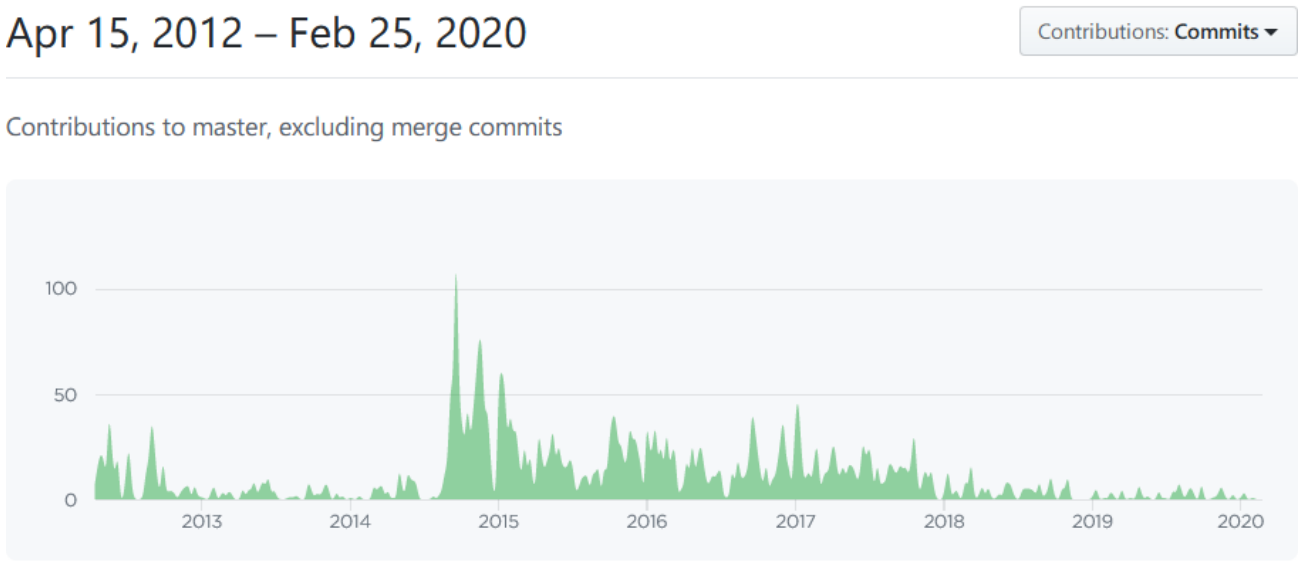

Figure 2: Graph of numbers of commits from 2012 to present[9]

As shown in Figure 2, in the most recent month, 11 issues are being closed and 14 created, 4 pull requests being merged and 2 proposed, so it is still under active maintenance. However, the commits are dramatically fewer compared to its most active period. We suppose that as the project grows, the possibility is low that new major features will be introduced.

