POLITECHNIKA BIAŁOSTOCKA

WYDZIAŁ INFORMATYKI

KRYSTIAN PATRYK ŻABICKI

.....
(PODPIS)

PORÓWNANIE WYDAJNOŚCI ORAZ SPOSOBÓW IMPLEMENTACJI BAZ DANYCH W SYSTEMIE IOS

Praca magisterska

napisana pod kierunkiem

dr inż. Marcina Skoczylasa

.....
(Podpis)

BIAŁYSTOK 2018

Karta dyplomowa

POLITECHNIKA BIAŁOSTOCKA Wydział..... ..... Katedra/Zakład.....	Studia..... stacjonarne/niestacjonarne ..... studia I stopnia/ studia II stopnia	Nr albumu studenta..... Rok akademicki..... Kierunek studiów..... ..... Specjalność..... .....

.....

Imię i nazwisko studenta

TEMAT PRACY DYPLOMOWEJ:

.....

.....

.....

Zakres pracy:

1. .....
2. .....
3. .....
4. .....

Słowa kluczowe (max 5): .....

.....	.....
Imię i nazwisko, stopień/ tytuł promotora - podpis	*Imię i nazwisko kierownika katedry - podpis*

.....	.....	.....
Data wydania tematu pracy dyplomowej - podpis promotora	*Regulaminowy termin złożenia pracy dyplomowej*	*Data złożenia pracy dyplomowej - potwierdzenie dziekanatu*

.....	.....
Ocena promotora	*Podpis promotora*

.....	.....	.....
Imię i nazwisko, stopień/ tytuł recenzenta	*Ocena recenzenta*	*Podpis recenzenta*

Thesis topic: **C**omparison of performance and methods of implementing databases in iOS

SUMMARY: The purpose of this study was to compare the performance of selected databases and how to implement them in iOS. Interface to each database has been implemented separately in a standard way for a given database. The study compared different types of relational databases and NoSQL (column-oriented). Important aspect of the study was to ensure adequate test environments, which may impact the correctness of the results obtained regardless of differences in architecture. The work also presents a comparison of the methods of implementing the tested databases.

Załącznik nr 4 do „Zasad postępowania przy przygotowaniu i obronie pracy dyplomowej na PB”

..... Białystok, dnia……

Imiona i nazwisko studenta

.....

Nr albumu

.....

Kierunek i forma studiów

.....

Promotor pracy dyplomowej

OŚWIADCZENIE

Przedkładając w roku akademickim 2017/2018 Promotorowi …….. pracę dyplomową pt.: .....……..
……..……..……..……..……..…
….……....……, dalej zwaną pracą dyplomową,

oświadczam, że:

1) praca dyplomowa stanowi wynik samodzielnej pracy twórczej;
2) wykorzystując w pracy dyplomowej materiały źródłowe, w tym w szczególności: monografie, artykuły naukowe, zestawienia zawierające wyniki badań (opublikowane, jak i nieopublikowane), materiały ze stron internetowych, w przypisach wskazywałem/am ich autora, tytuł, miejsce i rok publikacji oraz stronę, z której pochodzą powoływane fragmenty, ponadto w pracy dyplomowej zamieściłem/am bibliografię;
3) praca dyplomowa nie zawiera żadnych danych, informacji i materiałów, których publikacja nie jest prawnie dozwolona;
4) praca dyplomowa dotychczas nie stanowiła podstawy nadania tytułu zawodowego, stopnia naukowego, tytułu naukowego oraz uzyskania innych kwalifikacji;
5) treść pracy dyplomowej przekazanej do dziekanatu Wydziału Informatyki jest jednakowa w wersji drukowanej oraz w formie elektronicznej;
6) jestem świadomy/a, że naruszenie praw autorskich podlega odpowiedzialności na podstawie przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. z 2017 r. poz. 880), jednocześnie na podstawie przepisów ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (Dz. U. z 2016 r. poz. 1842, z późn. zm.) stanowi przesłankę wszczęcia postępowania dyscyplinarnego oraz stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego;
7) udzielam Politechnice Białostockiej nieodpłatnej licencji niewyłącznej na umieszczenie elektronicznej wersji pracy dyplomowej w repozytorium uczelnianym/Bazie Wiedzy PB i do korzystania z utworu bez ograniczeń czasowych i terytorialnych przez jego udostępnienie on-line dla użytkowników sieci wewnętrznej Politechniki Białostockiej. Upoważniam Politechnikę Białostocką do przechowywania i archiwizowania pracy dyplomowej na nośnikach cyfrowych oraz jej zwielokrotniania i udostępniania w formie elektronicznej w zakresie koniecznym do weryfikacji autorstwa pracy i ochrony przed przywłaszczeniem autorstwa, w tym na przekazanie pracy dyplomowej do Ogólnopolskiego Repozytorium Prac Dyplomowych;
8) zostałem/am poinformowany/a, że na podstawie art. 24 ust. 1 ustawy z dnia 29 sierpnia 1997 roku o ochronie danych osobowych (Dz. U. z. 2016 r. poz. 922, z późn. zm.) administratorem danych jest rektor Politechniki Białostockiej, ul. Wiejska 45A, 15-351 Białystok. Dane będą przetwarzane w celach realizacji procedury antyplagiatowej przyjętej na Politechnice Białostockiej i nie będą udostępniane odbiorcom danych w rozumieniu art. 7 pkt. 6 ustawy o ochronie danych osobowych. Osobie, której dane dotyczą, przysługuje prawo dostępu do treści swoich danych oraz ich poprawiania. Podanie danych jest obowiązkowe (art. 167b ustawy z dnia 27 lipca 2005 roku Prawo o szkolnictwie wyższym Dz. U. z 2016 r. poz. 1842 z późn. zm.). Dane będą przetwarzane w Politechnice Białostockiej przez okres 50 lat.

