Chapter 8
Realm

Phil Cobley and Ginger Geneste

Abstract In this chapter, we explore some of the fundamentals of the Realm database (sometimes referred to as RealmDB or simply Realm). It is widely known within the Digital Forensics discipline that SQLite is the most commonly found database format within any mobile device application and even some desktop applications. Realm is a relatively new database format built as a potential replacement for SQLite, as technology and applications continue to develop and evolve. At the time of writing, it is clear that the database is not as commonly found as some might have expected, but that is not to say the database format will not eventually find its way into many modern apps over the coming years. To that end, we decided to research the database to try and provide some of the details of interest relating to the fundamentals behind the new format. We hope this chapter will help digital forensic examiners and investigators learn and grasp some of the basic concepts of Realm, hoping that any new knowledge and understanding might support and assist in future research into the topic.

8.1 Organisation of this Chapter

You will find a chapter dedicated to the SQLite file format within this book. We shall be covering some of the basics behind SQLite for ease of readability and not assume prior knowledge. We look at some of the differences between SQLite and Realm before looking at Realm in more depth. We want to highlight that the Realm code is under constant development and is still in the early stages of that development,

Phil Cobley
MSAB, 2nd Floor East, Central Point, 25-31 London Street, Reading, RG1 4PS, UK e-mail: phil.cobley@msab.com

Ginger Geneste
Netherlands Forensic Institute, Laan van Ypenburg 6, 2497 GB Den Haag e-mail: g.geneste@nfi.nl

C. Hummert, D. Pawlaszczyk (eds.), *Mobile Forensics – The File Format Handbook*,
https://doi.org/10.1007/978-3-030-98467-0_8

meaning there are some limitations as to what can be confirmed both in the short and long term.

We have broken the chapter down into sections to help readers navigate through a journey of discovery, starting with some high-level and generic concepts, eventually drilling down into more detail later on. Each section builds on the last, but we have tried to write this chapter so that you can easily use the content as reference material if needed.

We start by looking at some of the similarities and differences with SQLite, exploring the concept of object-oriented database design and development over traditional relational tables and SQL queries. We then move into looking at how Realm works and how data is structured similar to a table-like format, but without actually creating any tables. We detail some object-oriented concepts before exploring the Realm files and their data structures in far greater detail. We discuss the concept of data and reference arrays and how these play an important part in Realm databases and break down the various file and array headers at the byte level.

We use example files in the chapter that can be created or downloaded, with the links available within the text itself, if you wish to either create those files or download them yourself. We use those files to break down a Realm array, looking at the offset pointers, the header, and how we can examine the data to identify the size of the payload and the type of data each array contains.

8.2 Introduction

In Digital Forensics, we are often interested in collecting and reviewing data held in databases. Most applications and operating systems rely on databases to store, organise and manage their data instead of swathes of unconnected files and unsorted data blocks. Databases make storage and retrieval simple and provide standardised mechanisms and schemas that modern applications can easily harness.

Within mobile device forensics the most commonly found database type used by applications is SQLite [78], which is a cross-platform, serverless database type, that has become a valuable tool for mobile app and operating system (OS) developers over recent years. The SQLite database was designed to be simple to use, easy to connect to applications across any platform, and could be installed and run upon the client device without the need for a bulky, backend server [83].

However, as devices, applications, and our usage needs of mobile devices evolve, so do the databases harnessed by applications. SQLite is a powerful database format, but it has its limitations. Often, modern-day developers are forced to generate and write additional code to enable their applications to do what they need them to do due to the emerging limitations present within SQLite. This additional code often involves implementing workarounds to enable natively unsupported data values stored within the SQLite tables.

This chapter seeks to explore the Realm database format [71], which has emerged over recent years as a possible successor to the now ageing SQLite format, looking at how this database format plugs those emerging gaps. Understanding how this database structure differs from SQLite should enable forensic examiners to understand better the types of data they are likely to encounter and appreciate how the format works in practice. For example, Realm databases do not use relational tables, a core feature within SQLite databases, but instead, work with linked objects. How does this impact forensic analysis? In this chapter, we shall look at what artefacts we can expect to find when a Realm database has been used and clarify what data may be found within.

! Attention

It is worth noting that research into this subject is still ongoing, and so, while this chapter seeks to explore how Realm databases work, it is not a comprehensive deep-dive into the full workings and data structures. This chapter may expand and evolve in future revisions of this book. However, we have ultimately tried to incorporate as many confirmed findings and factual content as possible at the time of writing. You will see that we have included enough for researchers to understand the fundamentals of these data structures and for forensic examiners to decode various headers and attributes, and we hope that this is the strong starting point to encourage and support examiners in taking this research further.

8.3 SQLite, It is Not!

While Realm might be replacing SQLite in some applications, the way they are coded and operate differ greatly. In order to understand how the object-oriented approach of the Realm database structure works, we will first go through an introduction to the more common relational database structure. We will then explore the concepts of an object-oriented approach for database structures, comparing its features with that of a relational database.

8.3.1 Relational Databases

There are many ways to define what a relational database is. However, in essence, it is a method of organising data into tables, which are linked together through common criteria or data components [38]. Data is typically organised into rows and columns, with each row being assigned a unique identifier. Tables can then reference data in other tables through the use of these unique identifiers, which is the “relational” aspect of the relational database concept.

As a simple example of how this might work, imagine we wish to use this concept when looking at grocery shopping. When grocery shopping, we may wish to search for the items we need by searching under specific categories, such as fruit, vegetables, meats, bakery, and so on. In a relational database setup, it may be that these are each represented as different tables, each containing the various items that you might find under that category (see Table 8.1).

Table 8.1: Example Grocery Tables

Fruit	**Price**
Apple	0.2
Orange	0.3
Pear	0.15
Banana	0.25

Vegetables	**Price**
Carrots	0.11
Potatoes	0.15
Cabbage	0.2
Cauliflower	0.3

Bakery	**Price**
Bread	0.8
Rolls	0.4
Wraps	0.95
Bagels	1.6

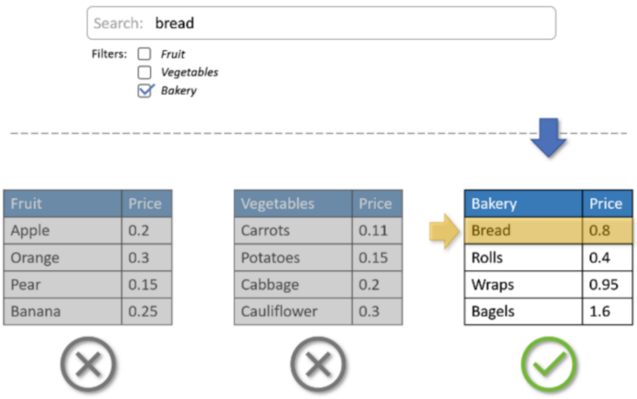
Structured Query Language (SQL) [39] is often used as a standardised language to both write data to and query data from such databases that support it. It has enabled developers to quickly and easily write and format queries and code to edit and pull data from relational databases through a global standard. Tables are connected to one another through functions known as “Joins” with search queries generating combined results sets in newly created tables containing various record (row) content, depending on the query made.

When carrying out searches across data sets, such as those found in our grocery example, SQL may be used in the background of a website or application to run queries across the tables, utilising search terms input by the user. This may include filters we commonly see on websites to narrow down the search. For example, we may be looking for a loaf of bread and therefore click on a "Bakery" filter and search for the term “bread” Fig. 8.1. SQL may then be used in the background to search for row items containing the keyword “bread”, but only within the “Bakery” table. Equally, applying no filters may conduct the search across all tables, thus conducting a wider search.

When searching such as this, there may also be an empty table either created or already available, that is used to hold copies of the search results. The column structure would likely be very similar to have compatibility with the existing data sets from the other tables but may have additional columns specific to a search. You could think of this table as possible where a typical search results page on a website may be drawing data from.

There are obviously countless ways to develop and programme these structures, and so this is just one (very simplistic) possible example of how data may be held, linked, and manipulated within a relational database. However, you will often have tables of fixed data content that are used as a reference point for the application. You will also have other tables populated and edited through user interaction or system processes, containing live or deleted content.

If we use our grocery example from a forensic standpoint, it might be that the grocery item tables are of little interest to us. However, the search table, or possibly

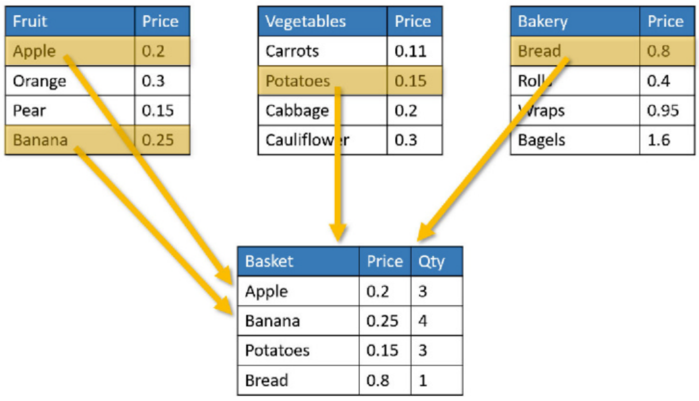


Fruit	Price
Apple	0.2
Orange	0.3
Pear	0.15
Banana	0.25

Vegetables	Price
Carrots	0.11
Potatoes	0.15
Cabbage	0.2
Cauliflower	0.3

Bakery	Price
Bread	0.8
Rolls	0.4
Wraps	0.95
Bagels	1.6

Fig. 8.1: Background Table Searches



Fruit	Price
Apple	0.2
Orange	0.3
Pear	0.15
Banana	0.25

Vegetables	Price
Carrots	0.11
Potatoes	0.15
Cabbage	0.2
Cauliflower	0.3

Bakery	Price
Bread	0.8
Rolls	0.4
Wraps	0.95
Bagels	1.6

Basket	Price	Qty
Apple	0.2	3
Banana	0.25	4
Potatoes	0.15	3
Bread	0.8	1

Fig. 8.2: Relational Table Results Collated in a New Table

a “Basket” table that is populated by the user when they add items to their basket might be of forensic interest as it would help identify user activity.

8.3.2 SQLite as a Relational Database

SQLite uses a relational database structure, storing all of the data tables and links within a single file, usually with a file extension similar to *.sqlite or *.db, although this can vary depending on the intended software platform and how the developers decide to build their applications (in mobile devices you will often find them with no file extension at all). This use of a simple, single, self-managing file, is what has helped make SQLite so popular with mobile developers, as there is no need to rely on additional backend servers, and the data becomes self-contained and very portable. These days we see mobile devices with huge storage capacities, but back when smartphones were first coming onto the market storage was still at a premium. SQLite databases enabled developers to build database storage into their mobile applications without taking up very much space on a user's device, and without the need to install additional software or depend on addition software or services running in the background, taking up valuable processing capacity, memory, or network bandwidth.

SQLite is also a cross-platform file format, which means it can be run and used on any common or major operating system, reducing the need for developers to concern themselves with huge changes in their application architecture when developing for multiple platforms, such as Android and iOS.

Given that SQLite utilises SQL as a query language, and given both SQL and SQLite are platform agnostic, it means that a software developer building an application in Swift for iOS will most likely utilise the same or similar SQL queries to a software developer building the same application in Java or Kotlin for the Android operating system. This, in turn, means that the structure, layout, and logic of the backend database does not have to change very much from one system to the next, allowing app developers to focus on only having to adapt the code that surrounds the database when building for different systems, rather than having to also be concerned with what database to use and how to update, edit, and retrieve data from within it.