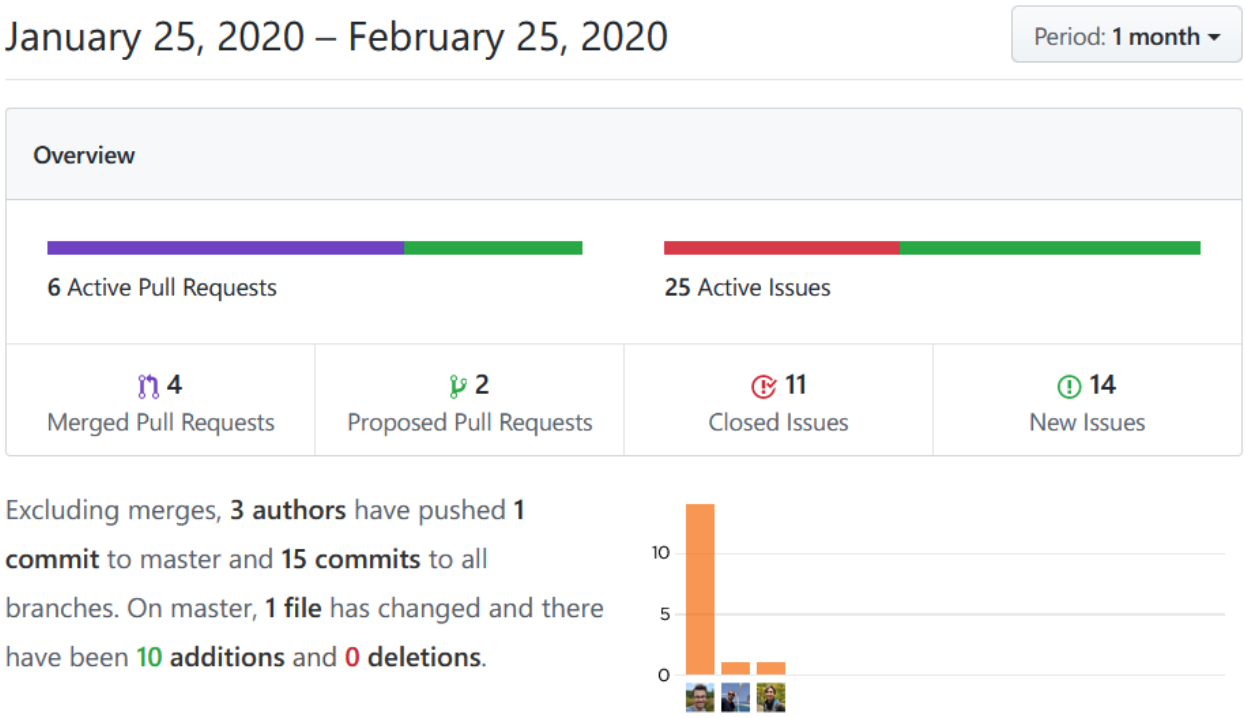

Figure 2: Overview of the "pulse" in a month[10]

Nowadays, it claims that over 100,000 developers use Realm[11]. The latest version of realm-java 6.1.0 was released on Jan. 17, 2020, and it has 8270 commits on the master branch and 385 open issues up to date. There are nearly 20 active developers who have over 10 commits.

After the acquisition of Realm by MongoDB, the latest roadmap indicated that Realm will be integrated into MongoDB as "MongoDB Realm" by mid-2020[12].

Standards

There are a few guidelines[13] for making contributions to the project, covering the topics like filing issues, accepting CLA (Contributor License Agreement), and writing conformant code, as explained as follows:

1. A good issue should include goals, expected results, actual results, steps to reproduce the problem, code sample, and version information of environments.
2. Before filing pull requests, contributors have to accept a CLA prepared by Realm.
3. It is required for contributors to do the followings for writing conformant code:
    - Follow the existing code style. Use clang-format for C++ source code.
    - Nullability of parameters and return types must be annotated with JSR305 annotations (@Nullable in this case).
    - Write JUnit4 tests and aim for 100% code coverage. Follow other guidelines specified for writing unit tests.
    - Write Javadoc for all public classes and methods. Follow other guidelines specified for writing Javadoc.
    - Know differences between master branch and releases branch.

More general details can be found in the Contributor Covenant[14]

How to contribute

The typical process of contributing code bug fixes can be summarized as follows:

1. Accept Realm's CLA;
2. Prepare a pull request:
    - Identify the issue and locate problematic code;
    - Modify the source files as needed to fix bugs and be sure to follow coding standards;
    - Write test cases to verify bugs are fixed;
3. Submit the pull request and wait for a review;
    - If merged, then the fix will be released and the process ends;
    - If rejected, read the comments from maintainers then go back to step 2.

Contributions like feature enhancements, typo fixes, documentation improvements, have a similar process stated above.

It is interesting to see that there is a pull request (https://github.com/realm/realm-java/pull/1867) opened on Dec. 1, 2015, with 179 comments, still being open and not merged. From that, we can see it may take a long time to fully accept a pull request because during this process new comments may be thrown or the issue has low priority.

Tools that can be used by contributors

1. Git: The source code of realm-java is hosted on GitHub and managed by Git version control system.
2. Jenkins: After pull requests are submitted, Jenkins can help automate the build process and see if test cases pass.
3. Gradle: Gradle is the build tool for realm-java and it is needed to build realm-java from source.
4. Android Studio: This is needed to manage Android SDKs and any other build tools like CMake. It is the recommended IDE because realm-java targets to run on Android devices and special configurations are provided [https://github.com/realm/realm-java/tree/master/realm/config/studio] to match official code styles and linter settings.
5. Static analyzers: There are config files for three different static analyzers. They can help both contributors and reviewers check potential bugs or code style problems.
    - Checkstyle: https://github.com/realm/realm-java/blob/master/realm/config/checkstyle/checkstyle.xml
    - FindBugs: https://github.com/realm/realm-java/blob/master/realm/config/findbugs/findbugs-filter.xml
    - PMD: https://github.com/realm/realm-java/blob/master/realm/config/pmd/ruleset.xml