…….
czytelny podpis studenta

Spis treści

1 Wstęp

W dzisiejszych czasach, kiedy dostęp do urządzeń mobilnych stał się powszechny, a większość tych urządzeń dorównuje swoimi parametrami tabletom czy też nawet laptopom aplikacje mobilne stały się bardzo popularne. Wraz ze wzrostem popularności aplikacji pisanych na smatphony wymagania stawiane przed tymi programami znacząco wzrosły. Od najprostszych aplikacji wymaga się najczęściej spotykanych funkcjonalności stron internetowych lub aplikacji komputerowych takich jak połączenie z internetem, uwierzytelnianie użytkownika, synchronizacji i zapisu danych.

Obecnie praktycznie każda z aplikacji zawiera w sobie bazę danych. Synchronizowaną z serwerem, przechowującą znaczące ilości danych odpowiadających za poprawne działanie programu, a także często zapewniającą prace aplikacji bez połączenia z internetem. Ważne jest więc aby zastosowana baza danych zapewniała satysfakcjonującą prędkości dostępu do danych i nie narażała użytkownika na opóźnienia związane z pozyskaniem danych. Istotną kwestią z punktu widzenia programisty - osoby tworzącej aplikację jest łatwość implementacji wydajnej bazy danych. Łatwość implementacji przekłada się w późniejszych etapach życia oprogramowania na szybsze i prostsze dodawanie nowych elementów, relacji czy tablic w zastosowanej bazie, dzięki czemu oszczędzany jest cenny czas przeznaczony na proces wytwarzania programu.

Niniejsza praca ma na celu przedstawienie obecnie dostępnych baz danych dla systemu iOS, pokazanie i porównanie sposobów ich implementacji oraz porównanie ich szybkości. Na potrzeby pracy stworzono aplikacje będącą środowiskiem testowym wybranych baz danych.

1.1 Cel pracy

Celem niniejszej pracy było porównanie wydajności wybranych baz danych oraz sposobów ich implementacji w systemie iOS. Każda z baz została zaimplementowana oddzielnie w standardowy dla danej bazy sposób. W pracy porównane zostały różne typy baz danych typowo relacyjnych jak i NoSQL (column-oriented), ważnym aspektem było więc zapewnienie odpowiednich środowisk testowych, które mogły zapewnić poprawność otrzymanych wyników niezależnie od różnic w architekturze. Praca przedstawia również porównanie sposobów implementacji testowanych baz danych.

Zakres pracy obejmuje następujące zagadnienia:

- Wykonanie zbiorczego przeglądu literatury w tematyce
- Opis dostępnych baz danych w systemie iOS (Core data, SQLite, Realm, UserDefaults, FMDB)
- Stworzenie narzędzia do porównania wydajności baz danych oraz metod testowych.
- Przygotowanie struktury testów porównawczych i środowisk testowych
- Zebranie wyników testów dla wybranych operacji oraz analiza wyników.
- Opis wniosków na podstawie zebranych danych badawczych.

1.2 Struktura pracy

Praca składa się z siedmiu rozdziałów:

- Opis wybranych baz danych - w rozdziale przedstawione zostały opisy wybranych baz danych dla systemu iOS: Core Data, Realm, FMDB oraz User Defaults.
- Dostępne porównania baz danych - w rozdziale zostały przedstawione dostępne porównania szybkości niektórych baz danych.
- Narzędzie do porównywania wydajności i model bazy - w rozdziale została przedstawiona aplikacja umożliwiająca przeprowadzenie testów wybranych baz danych. Pokazana zostanie struktura testowej bazy danych oraz opisane zostały przeprowadzane testy.
- Opis przeprowadzonych testów - w rozdziale zostały opisane testy przeprowadzone w ramach pracy. Przedstawiono kroki potrzebne do otrzymania poprawnych wyników w zależności od typu testu oraz pokazano w jaki sposób został mierzony czas uzyskania rezultatu każdej operacji.
- Różnice implementacji baz danych - w rozdziale zostały pokazane różnice i podobieństwa w implementacji poszczególnych rozwiązań bazodanowych. Przedstawiono fragmenty kodów wykonujące te same operacje przy użyciu różnych baz danych. Opisany też został stopień skomplikowania każdej z operacji.