In digital forensics this is great news for examiners and investigators, as most of the time we only need to concern ourselves with the content of the database – and for a standardised database format such as SQLite that is found on both iOS and Android, it means we only need to learn how to interrogate one database format, regardless of what type of device that data resides within.

8.3.3 SQLite Schema

A schema is essentially a specification confirming and defining the structure of a database, usually written or presented within the appropriate language or format for that database type. SQL developers define their database schema using what is known as Data Definition Language [77] which is used to create tables and define the types of data that each column should hold, such as integers, dates in specific formats, strings, and so on, as well as stating what columns can or cannot contain NULL values (see Fig. 8.3).

8.3.4 Temporary SQLite Files

Something that we commonly find alongside SQLite databases are the shared-memory (SHM) and write-ahead log (WAL) files, that we may find located in the

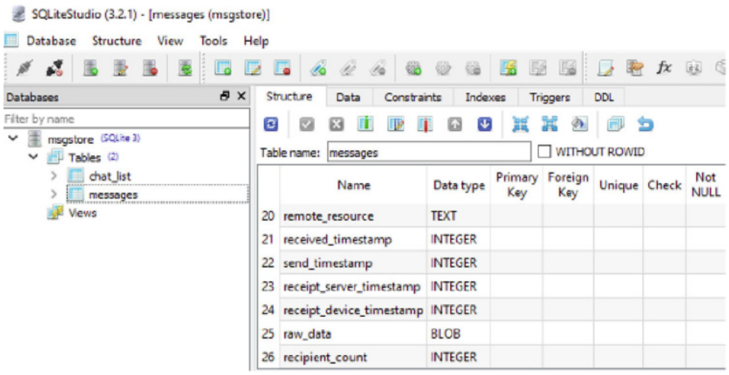
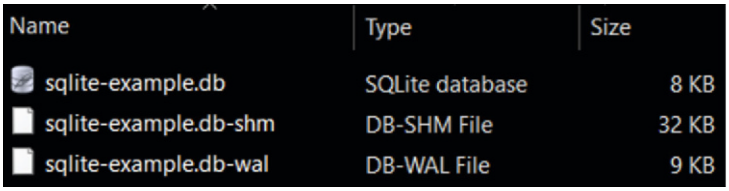

SQLiteStudio (3.2.1) - [messages (msgstore)]
Database Structure View Tools Help
Databases
Filter by name
msgstore (SQLite 3)
Tables (2)
chat_list
messages
Views
Structure Data Constraints Indexes Triggers DDL
WITHOUT ROWID

Table name: messages

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL
20	remote_resource	TEXT					
21	received_timestamp	INTEGER					
22	send_timestamp	INTEGER					
23	receipt_server_timestamp	INTEGER					
24	receipt_device_timestamp	INTEGER					
25	raw_data	BLOB					
26	recipient_count	INTEGER					

Fig. 8.3: SQLite Studio Displaying SQL Schema

same directory location as the SQLite database file [86]. There are actually several more additional temporary files that are sometimes found as well, but we will not go into detail of how these work here as they are explained in more depth within the SQLite chapter. However, we will touch on what the commonly found WAL and SHM files do and why they (sometimes) exist, for our later comparison.



Name	Type	Size
sqlite-example.db	SQLite database	8 KB
sqlite-example.db-shm	DB-SHM File	32 KB
sqlite-example.db-wal	DB-WAL File	9 KB

Fig. 8.4: SQLite Files, including SHM and WAL

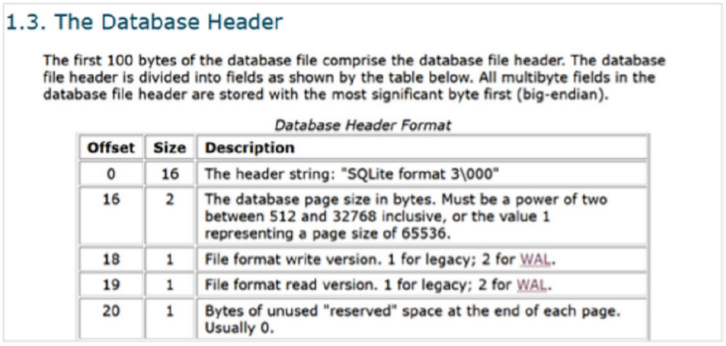
The WAL file is a form of journal that stores updates and changes to the SQLite database file. Rather than writing changes directly to the database file itself, which could cause complications and problems in numerous possible implementations of the database, those changes are written to the WAL first. Multiple changes and amendments can be made (resulting in duplicate entries within the WAL), pending the appropriate trigger to “Commit” those changes in the WAL. This committed content only gets written to the database itself once a “checkpoint” is triggered, where the most recent versions of all the changes and amendments are then transferred across to the database itself. Those triggers vary depending on the version and implementation of SQLite, but occasionally the commit does not take place during a single session, and may only be triggered when the database is reopened and accessed later on. This means that the WAL file can persist in a file system even after a connected application is closed, and it is not actually uncommon to find the WAL file to be even larger in size than the database itself.

The SHM is a file created to help manage concurrent connections to the database and allows the WAL to use a specified area of memory for indexing and managing the various changes and commits being made to the database. In essence, if you have multiple system services or process threads access the SQLite database file at the same time (which is very common) then an SHM file will be created to help service those connections. If one is created then it will typically persist on disk with the WAL file until the WAL file is deleted.

Why are these files important? Well, it is not uncommon for forensic analysists to find evidence and vital data buried within these temporary files, rather than the data being within the main database file itself. We shall see how this differs within Realm databases later on.

8.3.5 SQLite File Format

SQLite is an open-source file format with a very distinct and well documented structure. A lot of digital forensic training and education programmes will teach examiners about the specific layout and structure of the SQLite database header, as understanding the various attributes and byte values can be invaluable, particularly when dealing with more complex and challenging forensic scenarios where the database cannot be automatically parsed and decoded, albeit these instances are rare due to the wide range of comprehensive forensic software tools available. The header information can be found at: `https://www.sqlite.org/fileformat.html`, (see Fig. 8.5) [82]).



1.3. The Database Header

The first 100 bytes of the database file comprise the database file header. The database file header is divided into fields as shown by the table below. All multibyte fields in the database file header are stored with the most significant byte first (big-endian).

Database Header Format

Offset	Size	Description
0	16	The header string: "SQLite format 3\000"
16	2	The database page size in bytes. Must be a power of two between 512 and 32768 inclusive, or the value 1 representing a page size of 65536.
18	1	File format write version. 1 for legacy; 2 for WAL.
19	1	File format read version. 1 for legacy; 2 for WAL.
20	1	Bytes of unused "reserved" space at the end of each page. Usually 0.

Fig. 8.5: Screenshot from sqlite.org File Format Webpage

You can visit the referenced website to find the full database header format content, along with explanations and documentation around each offset specification.

8.4 How Realm Works

8.4.1 Realm Database Fundamentals

Realm is described on the realm.io website as:

"... an open source, developer-friendly alternative to CoreData and SQLite. Start in minutes, port your app in hours, and save yourself weeks of work." [71]

The database itself is an object database as opposed to a relational database, harnessing the principles of object-oriented programming over traditional database models such as SQLite. This approach allows the database to benefit in ways that are not possible in SQLite, such as having a zero-copy architecture and a near endless possibility to handle, store, and manipulate almost any file format or data type with ease.

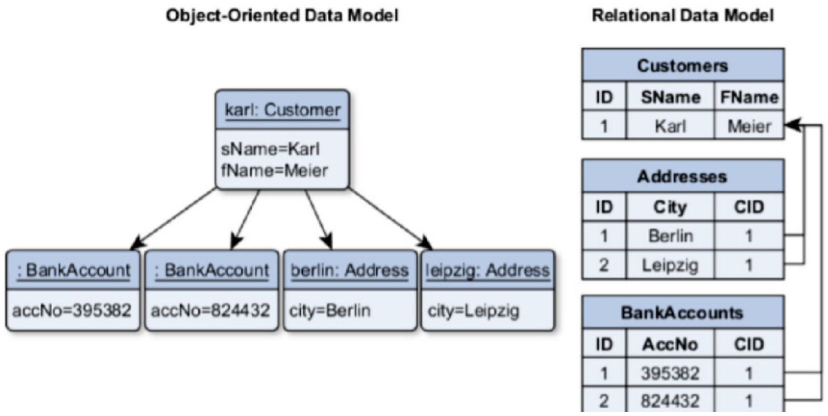
As we already know, relational databases, such as SQLite, consist of tables that join and work together to reference various data sets, locating and identifying records and data by navigating rows and columns. Sometimes the table connections (joins) can become incredibly complex, often requiring tables built specifically to hold unique reference variables to help tables navigate to and reference one another. Any queries that are run have their results copied into another table, duplicating data content for the purposes of generating query results. This can be time consuming, memory and processor intensive, and take up considerable data storage as databases grow and expand. The sheer act of copying data out of tables to represent the same content within another table could also be seen as being inefficient and unnecessary.

In their paper "Evolution of Object-Oriented Database Systems" [2] Alzahran compares traditional relational data models with object-oriented models, discussing the future of database structures and a shift in the current paradigm. An example of an object-oriented model (such as Realm) and relational data model (such as SQLite) is given in Fig. 8.6.

In the example presented in Fig. 8.6, we can see how data tables in a relational model are replaced with object instances of different classes, with a new object instance being created instead of a new row being added to a table. The columns in a table are now represented through object attributes, meaning that in order to locate data the object instance is queried and asked to return the attribute values, rather than tables being queried through SQL expressions.

8.4.2 Common Concepts and Terminology

Here we shall define and provide an overview for some common concepts and terminology used within Realm database architecture.



Customers

ID	SName	FName
1	Karl	Meier

Addresses

ID	City	CID
1	Berlin	1
2	Leipzig	1

BankAccounts

ID	AccNo	CID
1	395382	1
2	824432	1

Fig. 8.6: Screenshot from sqlite.org File Format Webpage

Basic Object-Oriented Programming Concepts

In object-oriented programming, used in languages such as Java and Python, there are a number of concepts and principles that are considered as very important when it comes to software design and development [16]. One of these concepts is known as “low coupling, high cohesion” and another being the concept of having small, well-defined objects that do specific jobs, rather than large, bulky objects that carry out multiple tasks.

The reason behind these concepts helps promote code design that is flexible and dynamic as well as being easily maintainable and adaptable. By having small, single function objects, you can build code where any other objects requiring that function simply call those specific object instances, rather than running the risk of duplicating the function within many larger objects. While singular, self-sufficient objects may seem like a good idea, it makes the code more difficult to maintain and manage. For example, say a specific function requires updating; if it is duplicated within multiple object types then they all require updating to ensure no objects are running with legacy code. However, if you have a single object for that function that other objects call upon to use, then simply updating that one object function will update the capability for all connected objects with minimal updates being required. This concept is sometimes referred to in other industries as “single-source” and relates to many different practices, not just software development.

The concept of low-coupling and high cohesion links into this through design principles in software development that suggest objects should work well together (high cohesion) but remain independent so that changes to one do not negatively impact the other (low-coupling). This allows developers to design code that is more easily maintainable and resilient to change. If objects can work well together and have a means to communicate without being dependant on exactly “how” the code has been implemented, then when that code needs changing or updating, so long as the communication methods remain in place, other objects remain unaffected.