Pull Requests

#6116 Add support for text-based query predicates

https://github.com/realm/realm-java/pull/6116

This PR adds support for Cores query parser. With the introduction of the new method rawPredicate, it is possible to convert string-based query input to Query objects. This PR also covers the following features: Full query support for string-based query input including subqueries, aggregates, multi-field comparison, and other things. The supported new query features added to the realm-core project immediately. It is possible to mix and match raw and typed predicates. Allows Query-based Subscriptions to return Query objects and local query results.

```
// Simple
RealmResults<Person> results = realm.where(Person.class)
    .rawPredicate("name = 'Jane'")
    .findAll();

// Mixed predicates
RealmResults<Person> results = realm.where(Person.class)
    .rawPredicate("name = 'Jane'")
    .rawPredicate("children@count > 2")
    .equalTo("lastName", "Doe")
    .findAll();

// With arguments
RealmResults<Person> results = realm.where(Person.class)
    .rawPredicate("name = '$s'", "Jane")
    .findAll();
```

#929 RealmConfiguration + new constructors

https://github.com/realm/realm-java/pull/929

This PR changes how Realms are created. It replaces the previous constructors with a RealmConfiguration that is constructed using the Builder pattern mentioned below. It makes it very scalable without having to add new constructors in Realm.java. It also enables the coming in-memory/read-only realm database. It also allows us to move the Android Context away from Realm.java. It will be less error-prone to create a configuration from context or a folder. Additionally, it introduces the concept of a default Realm without customized configuration.

```
// Full set of options for configuration:
RealmConfiguration config = new RealmConfiguration.Builder(getContext())
                    .name("default.realm")
                    .encryptionKey(getKey())
                    .schemaVersion(42)
                    .migration(new MyMigration())
                    .addModule(new MyModule())  // Add 1 module to already set modules
                    .setModules(new MyModule(),  new MyLibraryModule()) // Replace current modules
                    .deleteRealmIfMigrationNeeded()
                    .schema(Foo.class, Bar.class) // package protected method, only used for internal testing.
                    .build()

// Constructors are:
RealmConfiguration.Builder builder = new RealmConfiguration.Builder(Context ctx);
RealmConfiguration.Builder builder = new RealmConfiguration.Builder(File realmDir);

// Default Realm
Realm.setDefaultConfiguration(config); // A default configuration must be set first
Realm.getDefaultInstance(); // Making it even easier to get a default instance
Realm.getInstance(config); // Getting the instance for any given configuration
```

#1214 async queries & write transaction

https://github.com/realm/realm-java/pull/1214

This PR added new methods to support async queries and async write transactions. Normally, we execute the sync query in the caller thread using query.findAll(). New query methods findAllAsync, findAllSortedAsync, findFirstAsync enable executing the query on a worker thread. they accept the same parameters as their synchronous counterpart. The sync executeTransaction is overloaded to accept a new parameter Transaction Callback. New async method executeTransactionAsync(transaction) would perform writes on a background thread without blocking. The async transaction returns a Request object that can be used to cancel a scheduled/pending write transaction if it didn't complete.

#6730 Add support for Embedded Objects

https://github.com/realm/realm-java/pull/6730

This PR adds support for "Embedded Objects". For non-synced Realms, it can be be used to implement cascading deletes in some scenarios. For synced Realms, it also describes how the data is serialized once it reaches a MongoDB server. An embedded object has the following features: It will be deleted if the parent is deleted or the object is set to null. Only one parent can ever link to an embedded object. Embedded objects can be parents themselves. Queries across all embedded objects are not possible. A query must always be from the perspective of the parent object or the RealmList in the parent object. It could not be moved once assigned a parent.

#6590 Add support for frozen objects

https://github.com/realm/realm-java/pull/6590

This PR adds support for frozen Realm objects. The major benefit of frozen objects is that they are not thread confined and can thus be accessed on any thread. This is especially important for most kinds of stream-based architectures (which is most of them at this point).
The trade-off is that the objects are then immutable (which is actually considered a positive for most reactive architectures on Android). Anything can be frozen in Realm. This PR adds two new methods to all top-level Realm objects: freeze() and isFrozen(). Freezing an object will freeze the entire object graph at that version, effectively turning it read-only. Doing this will also lift the thread-confined restriction on it.

Issues

#2280 Import/Export Support

https://github.com/realm/realm-java/issues/2880

This issue is opened by a developer who wants to make importing and exporting a Realm source file easily. To be more specific, he hopes that realm-java can support file types such as CSV, XLS or PDF. Currently, if you want to manage your Realm data without coding, you can only use official Realm Studio to visualize it since the real Realm data source file is exported as "<filename>.realm." Therefore, this would make developers manage Realm data conveniently. This is a really useful function for developers so there are a lot of discussions below. However, it is still not being implemented(still opened) since the core maintainer thought the priority of this feature is not that high. Alternatively, some key developers offer someone's custom solutions from Stackoverflow, or developers should implement it by using current public APIs.