- Analiza - w rozdziale zostały przedstawione wyniki przeprowadzonych testów. Każdy z rezultatów testów został zanalizowany i opisany.
- Podsumowanie - zawarte są w nim wnioski i analiza rezultatów pracy.

2 Opis wybranych baz danych

W tym rozdziale przedstawione zostały opisy wybranych baz danych dla systemu iOS: Core Data, Realm, FMDB oraz User Defaults. W systemie iOS istnieje wiele rozwiązań umożliwiających wdrożenie bazy danych w aplikacji, przedstawione tutaj przykłady są jednymi z najczęściej stosowanych przez programistów bazami danych w aplikacjach iOS.

2.1 SQLite

SQLite jest wewnątrz procesową biblioteką dostępną w systemie iOS. Kod opisywanej bazy jest dostępny publicznie, można go używać w dowolnym celu komercyjnym lub prywatnym bez żadnych opłat. Sprawia to, że baza ta jest jednym z najczęściej wybieranych przez programistów rozwiązań do przechowywania danych w aplikacjach. Wybór tej biblioteki jest też kierowany tym, iż projekt istnieje już od 2000 roku a jego twórcy deklarują wsparcie aż do 2050 roku[**SQLite-doc**] .

SQLite jest „lżejszą"wersją SQL więc korzystanie z funkcji biblioteki zostało znacząco uproszczone. Baza ta nie posiada wydzielonych procesów serwerowych, dane zapisuje i odczytuje ze zwykłych plików dyskowych. Kompletna baza danych programu zawierająca wszystkie tabele, relacje i tym podobne komponenty zapewniające poprawne działanie bazy przechowywane są w jednym pliku na dysku. Dzięki przechowywaniu wszystkich tych informacji w jednym pliku, informacje te mogą być kopiowane w stanie nienaruszonym pomiędzy programami.

System iOS zapewnia wbudowaną obsługę baz SQLite, nie istnieje więc potrzeba dodawania do projektu dodatkowych bibliotek. Cała komunikacja z biblioteką jest natywna i wymaga jedynie zaimportowania SQLite3. SQLite jest bazą relacyjna i zapewnia dodawanie relacji pomiędzy tabelami takich jak:

- N:M – wiele do wielu
- 1:N – jeden do wielu
- 1:1 – jeden do jednego

Zapewniona została też obsługa danych takich jak:

- NULL - wartość pusta

- INTEGER - wartość całkowita ze znakiem, przechowywana w 1, 2, 3, 4, 6 lub 8 bajtach w zależności od wielkości wartości
- REAL - wartość zmiennoprzecinkowa, przechowywana jako 8-bajtowa liczba zmiennoprzecinkowa IEEE
- TEXT - wartością jest ciąg znaków przechowywany przy użyciu kodowania bazy danych (UTF-8, UTF-16)
- BLOB - wartość przeznaczona do przechowywania wielkich plików w formie bajtowej takich jak obrazy, muzyka itp.

SQLite nie zapewnia wsparcia dla przechowywania obiektów zawierających datę. W celu zapisania daty w tablicy należy używać wartości tekstowych lub liczbowych a następnie zapewnić ich odpowiednie odczytanie i konwersje do odpowiednich obiektów języka, w którym przebiega implementacja programu.

2.2 Core Data

Core Data jest dedykowanym rozwiązaniem Apple umożliwiającym zaimplementowanie bazy danych w aplikacjach iOS i MacOS. Została wprowadzona od wersji iOS 3.0 w 2009 roku. Znacznie szybciej istniała możliwość używania tej bazy danych w MacOS, pierwszą wersją systemu, która to umożliwiła był MacOS Tiger, który ukazał się w 2005 roku.

Mimo że Core Data służy do przechowywania danych sama w sobie nie jest bazą danych a framework'iem który zarządza diagramem obiektów aplikacji. Diagram obiektów jest to zbiór obiektów powiązanych ze sobą relacjami. Core Data zajmuje się zarządzaniem cyklem życia obiektów w diagramie obiektów na dysku, a także oferuje interfejs do przeszukiwania obiektów, którymi zarządza. Core Data daje także możliwość sprawdzania poprawności danych wejściowych, modelowania modeli danych czy też śledzenia zmian.