Realm databases are able to leverage these benefits through an object-oriented approach to the database design, where the application creates and maintains lightweight, connected object instances as opposed to bulky, rigid, relational table structures. In SQLite, complex systems and database designs often rely upon queries that fully understand and recognise exactly what they're looking for, where, and how. The benefit this brings forensic examiners is that those queries and table structures are relatively straight-forward to reverse engineer and piece together, given how explicit the calls and queries often have to be. In Realm, this is not necessarily the case, as the database queries and calls are highly dependent upon how the developer has decided to implement the Realm database objects and instances, how they have coded the various communication methods and object attributes, and how complex the communication structure ends up becoming. The way this often happens is through objects communicating in a chain, from one to the next, making singular queries to one another, rather than a single, large, complex query statement across multiple tables (Fig. 8.7).

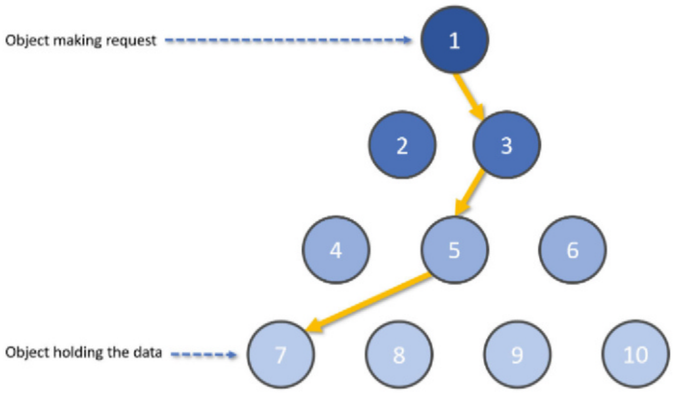


Fig. 8.7: Object Instance Communication

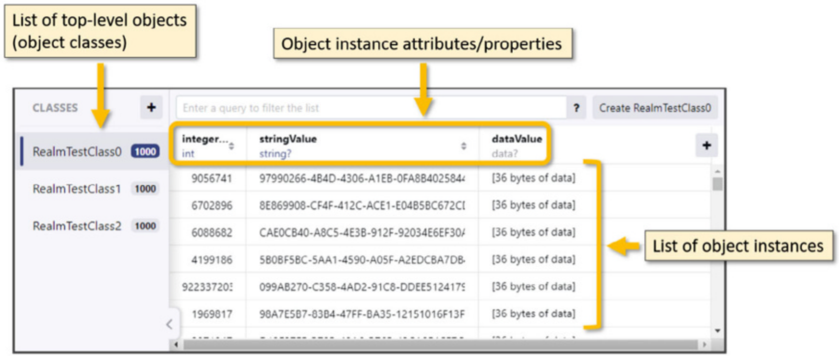
Top-level Objects

Top-level objects could be considered the equivalent of a relational table within an SQLite database. They are typically found to be object classes, such as "Fruit" or "Vegetables", similar to our example earlier in section 8.3.1.

In the following example we use a screenshot from a demo file available for download from the realm.io website [72] viewed within the official free Realm Studio tool [74], used for testing and training purposes to demonstrate some of what Realm can do.

In the screenshot you can see:

- Listed object classes – these are equivalent to a relational table
- A list of object instances – these are equivalent to the rows/records in a table
- The defined object properties/attributes – these are equivalent to table columns



integer... int	stringValue string?	dataValue data?
9056741	97990266-4B4D-4306-A1EB-0FA8B402584	[36 bytes of data]
6702896	8E869908-CF4F-412C-ACE1-E04B5BC672CI	[36 bytes of data]
6088682	CAE0CB40-A8C5-4E3B-912F-92034E6EF30	[36 bytes of data]
4199186	5B0BF5BC-5AA1-4590-A05F-A2EDCBA7DB	[36 bytes of data]
92233720	099AB270-C358-4AD2-91C8-DDEE5124179	[36 bytes of data]
1969817	98A7E5B7-83B4-47FF-BA35-12151016F13F	[36 bytes of data]

Fig. 8.8: Realm Top-level Object Structure Example

Object Types

It may sound obvious to some, but within Realm databases, an Object Type is a term used to help define exactly what any given object is recognised to be by the database. Object Types link to the database schema, which is defined in code by the database developer and works in a similar way to that of a database schema within SQLite. However, this is one area where Realm goes beyond the capabilities of SQLite in its capabilities.

In SQLite schemas column values are restricted to predetermined values, such as string or integer values, with a finite list of available types that are coded into the SQLite codebase. Anything that does not fit within this predefined list is typically managed through the use of BLOB data, which stands for Binary Large Object [49]. BLOB data can be almost anything that a developer wishes to include, but the data management can sometimes be complicated, and sometimes requires encoding. Furthermore, the database itself will usually not be able to determine exactly what the BLOB data represents, as this has to sometimes be managed through additional software components or tools outside of the database environment.

Within Realm , as the entire schema can be developed alongside the object code, the "type" values can be absolutely anything that the developer wishes to include or use, so long as the type represents a coded object class. This means that proprietary data objects can be built within the Realm code, teaching the database how to handle, manage, store, and manipulate that data natively, regardless of what it is or how it is constructed.

This can be potentially very powerful for database and application developers, but prove to be a huge challenge to forensic examiners and investigators, as it may require reverse engineering the original code in order to understand any proprietary or custom object types or formats. The MongoDB documentation for Realm has an example schema which shows possible schema code for storing data about books in libraries, where “Library” and “Book” are object types [49]:

```
[
  {
      "type": "Library",
      "properties": {
          "address": "string",
          "books": "Book[]"
      }
  },
  {
      "type": "Book",
      "primaryKey": "isbn",
      "properties": {
          "isbn": "string",
          "title": "string",
          "author": "string",
          "numberOwned": { "type": "int?", "default": 0 },
          "numberLoaned": { "type": "int?", "default": 0 }
      }
  }
]
```

Every object within a Realm database must be of a type that is validated by the schema and properly defined. It is worth noting, given Realm is based on the concept of everything being an object instance, that the Realm database itself is an object of type “realm”. When a Realm database file is opened and accessed an instance of the Realm database is initialised, with the relevant attributes and properties loaded into that instance from the stored data.

Group

When a Realm database is accessed or opened, the schema is read and interpreted to begin validating and initialising the appropriate object instances. Groups are collections of top-level objects (so, the equivalent of a collection of tables in SQLite) which together help identify and clarify the schema requirements.

Essentially, whenever a Realm database file is accessed through one of the Realm database SDK’s, the file is verified and loaded into a Realm object by calling the Realm Group [73].

Arrays

Realm databases predominantly store their data within data arrays, and so first we shall take a quick look at what an array actually is.

It is probably fair to say that almost every programming language can implement arrays in some form, and whilst their implementation may differ depending on the language, the concepts behind them remain fundamentally unchanged. They are simply a data structure that can be used to store an ordered collection of data within a single programmable component. What does that mean, exactly? Let us use an example to explore the answer to that question. Imagine you are programming a simple application and have decided you want to assign variables to hold the names of people who are attending an event. Now, there would be hundreds of different way to do this, and some more efficient than others, but this is just a simplified example to help with our understanding.
You may decide to programme individual variables, maybe something similar to the following:

```
attendee1 = "John"
attendee2 = "Sarah"
attendee3 = "Sam"
```

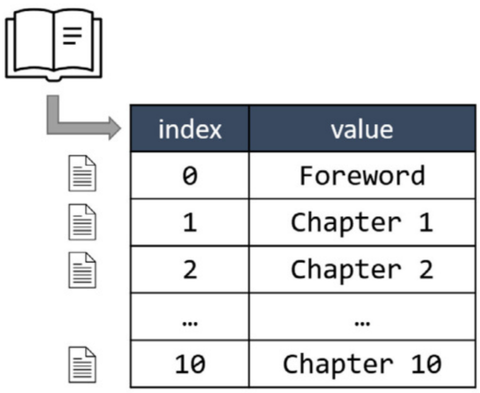
This might work really well for the first few attendees, but when you expand your system to a thousand, it may begin to get tedious and time consuming, not to mention a huge amount of code. Instead, you may consider an array. This component allows you to define a certain number of elements which are automatically numbered sequentially as the values are added to the component, so it is similar to a table that has two columns and a finite number of rows. The first column is automatically determined by the array and grows sequentially from 0 upwards, and the second column is for the data you wish to assign to each row. This allows you to forget about needing to code variables for each user, and instead lets you simply add the names directly to the array, so the array might be conceptualised similar to Table 8.2.

Table 8.2: Conceptual Array

index	**value**
0	”John”
1	“Sarah”
2	“Sam”
...	...
1000	"Yvette"

The order of adding values is very important with an array, as the indexes are filled sequentially to ensure efficiency, so no gaps are purposefully left. Arrays make it very easy for the software to locate specific data values as the index can be used to locate the data quickly. However, arrays do not care about the order in which they store their data, which differentiates them from other similar data structures where ordering and sorting is an important part of their function.

Another way of looking at arrays is like the chapters in a book. The chapters in a book are fixed and will not change, and the book provides a way for the reader to use the chapters to locate the information of interest to them. In this example, the book is the array and the chapters are the indexes and their data values (Fig. 8.9).



index	value
0	Foreword
1	Chapter 1
2	Chapter 2
...	...
10	Chapter 10

Fig. 8.9: Array Book Analogy

A Realm database file consists of a file header and is followed almost exclusively by 'Realm Arrays'. If the database is not encrypted then these arrays can be located and identified with a hex viewer and parsed out either manually or with appropriate scripting. There are two types of arrays found within Realm databases that we found through testing:

1. Arrays containing references to other arrays (referred to as a **Reference Array**)
2. Arrays containing data (referred to as a **Data Array**)

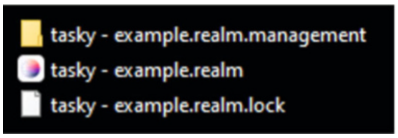
Essentially, a Realm database utilises what is known as a B+-Tree structure, where the tree can be recreated and mapped by following the pointers of the headers and reference arrays (the branches), until you reach the data arrays (the leaves). These arrays are all essentially nodes within that structure.

In the next section we shall begin exploring the structure behind some of these arrays, along with details of the Realm header and associated files that may be found with the *.realm database file.

8.5 File Storage and Structures

8.5.1 Realm Files and Folders

Here we are going to have a quick look at the files you may encounter when examining realm databases, including some of the temporary files that may be created, similar to how we sometimes find SHM and WAL files accompanying SQLite databases. Two core files will be commonly found with a Realm database, along with a folder that may be empty when recovered [48].



tasky - example.realm.management
tasky - example.realm
tasky - example.realm.lock

Fig. 8.10: Realm Files

Attention

We will not be covering Realm encryption in this book chapter, but it is worth being aware that they can be encrypted at source.

8.5.2 The Realm File

The most obvious file is the Realm database itself. The Realm database is referred to in documentation simply as a "**Realm**" which is a term that encapsulates the database and all associated files and data. The Realm is a single file that has a *.realm file extension, and contains all the generated data and associated objects. Developers are encouraged to initialise the Realm instance (create a Realm file) the first time an application is opened and run on a device, which means it may be possible to have realm-based applications that have been installed, but where no database is yet present if the application has not been run since installation. Realms can be encrypted by the developers, which would mean that static analysis may not be possible through standard tools without initial decryption taking place.