#776 Enums Support

https://github.com/realm/realm-java/issues/776

This issue is opened by a developer who thinks enumerations are a better choice to substitute integers while using the realm-java APIs. We think this topic is interesting because how to save enumeration data into a database is a highly debatable question. Generally, people are considering ways between integers and strings. They all have pros and cons individually. For example, if storing numeric values in each row, it is hard to understand what it means originally. On the other hand, if storing a string value per row, it will cost too many spaces. For now, it is okay for most cases storing the string type in a database since the names of enumerations are usually short. It will not cause too many problems. Among the discussions, I found something “interesting”. The Realm official implemented this function and merged it with the master after 1 year, and they even provided a sample to the developer on how to use it. However, this issue is still opening. Developers changed their topic and some of them started to ask some implement questions.

#761 Inheritance / Polymorphism

https://github.com/realm/realm-java/issues/761

This issue is actually opened by the core maintainer Christian Melchior. This feature is very useful because for now, there is no way to inherit from RealmObject. Developers need to create multiple instances. This causes messy and repetitive codebases. Among the discussion, some key developers respond to this issue and mentioned that developers can use composition instead of inheritance to fulfill their needs. Still, this solution is not working for every case. Therefore, developers are still waiting for the official release for this project. Recently, the head of product solutions of MongoDB, Robert Oberhofer, mentioned an update for this feature. It is quite exciting because the officials are finally implementing this feature now and will releasing them within a year.

#759 Support for Set and Map in model classes

https://github.com/realm/realm-java/issues/759

This issue is posted by the core maintainer Christian Melchior. I think this is a good topic because Set and Map are very popular data types that will be used often in Java. If the realm-java provides a predefined data type such as “RealmSet” or “RealmMap”, it would save developers some time to create boilerplate code. For now, developers need to create a MapEntry class that contains key and value to make you Realm model class uses this class as a data type.

```
class CustomObject extends RealmObject {
    private RealmList<CustomMapEntry> customMap;
}

class CustomMapEntry extends RealmObject {
    private String key;
    private CustomClass value;
}
```

If the official release this topic, it will be like the following code, it is definitely a very “interesting” implementation for developers:

```
class CustomObject extends RealmObject {
    private RealmMap<String, CustomClass> customMap;
}
```

#1470 @SerializedName annotation for defining Class/Field names

https://github.com/realm/realm-java/issues/1470

This issue is opened by one of the key developers Chen Mulong. Originally, he wanted to create a new annotation '@SerializedName' that is used in the Android Retrofit library to map names between JSON string and RealmObject. This topic is very useful because the API request is tightened to Database service. Later, another developer suggested adding an additional annotation for this topic, which is @SerializedObjectId(s)Name. He wanted to solve the problems that the API only returns an object, which makes current APIs in Realm hard to process. The core maintainer Christian Melchior actually replied to his thinking process and a plan for this topic. He thought this was a cross-platform issue and it had a lot to change for them. Until recently, there is still no support for this issue. However, a developer implemented his own library to support this topic and released it to the public, which is very nice. Even so, we still hope the official can offer stable support for public APIs.

REFERENCES

[1] System architecture https://en.wikipedia.org/wiki/Systems_architecture

[2] Structure of a DBMS https://docs.google.com/presentation/d/1ZLFpl2IgV1RrvSP2MWcIiqcOpMUb1IhX2Vbi3pxcX-8/edit

[3] query engine https://github.com/realm/realm-core/blob/master/doc/development/query_engine.pdf

[4] core architecture https://github.com/realm/realm-core/blob/master/doc/primer/primer_architecture.md#core-architecture

[5] GroupWriter https://github.com/realm/realm-core/blob/master/src/realm/group_writer.hpp

[6] transition manager https://www.techopedia.com/definition/24043/transaction-manager

[7] Realm Wiki https://en.wikipedia.org/wiki/Realm_(database)#History

[8] initial commit https://github.com/realm/realm-java/commit/b03c621431fdc7e6e43566eeef505a32f5f6ce83

[9] contributors https://github.com/realm/realm-java/graphs/contributors

[10] pulse https://github.com/realm/realm-java/pulse/monthly

[11] customers https://realm.io/customers/

[12] roadmap https://realm.io/blog/sharing-the-mongodb-realm-roadmap/

[13] contributing https://github.com/realm/realm-java/blob/master/CONTRIBUTING.md

[14] conduct https://realm.io/conduct