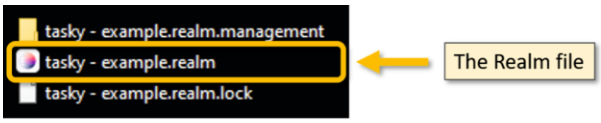

tasky - example.realm.management
tasky - example.realm
tasky - example.realm.lock

Fig. 8.11: The *realm File

The Lock File

The Lock file is created when the first connection is made, then recreated and reinitialised at the beginning of every session. This means that the file does not need to be present when the database is initially opened. The purpose of the file is to enable “synchronization between writes” [48] and even if deleted, will be recreated when the database is reopened.

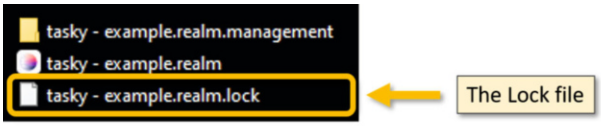

tasky - example.realm.management
tasky - example.realm
tasky - example.realm.lock

Fig. 8.12: The Lock File

A session is initiated and closed with the opening and closing of a Realm file via database objects. However, it also includes any sequence of temporarily overlapping openings of a particular Realm file via multiple database objects. For example, if there are two database objects, A and B, and the file is first opened via A, then opened via B, then closed via A, and finally closed via B, then the session stretches from the opening via A to the closing via B, rather than two individual sessions. This might be two different application instances opening the same Realm database file simultaneously, for example (like a multi-user session).

The Management Directory

This folder appears, like the lock file, when the database is opened and a connection established. Through testing we have yet to come across any files within the folder itself, but it is reported by MongoDB to “[contain] internal state management files” [48] which are likely to be of little interest within a forensic investigation.

Stateless Realm Instances

It is possible to create and run a Realm entirely within memory, resulting in no actual files being saved to persistent storage. In these instances no trace of any realm

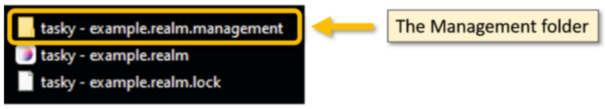

tasky - example.realm.management
tasky - example.realm
tasky - example.realm.lock

Fig. 8.13: The Management Directory

data will be located within extracted data storage but the data may be present within extractions of any volatile memory from an active device.

8.5.3 Creating Realm Test Instance

We are going to explore two different databases through this chapter. One is a demo file provided by Realm.io, which can be downloaded directly from their website or via a link within the Realm Studio software package:

- Realm Studio can be downloaded from the following URL: https://docs.mongodb.com/realm-legacy/products/realm-studio.html
- The Realm demo file can be downloaded from the following URL: https://static.realm.io/downloads/realm-studio/demo-v20.realm

The second database we are going to look at is a simple realm database created using Java within Android Studio [22] designed as a simple tasking app.

> Important

When writing this chapter we considered including a step-by-step guide on how to create a simple Realm database within Android Studio. However, we found, through our research, that constant changes to Android Studio, Java, Realm, and associated libraries and dependencies, meant that the guides would be out of date by the time they were published, with errata being required almost immediately. We stumbled upon a well-documented guide as written by developer Joyce Echessa, published in a blog article on behalf of auth0 [12] which we have used to build the Task app referenced within this section. This has been done so that you can follow the referenced web page and create your own database, if you wish. We found we had to update a number of referenced versions and dependencies, but overall the guide was still valid at the time of writing, and we were kindly given permission to include a reference to it within this book.

Upon creating our Task application within Android Studio we now need to run our task app for the first time to initialise the database and create a Realm instance. We then need to access our emulated device via the ADB (Android Debug Bridge) [21] to pull the newly created files out.

! Attention

If you are not familiar with ADB then then we encourage you to visit the android-studio documentation [1] to learn more about it and download the relevant software and tool packages. This walkthrough is completed using Windows 10, but you can achieve the same results on other operating systems.

Step 1: Launch the Task Application

From Android Studio, open up your emulated Android environment with your Task app present and load the operating system. Navigate to the applications list and you should see your Task app present (Fig. 8.14):

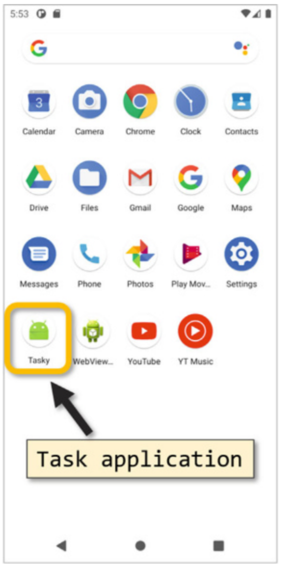


Fig. 8.14: ADB Walkthrough - Find Application

Now launch the application by clicking on the icon (Fig. 8.15) and then close it down:

[1] https://developer.android.com/studio/command-line/adb

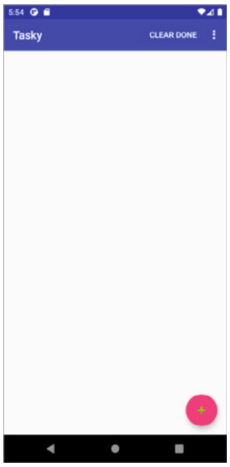
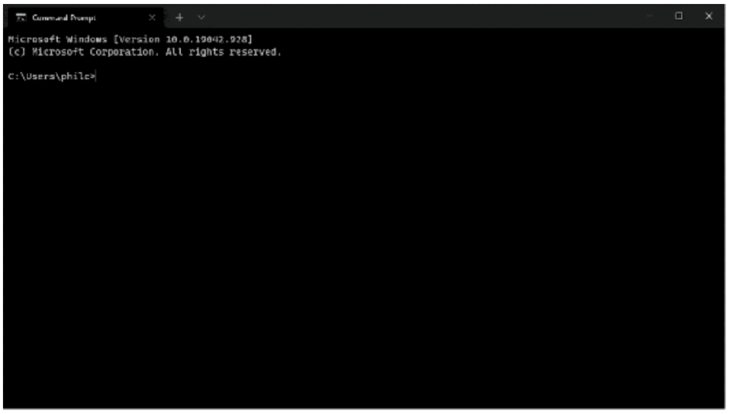


Fig. 8.15: ADB Walkthrough - Launch Task App

Step 2: Open a CMD Window

Open a CMD Command Prompt (or Powershell if you prefer), which can be done simply by opening your Start menu and typing "cmd", which will present the option to open a Command Prompt window similar to Fig. 8.16.



```
Microsoft Windows [Version 10.0.19042.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\philc>
```

Fig. 8.16: ADB Walkthrough - Open CMD Window

Step 3: Create an Output Folder

Create an output folder where your Android files will be placed. For this example we have created a folder called "Android" at the root of the Windows `C:\` drive as this will keep the commands in later steps, much smaller and easier to manage, but you can choose any location you like.

Step 4: Start ADB

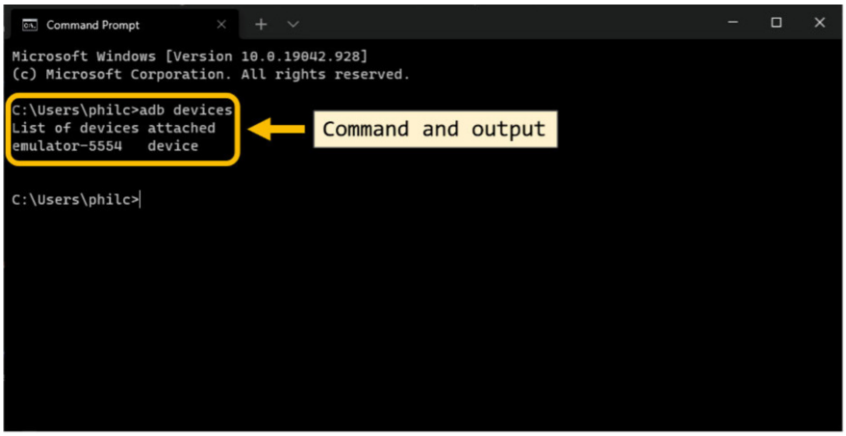
In your CMD window type the command:

```
adb devices
```

Important

This is assuming you have added ADB to your PATH. If not, we suggest you do this before proceeding.

What you should see is a list of devices attached to your computer via ADB (Fig. 8.17). Your emulated Android may have a different reference number or name, but you should see something similar to:



```
Microsoft Windows [Version 10.0.19042.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\philc>adb devices
List of devices attached
emulator-5554   device

C:\Users\philc>
```

Fig. 8.17: ADB Walkthrough - "adb devices" Command

This has confirmed that your emulated device is visible to your computer via ADB, and we can proceed with pulling the data from the device.

Step 5: Get ADB Root

This is only really going to work as we are emulating our Android device and simply using the content for research purposes. However, usually you would have to use additional steps to pull the application data from a modern Android handset due to permission restrictions and device security settings. However, in the interests of speed and simplicity, type in the following command to your CMD window:

```
adb root
```

This will provide you with root access to the device via ADB, meaning we can bypass a lot of the existing security and protections.

Step 6: Find the Application Data

Here we shall navigate through the device to confirm the location of the application data. We would expect the find the app package and associated directories, located within the file path:

```
/data/data/<package ID>
```

In this example I have named the app "Tasky" and it has a package ID of:

```
com.tutorial.tasky
```

Yours may be different, depending on how you build the app and what name you gave it, so just bear this in mind when looking for the package. First, in your CMD window type the following commands in order and press enter/return at the end of each one:

```
adb shell
```

Then type:

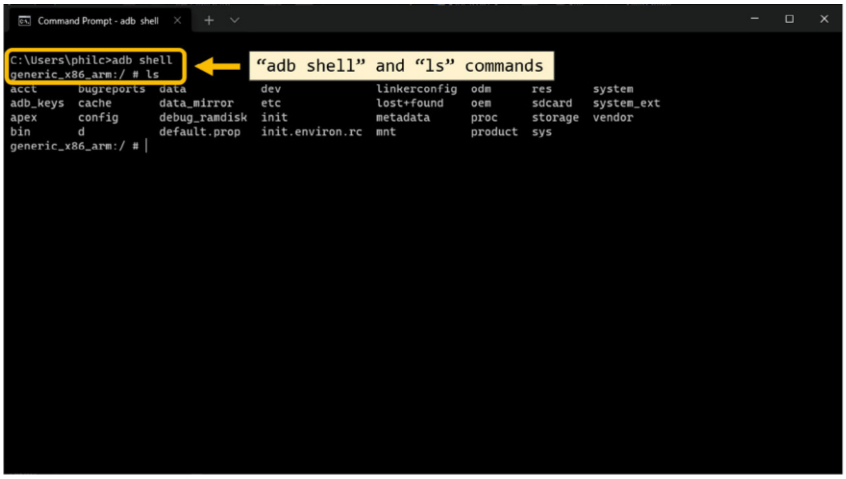
```
ls
```

Next type the command:

```
cd /data/data
```

This takes us to the data folder where all of the packages are located. We could have done two separate steps of running the command `cd /data` twice in succession as there are two directories called "data", one nested within another. However, the command we used combined both into a single command. Next, use the `ls` command to locate your application package:

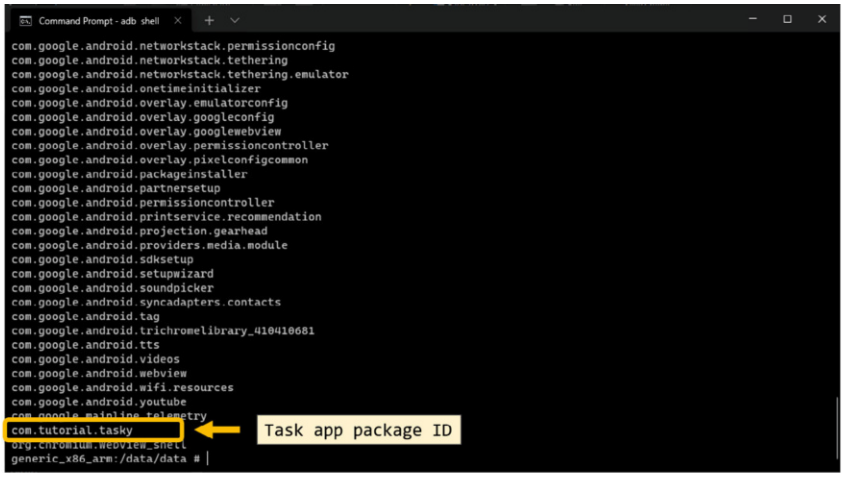
```
ls
```



```
C:\Users\philc>adb shell
generic_x86_arm:/ # ls
acct       bugreports  data           dev             linkerconfig  odm      res      system
adb_keys   cache       data_mirror    etc             lost+found    oem      sdcard   system_ext
apex       config      debug_ramdisk  init            metadata      proc     storage  vendor
bin        d           default.prop   init.environ.rc mnt           product  sys
generic_x86_arm:/ # |
```

Fig. 8.18: ADB Walkthrough - "adb shell" and "ls" Commands

The list of packages will be displayed, and you can look through the list to find your application. In this example, the app we have created is located near the end of the list (Fig. 8.19):



```
com.google.android.networkstack.permissionconfig
com.google.android.networkstack.tethering
com.google.android.networkstack.tethering.emulator
com.google.android.onetimeinitializer
com.google.android.overlay.emulatorconfig
com.google.android.overlay.googleconfig
com.google.android.overlay.googlewebview
com.google.android.overlay.permissioncontroller
com.google.android.overlay.pixelconfigcommon
com.google.android.packageinstaller
com.google.android.partnersetup
com.google.android.permissioncontroller
com.google.android.printservice.recommendation
com.google.android.projection.gearhead
com.google.android.providers.media.module
com.google.android.sdksetup
com.google.android.setupwizard
com.google.android.soundpicker
com.google.android.syncadapters.contacts
com.google.android.tag
com.google.android.trichromelibrary_410410681
com.google.android.tts
com.google.android.videos
com.google.android.webview
com.google.android.wifi.resources
com.google.android.youtube
com.google.mainline.telemetry
com.tutorial.tasky
org.chromium.webview_shell
generic_x86_arm:/data/data # |
```

Fig. 8.19: ADB Walkthrough - Locate Package ID

We have now confirmed that our app package directory is located at:

```
/data/data/com.tutorial.tasky
```

Now, exit the adb shell using this command:

```
exit
```

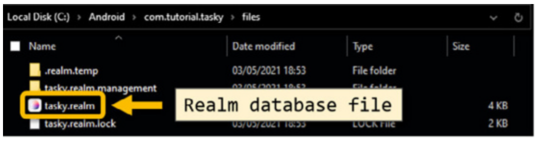
Step 7: Use the "pull" Command

In this final step we use the "pull" command to pull a copy of the package directory out from the device and place the copy into a location of our choice on our computer. In your CMD window type the following command:

```
adb pull "/data/data/com.tutorial.tasky" "C:\Android"
```

Here we have specified the command `adb pull`, and provided what are known as `parameters`. The pull command can recognise several parameters, and we have provided both a target for our pull action, as well as a destination of where to place the copied content.

You should now be able to open the destination location on your computer and find the exported copies of the package folders. Within these folders you will find your initialised Realm database file.



Local Disk (C:) › Android › com.tutorial.tasky › files

Name	Date modified	Type	Size
.realm.temp	03/05/2021 18:53	File folder	
tasky.realm.management	03/05/2021 18:53	File folder	
tasky.realm			4 KB
tasky.realm.lock	03/05/2021 18:53	LOCK File	2 KB

Fig. 8.20: ADB Walkthrough - Output Files

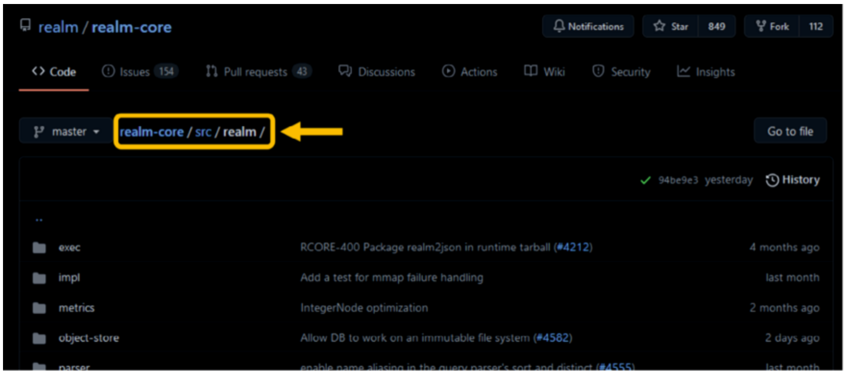
8.5.4 The Realm Database File Structure

As digital forensic examiners and investigators it is usually helpful to understand the inner workings of the artefacts that we analyse and decode. It is not always possible for us to fully reverse engineer these artefacts, but with research and testing we can often, as a community, begin to figure out some of the key and important hex strings and data blocks that reside within. With Realm this is no exception, and as the source code is publicly available, we have the benefit of being able to use this code to help identify how the database is structured at a byte level for some of those important components.

So far in this chapter we have discussed how Realm compares to traditional SQLite databases, how it differs, and have provided a brief overview of what to expect from a Realm database. We shall now take that understanding and dive deeper into the inner-workings of the files themselves, highlighting and identifying some key structures. We include this in the hopes that it can both direct examiners to relevant documentation to continue this research, as well as assist those who wish to begin creating scripts and other tools to begin parsing these databases themselves.

This section will guide you through the basic concepts of the Realm database file structure based on the implementation of the 'Realm Core' . The source code of the Realm Core implementation is available on Github at https://github.com/realm/realm-core [62]. At the time of writing, the source code referenced with this chapter relates to 'realm-java-v10.4.0'. The Realm Core is actively being developed which does mean that any static analysis of the source code may change over time.

Navigating to the Github repository directory `/src/realm/` we find many C++ source code files (*.cpp). We have analysed and researched some of these files to help identify some of the content for this section, identifying some structures and confirming some byte references and offsets. However, remember that this code is under active development, and therefore we advise examiners and investigators to validate and verify these findings, as is good practice, for all future versions of the source code.



..		
exec	RCORE-400 Package realm2json in runtime tarball (#4212)	4 months ago
impl	Add a test for mmap failure handling	last month
metrics	IntegerNode optimization	2 months ago
object-store	Allow DB to work on an immutable file system (#4582)	2 days ago
parser	enable name aliasing in the query parser's sort and distinct (#4555)	last month

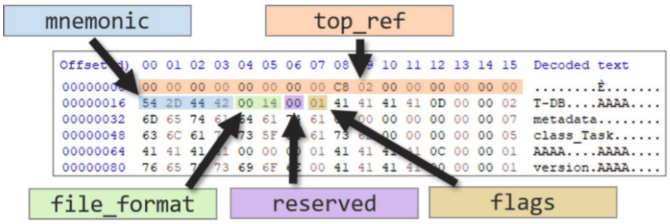
Fig. 8.21: Screenshot of the realm-core/src/realm/ directory

8.5.5 Realm File Header

Each Realm database contains a 24-byte header that can be broken down into component parts. We found the header to be defined within the file `alloc_slab.hpp` which is a form of header file, and can be found at https://github.com/realm/realm-core/blob/master/src/realm/alloc_slab.hpp [65]. The code relates to the definition of what is known as a struct, which is a C++ data structure where the term literally stands for "structure". It is used to store different elements of different data types within a fixed, structured environment, which is perfect for building a header of a set size and design. At the time of writing, the code was located at line 520 and reads as follows:

```
// 24 bytes

    struct Header {

        uint64_t m_top_ref[2]; // 2 * 8 bytes

        // Info-block 8 bytes

        uint8_t m_mnemonic[4]; // "T-DB"

        uint8_t m_file_format[2]; // See ‘library_file_format‘

        uint8_t m_reserved;

        // bit 0 m_flags is used to select between the two top
            refs.

        uint8_t m_flags;
    };
```

Based on this declared Header struct, the 24-byte header contains a reference to a ‘top ref’, the ‘mnemonic’, a ‘file format’, ‘reserved’ and ‘flags’. Each of these elements will be described below. However, in essence, the byte allocations are as follows:



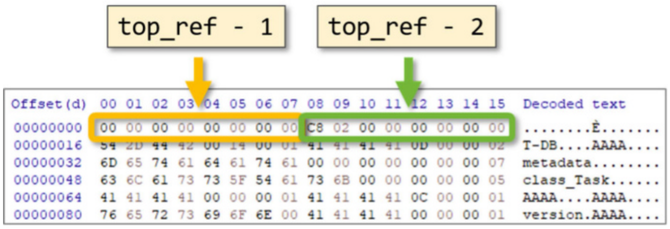
```
Offset(d) 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000  00 00 00 00 00 00 00 00 C8 02 00 00 00 00 00 00  ........È.......
00000016  54 2D 44 42 00 14 00 01 41 41 41 41 0D 00 00 02  T-DB....AAAA....
00000032  6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 07  metadata........
00000048  63 6C 61 73 73 5F 54 61 73 6B 00 00 00 00 00 05  class_Task......
00000064  41 41 41 41 00 00 00 01 41 41 41 41 0C 00 00 01  AAAA....AAAA....
00000080  76 65 72 73 69 6F 6E 00 41 41 41 41 00 00 00 01  version.AAAA....
```

Fig. 8.22: Realm Header Structure

- **16 bytes**: 2 x 8 byte references to the ‘top_ref’
- **4 bytes**: mnemonic / ‘magic value’
- **2 bytes**: file_format
- **1 byte**: reserved
- **1 byte**: flags - bit 0 is used to select between the two top_ref pointers

“Top Ref” - Bytes 0x00 to 0x0F (d0–d15)

The top_ref element stands for “Top reference” and relates to the root of the database. The element is sixteen bytes in length, using the first sixteen bytes of the database file, but is actually made up of two eight byte components (see Fig. 8.23). Both components are a top_ref but each eight byte string references an offset within the file, with the first referencing the start of the first top_ref, and the next eight byte string referencing the second top_ref.



```
Offset(d) 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000  00 00 00 00 00 00 00 00 C8 02 00 00 00 00 00 00  ........È.......
00000016  54 2D 44 42 00 14 00 01 41 41 41 41 0D 00 00 02  T-DB....AAAA....
00000032  6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 07  metadata........
00000048  63 6C 61 73 73 5F 54 61 73 6B 00 00 00 00 00 05  class_Task......
00000064  41 41 41 41 00 00 00 01 41 41 41 41 0C 00 00 01  AAAA....AAAA....
00000080  76 65 72 73 69 6F 6E 00 41 41 41 41 00 00 00 01  version.AAAA....
```

Fig. 8.23: Realm Header - top_ref

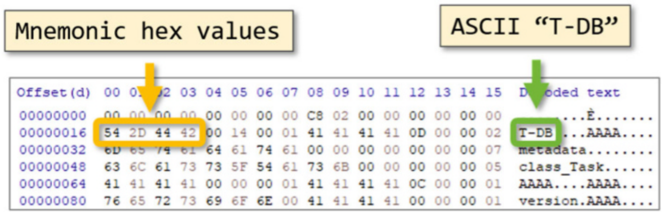
These references point to two separate arrays that act as the root nodes. These, in turn, point to two distinct branches that each link to a series of arrays, both branches seemingly mirrored (or almost mirrored). At the time of writing, our understanding based on testing suggests that the database utilises multiple branches through the top_ref mechanism as a form of journaling, alternating writes and commits between the two different root nodes. Starting at one of the two top_ref arrays, one can follow the references to other reference arrays to build up a tree. The current, most up to date state of the database is represented by rebuilding the Tree from the root node of the ‘current top ref’. The database identifies the current top_ref through the flag byte, and writes new data to the other top_ref, preserving the current branch of arrays. This could be thought of as a form of WAL, where the “current” top_ref is like an SQLite database file and remains untouched, but the other referenced root node is used to write changes prior to any commits.

However, similar to an SQLite database where WAL checkpoint rules do not always appear to be followed by the database itself, the rules governing changes with Realm arrays also appear to be fairly flexible and not always consistent.

“Mnemonic” - Bytes 0x10 to 0x13 (d16–d19)

These four bytes contain the ASCII value ‘T-DB’ which is called the ‘mnemonic’ of the Realm file. In other words, bytes 0x10 to 0x14 contain the magic value of the Realm file.

At the time of writing the mnemonic values are static and always equate to the ASCII “T-DB”. This may change with future iterations of the database code, but currently this is a very good way of being able to immediately identify the file as a Realm database, and enables examiners to utilise the hex string 0x542D4442 in searches when seeking to find Realm database amongst datasets.

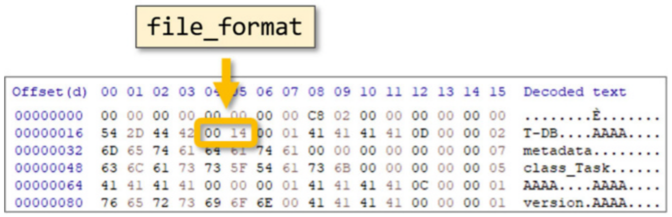


```
Offset(d)  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000   00 00 00 00 00 00 00 00 C8 02 00 00 00 00 00 00  ........È.......
00000016   54 2D 44 42 00 14 00 01 41 41 41 41 0D 00 00 02  T-DB....AAAA....
00000032   6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 07  metadata........
00000048   63 6C 61 73 73 5F 54 61 73 6B 00 00 00 00 00 05  class_Task......
00000064   41 41 41 41 00 00 00 01 41 41 41 41 0C 00 00 01  AAAA....AAAA....
00000080   76 65 72 73 69 6F 6E 00 41 41 41 41 00 00 00 01  version.AAAA....
```

Fig. 8.24: Realm Header - mnemonic

“File Format” - Bytes 0x14 to 0x15 (d20–d21)

These two bytes are described in the Realm core source code as the ‘file format’. Both bytes form an integer value and represent the version number.



```
Offset(d)  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000   00 00 00 00 00 00 00 00 C8 02 00 00 00 00 00 00  ........È.......
00000016   54 2D 44 42 00 14 00 01 41 41 41 41 0D 00 00 02  T-DB....AAAA....
00000032   6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 07  metadata........
00000048   63 6C 61 73 73 5F 54 61 73 6B 00 00 00 00 00 05  class_Task......
00000064   41 41 41 41 00 00 00 01 41 41 41 41 0C 00 00 01  AAAA....AAAA....
00000080   76 65 72 73 69 6F 6E 00 41 41 41 41 00 00 00 01  version.AAAA....
```

Fig. 8.25: Realm Header - file_format

In the documentation the file format is actually also referred to as the “version”, and in the `alloc_slab.cpp` file [63] there is actually reference to the variable `file_format_version` variable, along with an object method of `get_committed_file_format_version()` `get_committed_file_format_version()`. In this method, it was observed that the file format may be updated to 0x14 (d20) whenever the process (called ‘session’ in the Realm Core source code) accessing the Realm database requires so. This functionality is likely built in to provide compatibility with processes that handle newer/future file formats. For example, in the `group.cpp` file, the object method `read_only_version_check()` requires a file format upgrade if the `file_format_version` is lower than 0x14 (d20) when the database needs to be opened in read only mode. At the time of writing the maximum value for the file format was 0x14 (d20) but may be subject to change in the future.

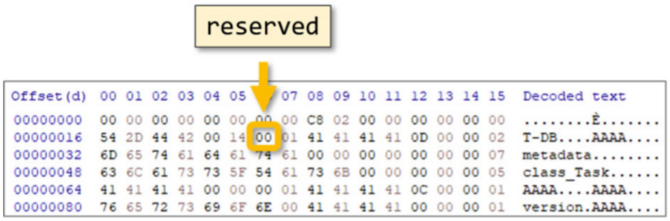
The file `group.cpp` of the source code contains an object method called void-Group::open(). In this method, the `target_file_format_version` is assigned to a value determined by `get_target_file_format_version_for_session()`, suggesting the desired file format version of the file itself is determined by the current process which calls `open()`. The `open()` object method only returns without errors if the `target_file_format_version` is 0 or equal to the file format version.

The file format version is assigned to 0 upon the creation of an empty database file where the Realm file header needs to be initialised. The initialisation of an empty header is defined in `alloc_slab.cpp` as follows:

```
const SlabAlloc::Header SlabAlloc::empty_file_header = {

    {0, 0}, // top-refs
    {'T', '-', 'D', 'B'},
    {0, 0}, // undecided file format
    0,      // reserved
    0       // flags (lsb is select bit)
};
```

“Reserved” - Byte 0x16 (d22)

The reserved byte, at the time of writing, is always set to zero, and has never changed throughout testing. This byte currently appears to be unused, as the name suggests, but may be utilised.



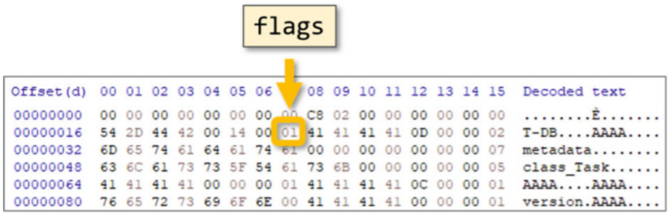
```
Offset(d) 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000  00 00 00 00 00 00 00 00 C8 02 00 00 00 00 00 00  ........È.......
00000016  54 2D 44 42 00 14 00 01 41 41 41 41 0D 00 00 02  T-DB....AAAA....
00000032  6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 07  metadata........
00000048  63 6C 61 73 73 5F 54 61 73 6B 00 00 00 00 00 05  class_Task......
00000064  41 41 41 41 00 00 00 01 41 41 41 41 0C 00 00 01  AAAA....AAAA....
00000080  76 65 72 73 69 6F 6E 00 41 41 41 41 00 00 00 01  version.AAAA....
```

Fig. 8.26: Realm Header - reserved

“Flags” - Byte 0x17 (d23)

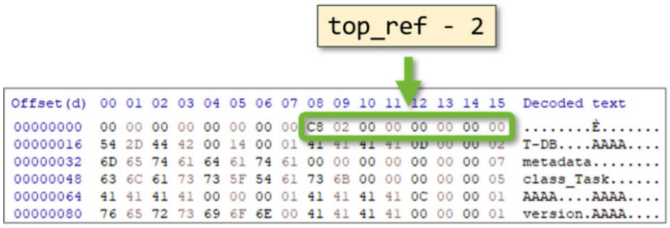
The final byte value of the header represents flags. The first bit (the least significant bit) of the last byte indicates which top reference is currently active. If this bit is set to 0, the first top reference is currently active and when this bit is set to 1, the second top reference is active. The other seven bits of the last byte are at the time of writing unused.

In Fig. 8.27 we can see that the byte value is 0x01, meaning that the last bit must be a 1 (the binary breakdown of 0x01 being b0000001), indicating that in this example the current referenced array branch is top_ref 2 (Fig. 8.28).



```
Offset(d)  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000   00 00 00 00 00 00 00 00 C8 02 00 00 00 00 00 00  ........È.......
00000016   54 2D 44 42 00 14 00 01 41 41 41 41 0D 00 00 02  T-DB....AAAA....
00000032   6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 07  metadata........
00000048   63 6C 61 73 73 5F 54 61 73 6B 00 00 00 00 00 05  class_Task......
00000064   41 41 41 41 00 00 00 01 41 41 41 41 0C 00 00 01  AAAA....AAAA....
00000080   76 65 72 73 69 6F 6E 00 41 41 41 41 00 00 00 01  version.AAAA....
```

Fig. 8.27: Realm Header - flags



```
Offset(d)  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000   00 00 00 00 00 00 00 00 C8 02 00 00 00 00 00 00  ........È.......
00000016   54 2D 44 42 00 14 00 01 41 41 41 41 0D 00 00 02  T-DB....AAAA....
00000032   6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 07  metadata........
00000048   63 6C 61 73 73 5F 54 61 73 6B 00 00 00 00 00 05  class_Task......
00000064   41 41 41 41 00 00 00 01 41 41 41 41 0C 00 00 01  AAAA....AAAA....
00000080   76 65 72 73 69 6F 6E 00 41 41 41 41 00 00 00 01  version.AAAA....
```

Fig. 8.28: Realm Header - top_ref 2 example from flag value of 0x01

8.5.6 Realm File Arrays

After the Realm header a *.realm file is made-up entirely of arrays in a B+tree format, with a single root node, inner nodes that act as sign posts, and leaf nodes that generally contain the data of interest to investigators.

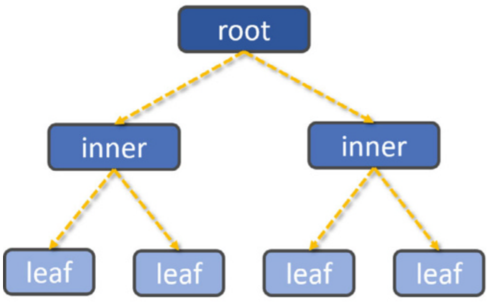
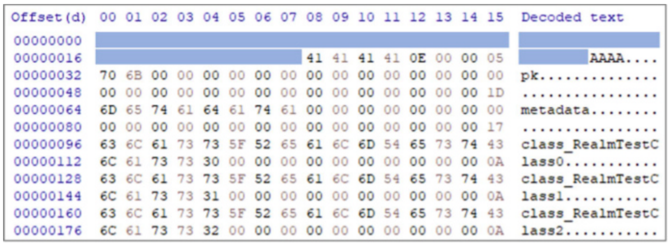


Fig. 8.29: Example Node Structure

The top_ref header information points to offsets within the file where you will find (typically) "reference arrays". These reference arrays point to other nodes and arrays within the database, some of which will be other reference arrays, and some will be what are known as "data arrays". These data arrays are where the actual core object data is stored.

Opening up a copy of the downloadable “demo.realm” file, available from realm.io, we will use the first array found after the Realm header, as an example of how arrays are structured and can be broken down (see Fig. 8.30).



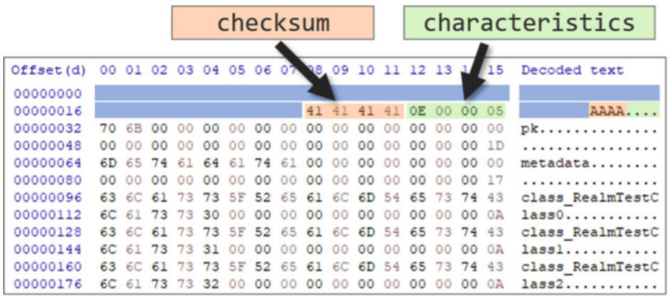
```
Offset(d) 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000
00000016                          41 41 41 41 0E 00 00 05          AAAA....
00000032  70 6B 00 00 00 00 00 00 00 00 00 00 00 00 00 00  pk..............
00000048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 1D  ................
00000064  6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 00  metadata........
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 17  ................
00000096  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000112  6C 61 73 73 30 00 00 00 00 00 00 00 00 00 00 0A  lass0...........
00000128  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000144  6C 61 73 73 31 00 00 00 00 00 00 00 00 00 00 0A  lass1...........
00000160  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000176  6C 61 73 73 32 00 00 00 00 00 00 00 00 00 00 0A  lass2...........
```

Fig. 8.30: Example Node Structure

8.5.7 Realm Array Header

Every array within Realm starts with an 8-byte header broken into two distinct parts:

1. Checksum value (4 bytes)
2. Array characteristics (4 bytes)



```
Offset(d) 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000
00000016                          41 41 41 41 0E 00 00 05          AAAA....
00000032  70 6B 00 00 00 00 00 00 00 00 00 00 00 00 00 00  pk..............
00000048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 1D  ................
00000064  6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 00  metadata........
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 17  ................
00000096  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000112  6C 61 73 73 30 00 00 00 00 00 00 00 00 00 00 0A  lass0...........
00000128  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000144  6C 61 73 73 31 00 00 00 00 00 00 00 00 00 00 0A  lass1...........
00000160  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000176  6C 61 73 73 32 00 00 00 00 00 00 00 00 00 00 0A  lass2...........
```

Fig. 8.31: Realm Array Header Example

This can actually be broken down further into the following components:

Table 8.3: Table of Realm Array Header Components

Header Section	**Offset**	**Size**	**Description**
Checksum	0x00 (0)	4 bytes	Checksum (dummy) (“AAAA” in ASCII)
Characteristics	0x04 (4)	1 byte	Flags
Characteristics	0x05 (5)	3 bytes	Size

8.5.8 Checksum

The first four bytes contain a checksum value that, at the time of writing, is only considered a “Dummy Checksum” within documentation, suggesting that this component may change in future iterations of the code. The checksum consists of four matching byte values, namely 0x41414141, which reads as “AAAA” in ASCII. The source code shows us that these four ASCII characters are used when scanning for arrays within the database [69].

8.5.9 Flags

The fifth byte of the array header represents flags and utilises several bit groupings to denote different configurations for the array [66]. This means that to read or parse the bit groupings we need to breakdown the byte into 8-bits and identify which bits represent which groups. The breakdown is as follows:

Table 8.4: Breakdown of “Flags” byte into bit groupings

Bit	**Group**	**Description**
1	1	is_inner_bptree_node
2	2	has_refs
3	3	context_flag
4 5	4	width_scheme
6 7 8	5	width_ndx

Bit Group 1: is_inner_bptree_node

The first bit of the flags byte indicates whether a Realm array is an 'inner node'. A value of 1 would indicate that the node is an inner node, which would mean that the array must be a reference array, as opposed to a data array. While analysing several Realm database files, none of the Realm arrays had the `is_inner_bptree_node` flag set. One reason for not using the `is_inner_bptree_node` flag in the Realm array header could be that the tree structure of the Realm database can still be constructed without checking this flag: starting from the root node (the top_ref array), and following all references of the tree until all identified references have been exhausted and pursued, allows a researcher to build the tree manually. Take a look at the code snippet below, which comes from the file `array.cpp`:

```
void Array::set_type(Type type)
{
    REALM_ASSERT(is_attached());

    copy_on_write(); // Throws

    bool init_is_inner_bptree_node = false, init_has_refs =
        false;
    switch (type) {
        case type_Normal:
            break;
        case type_InnerBptreeNode:
            init_is_inner_bptree_node = true;
            init_has_refs = true;
            break;
        case type_HasRefs:
            init_has_refs = true;
            break;
    }
    m_is_inner_bptree_node = init_is_inner_bptree_node;
    m_has_refs = init_has_refs;

    char* header = get header();
    set_is_inner_bptree_node_in_header(init_is_inner_bptree_node,
        header);
    set_hasrefs_in_header(init_has_refs, header);
}
```

It appears an array with the `is_inner_bptree_node` flag set, also sets the bit flag for `has_refs`. Therefore, it is concluded that any array that is considered an `inner_bptree_node`, also contains references to other arrays.

Bit Group 2: has_refs

A Realm Array header where the second bit of the flags byte is set to 1, indicates the Realm Array contains references to other Realm Arrays in its payload, making it a "reference array". This, in turn, makes the array the parent of any other arrays that it directly references. The payload of any reference array will typically consist of elements that store pointers to the child arrays. These pointers will be integer values that directly correspond to an offset in the file. These offsets always consist of 8-byte strings.

Bit Group 3: context_flag

The third bit within the flags byte is known as the "context_flag". However, rarely set to 1, and the full purpose of the flag remains unclear. Code from the `array.hpp` file [66] enables us to deduce that the context flag can be used to tell what type of leaf node the given array is. Unfortunately, there is not much more information currently shared regarding the 'context flag' in the Realm Core source code at the time of writing.

Bit Group 4: width_scheme

The array header contains information that enables us to calculate the total size, in bytes, of the payload held within the array. This calculation is done by using both the `width_scheme` and `width_ndx` bit groupings. We can therefore calculate the total size of the array by identifying the values held within these two bit groups.

The `width_scheme` consists of two bits which are added together to create a integer value. This type of calculation, therefore, allows for three possible value outcomes: 0, 1, or 2. Depending on the value, the payload is calculated in a certain way, as defined in `node_header.hpp` [67]. The following code snippet outlines the calculations and intentions:

```
static void set_wtype_in_header(WidthType value, char* header)
    noexcept
    {
        // Indicates how to calculate size in bytes based on
            width
        // 0: bits      (width/8) * size
        // 1: multiply  width * size
        // 2: ignore    1 * size

        typedef unsigned char uchar;
        uchar* h = reintercept_cast<uchar*>(header);
        h[4] = uchar((int(h[4]) & ~0x18) | int(value) << 3);
    }
```

Therefore, the `width_scheme` could be translated as per Table 8.5. The 'size' in the calculation is the value represented in the last three bytes of the Realm Array header. The value of 'width' in the calculation is represented in bit group 5, after applying the width translation table (see Table 8.6 below).

Table 8.5: Calculations Required for width_scheme Values

Value of width_scheme	**Meaning**	**Calculation for array payload**
0	Calculate size with number of bits	Cell(width*size/8)
1	Calculate size with number of bytes	Width*size
2	Ignore width in size calculation	size

Bit Group 5: width_ndx

Bits 6, 7, and 8 of the `flags` byte form ‘bit group 5’, referred to as `width_ndx`. These three bits collectively are used to represent the values 0 to 7 (7 being when all three bits are set to 1), and are used to indicate the value of ‘width’. With the translation table below, the translation from `width_ndx` to the actual value of ‘width’ can be found. The value “width” represents the number of elements that are contained within the Realm Array payload.

Table 8.6: Translation table for width_ndx

width_ndx calculated value	0	1	2	3	4	5	6	7
Value of ‘width’	0	1	2	4	8	16	32	64

8.5.10 Size

The size of the Realm Array payload is described in the last 3 bytes of the Realm Array Header. The size represents the amount of bits or bytes of one element in a Realm array. The value of `width_scheme` determines how the size value is used to calculate the overall payload. Since the amount of elements is represented in `width_ndx`, the size of each element can be calculated by using the knowledge found in the `width_scheme` along with the value of size, one can calculate the complete size of a payload, and thus calculate the total size of the array. So to summarise:

Table 8.7: Calculations Required for width_scheme Values

Width_scheme	The method used to calculate the overall size of the payload
Width_ndx	The number of elements present within the payload
Size	The size of each element within the payload

8.5.11 Realm Array Payload

After the 8-byte header, the remainder of the Realm Array is followed by multiples of 8-bytes, which makes up the “payload”. The amount of bytes that follow after the Realm Array Header (so, the total size of the payload) can be calculated by taking the `width_ndx`, `width_scheme` and size as explained in the section above.

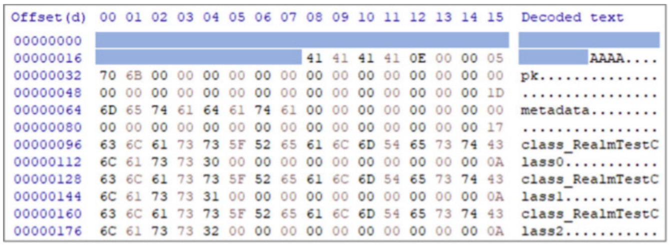
This is outlined within the `node_header.hpp` file found within the Realm-core documentation [67] where you will find the following code:

```
static size_t calc_byte_size(WidthType wtype, size_t size,
    uint_least8_t width) noexcept
    {
        size_t num_bytes = 0;
        switch (wtype) {
            case wtype_Bits: {
                // Current assumption is that size is at most
                    2^24 and that width is at most 64.
                // In that case the following will never
                    overflow. (Assuming that size_t is at least
                    32 bits)
                REALM_ASSERT_3(size, <, 0x1000000);
                size_t num_bits = size * width;
                num_bytes = (num_bits + 7) >> 3;
                break;
            }
            case wtype_Multiply: {
                num_bytes = size * width;
                break;
            }
            case wtype_Ignore:
                num_bytes = size;
                break;
        }

        // Ensure 8-byte alignment
        num_bytes = (num_bytes + 7) & ~size_t(7);

        num_bytes += header_size;

        return num_bytes;
    }
```

From this code-snippet we can deduce that if we take the value of size and multiply it with the width value, these are the amount of bits or bytes of the complete Realm Array payload. Whether this value is in bits or bytes depends on the `width_scheme` (referred to by wtype in the code example). This number is padded until it becomes a multiple of 8. The total size of a Realm Array is the payload size in addition to the Realm Array Header size (which is 8 bytes).

8.5.12 Size Calculation Example

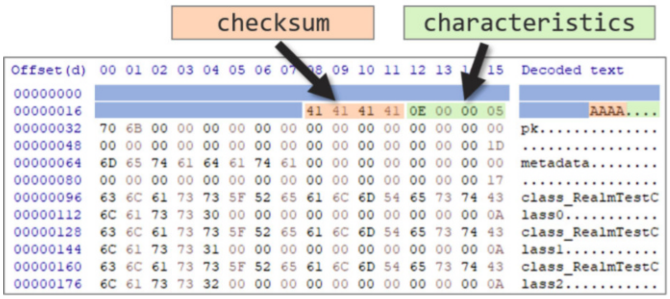
We shall revisit the example content used when introducing the Realm Array Header in 8.5.7 above, to demonstrate how some of these calculations work.



```
Offset(d) 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000
00000016                          41 41 41 41 0E 00 00 05      AAAA....
00000032  70 6B 00 00 00 00 00 00 00 00 00 00 00 00 00 00  pk..............
00000048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 1D  ................
00000064  6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 00  metadata........
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 17  ................
00000096  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000112  6C 61 73 73 30 00 00 00 00 00 00 00 00 00 00 0A  lass0...........
00000128  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000144  6C 61 73 73 31 00 00 00 00 00 00 00 00 00 00 0A  lass1...........
00000160  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000176  6C 61 73 73 32 00 00 00 00 00 00 00 00 00 00 0A  lass2...........
```

Fig. 8.32: Realm Array Example

This example looks at an Array that is stored immediate after the Realm file header, which we have blanked out to help focus on the Array itself (Fig. 8.32). Remember, our Array Header consists of eight bytes that can be broken down into two distinct elements (Fig. 8.33).



```
Offset(d) 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000
00000016                          41 41 41 41 0E 00 00 05      AAAA....
00000032  70 6B 00 00 00 00 00 00 00 00 00 00 00 00 00 00  pk..............
00000048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 1D  ................
00000064  6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 00  metadata........
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 17  ................
00000096  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000112  6C 61 73 73 30 00 00 00 00 00 00 00 00 00 00 0A  lass0...........
00000128  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000144  6C 61 73 73 31 00 00 00 00 00 00 00 00 00 00 0A  lass1...........
00000160  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000176  6C 61 73 73 32 00 00 00 00 00 00 00 00 00 00 0A  lass2...........
```

Fig. 8.33: Realm Array Header Example

We can see that the header for our array is made up of the following eight bytes:

Table 8.8: Array Header Bytes

Byte	1	2	3	4	5	6	7	8
Hex Value	41	41	41	41	0E	00	00	05

8.5.13 Array Example - Header

The first four bytes are our checksum, which, at the time of writing, is always 0x41414141 ("AAAA" in ASCII) and then our next set of four bytes are the characteristics values. If you recall from earlier in the chapter, this is further broken down into two different components, namely the flags and size values.

8.5.14 Array Example - Flags

The fifth byte is our flags, which is currently the value 0x0E, which is the binary value of b00001110. If you recall, the flags byte is broken into five "Bit Groups" that represent different things depending on their values. For our example value, we can breakdown the byte as follows:

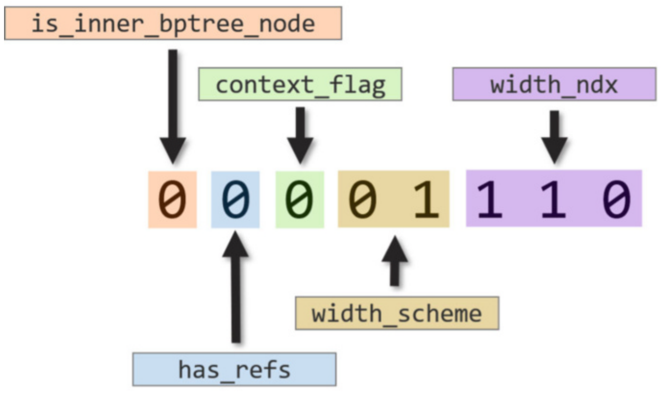


Fig. 8.34: Realm Array Example - Header Breakdown

From this we can identify the following:

- Our array is not an "Inner Node", given our `is_inner_bptree_node` value is set at 0.
- Our array is likely a "Data Array" as the `has_refs` value is set at 0.
- Our value of `width_scheme` is set at 1. Looking back at Table 8.5 in 8.5.9 above, this tells us that the scheme we need to use is calculating with the number of bytes (Width*size).
- The value of `width_ndx` is 110 in binary, which is the value 6 in decimal, which gives us the `width_ndx` value of 32 from the translation table provided by Table 8.6 in section 8.5.9 above.

To clarify how we made this conversion, we followed these simple steps:

1. Convert the Binary Value 110 into Decimal

This can be done using a calculator or converter, but to manually do this conversion we can simply look at the base 2 number values:

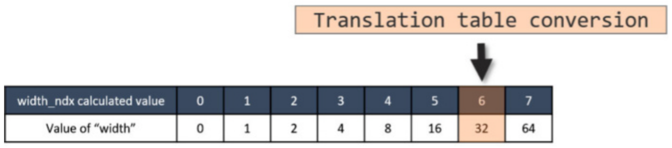
Table 8.9: Calculations Required for width_scheme Values

Base 2 columns	4	2	1
Binary Value	1	1	0

So our decimal calculation is:

1 x 0 = 0
2 x 1 = 2
4 x 1 = 4
0 + 2 + 4 = 6

2. Find the Decimal Value on the Translation Table


Translation table conversion

width_ndx calculated value	0	1	2	3	4	5	6	7
Value of "width"	0	1	2	4	8	16	32	64

Fig. 8.35: Annotated Copy of the width_ndx Translation Table

3. Identify the Given Value that the Table Returns

As we can see above, the returned value for "width" is 32.

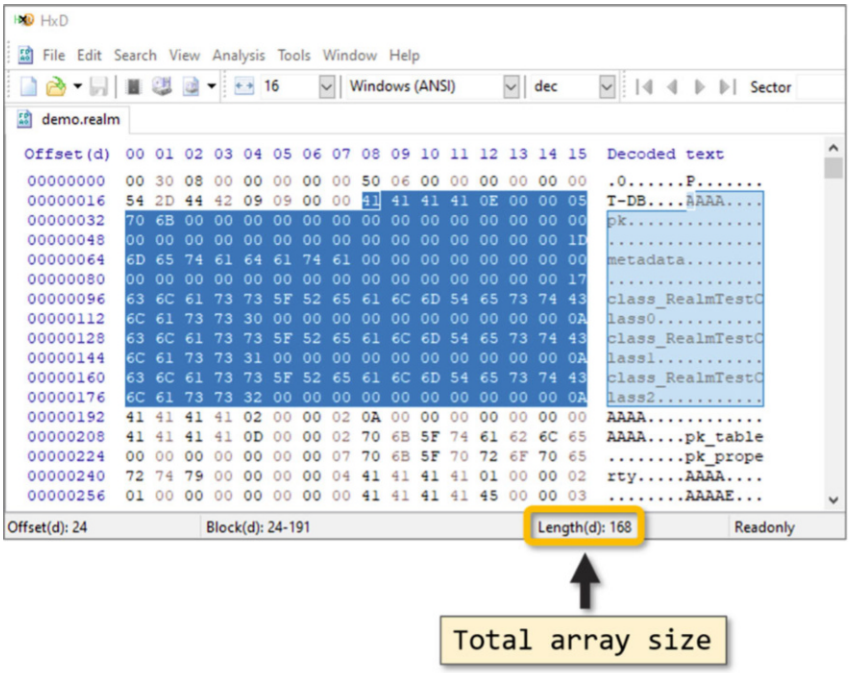
8.5.15 Array Example - Size

Our final series of bytes within the Array Header denotes the value for size. Given we have a three-byte value of 0x000005, which is the decimal value of 5, we can confirm the value of size to be 5.

Now that we have our values for both "width" and "size" we can use the given `width_scheme` calculation method to identify the total size of the payload for this array, as follows:

Size x Width = Payload size in bytes
5 x 32 = 160 bytes

Remember that the total size of our array is the header plus the payload size. Our header is always going to be 8 bytes long, and our payload is 160 bytes, so this array should be 168 bytes in total. You can see in the following screenshot of our Realm file viewed through the software tool HxD [36] that our array is, indeed, 168 bytes in length (Fig. 8.36):



```
Offset(d) 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15  Decoded text
00000000  00 30 08 00 00 00 00 00 50 06 00 00 00 00 00 00  .0......P.......
00000016  54 2D 44 42 09 09 00 00 41 41 41 41 0E 00 00 05  T-DB....AAAA....
00000032  70 6B 00 00 00 00 00 00 00 00 00 00 00 00 00 00  pk..............
00000048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 1D  ................
00000064  6D 65 74 61 64 61 74 61 00 00 00 00 00 00 00 00  metadata........
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 17  ................
00000096  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000112  6C 61 73 73 30 00 00 00 00 00 00 00 00 00 00 0A  lass0...........
00000128  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000144  6C 61 73 73 31 00 00 00 00 00 00 00 00 00 00 0A  lass1...........
00000160  63 6C 61 73 73 5F 52 65 61 6C 6D 54 65 73 74 43  class_RealmTestC
00000176  6C 61 73 73 32 00 00 00 00 00 00 00 00 00 00 0A  lass2...........
00000192  41 41 41 41 02 00 00 02 0A 00 00 00 00 00 00 00  AAAA............
00000208  41 41 41 41 0D 00 00 02 70 6B 5F 74 61 62 6C 65  AAAA....pk_table
00000224  00 00 00 00 00 00 00 07 70 6B 5F 70 72 6F 70 65  ........pk_prope
00000240  72 74 79 00 00 00 00 04 41 41 41 41 01 00 00 02  rty.....AAAA....
00000256  01 00 00 00 00 00 00 00 41 41 41 41 45 00 00 03  ........AAAAE...

Offset(d): 24    Block(d): 24-191    Length(d): 168    Readonly
```

Fig. 8.36: Screenshot of Realm File in HxD showing Total Size of an Array

If you look closely at the end of the array in Fig. 8.36, you will notice that the byte following the end of the highlighted array is the start of another array, with the array header ASCII values “AAAA” clearly visible within the decoded text.

8.6 Conclusion

Realm databases have been considered, by some, to be the new format that will ultimately replace SQLite. While we have seen some evidence of this with some applications, the anticipated change has been far slower than originally believed, and

Realm databases are still not commonly found within the most common applications examined by Digital Forensic examiners. This may change in the future as momentum builds for the database, or the movement may continue to struggle to gain traction as developers find workarounds and clever ways to continue implementing the more widely known and understood SQLite format.

The Realm database format has, at the time of writing, an active developer community with an ever-evolving code-base. The format is still relatively new, with a number of areas of code still using "dummy" or "temporary" values or data structures while the code is developed. The open-source nature of the realm-core code enables forensic examiners to reverse engineer the code in order to discover exactly how these database are structured and operate, which may be vital in digital forensic investigations.

However, the move away from SQL structures and into editable and customisable object-oriented code also adds challenges for forensic examiners, especially when every database could, in theory, be coded to operate and function is very different ways from any other. Understanding the fundamental concepts and structures behind Realm databases should enable examiners to navigate and understand some of the core data structures, even if the object functions remain challenging to decode and decipher.

This chapter has aimed to introduce some of the core fundamentals behind the Realm database, with a view of providing the foundations and tools that could be helpful in continued, further research and analysis. We hope that readers are able to use any knowledge gained from this chapter and referenced materials, to continue to explore and build their understanding of this database format, and we hope that the community continues to explore, share knowledge, and help one another enhance our understanding of new and developing file formats and structures for the benefit of all.