Core Data jako rozwiązanie implementacji bazy danych korzysta z SQLite, przechowuje za jego pomocą wszystkie obiekty na dysku. Rozszerza jednak oferowane funkcjonalności SQLite o obsługę bazy w kilku wątkach, tworzenia tymczasowych obiektów (kiedy nie jesteśmy pewni, czy one zostaną zapisane na dysk), automatycznie nasłuchiwanie na zmiany w bazie danych, między innymi jeśli wyświetlamy dane użytkownika na ekranie, a dane

w bazie danych się zmienią (za sprawą na przykład drugiego wątku pracującego z zewnętrznym serwerem API) to dostaniemy odpowiednie powiadomienie. Core Data zapewnia łatwą integracje z kontrolkami interfejsu użytkownika takimi jak UITableView (kontrolka odpowiedzialna za wyświetlanie tabel/list) między innymi zapewniając automatyczne tworzenie indeksów i sekcji.

Core Data jest zorganizowana w dużą hierarchie klas, lecz używane są najczęściej podstawowe obiekty takie jak:

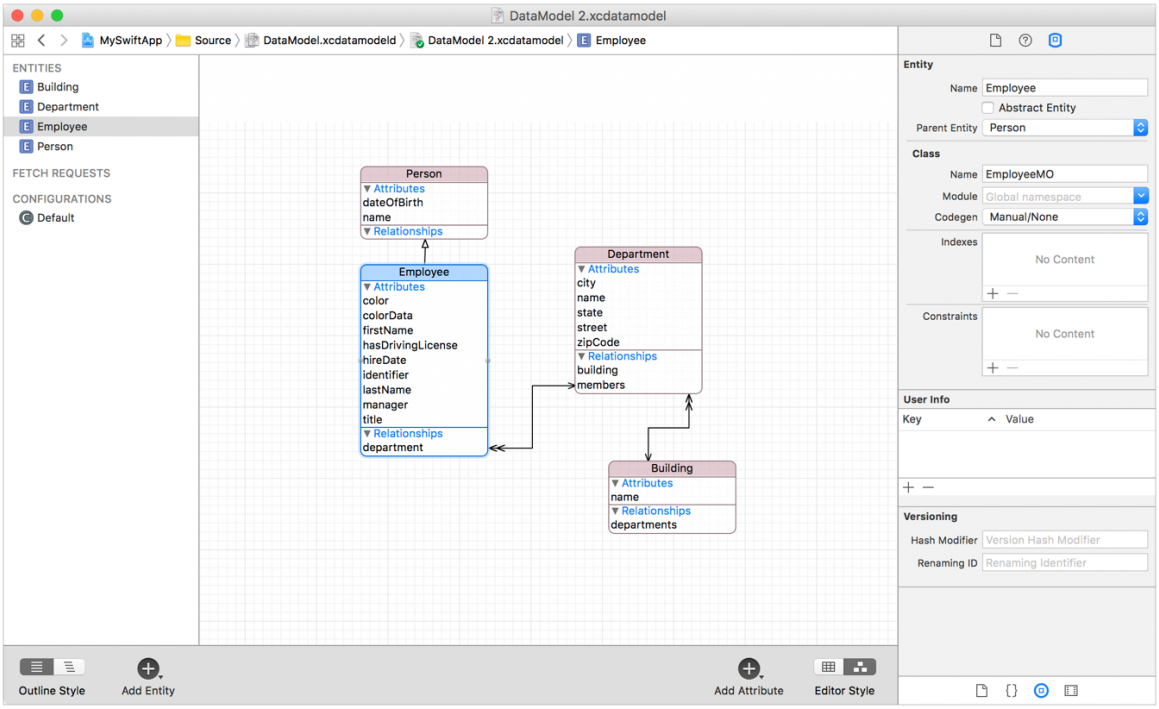
- NSManagedObject - Klasa reprezentująca jeden wiersz tabeli, zapewniajacy dostęp do danych przechowywanych w wierszu.
- NSManagedObjectContext - Główny kontekst bazy, wykonuje operacje na bazie. Pozwala na zapis aktualnego stanu bazy danych.
- NSManagedObjectModel - Klasa reprezentujący model tablicy.
- NSFetchRequest - Klasa pozwalająca utworzyć zapytanie mające na celu przeprowadzenie operacji na danych.
- NSPersistentStoreCoordinator - Klasa odpowiedzialna za operacje na pliku który przechowuje dane bazy. Za pomocą tej klasy możliwe jest całkowite jest usunięcie.
- NSPredicate - Klasa za pomocą której tworzymu zapytanie do bazy danych, używana jest jako parametr metody klasy NSFetchRequest.

Tak samo jak poprzednio opisywana baza danych Core Data pozwala na tworzenie wszystkich możliwych relacji pomiędzy tabelami. Framework zapewnia też możliwość przechowywania w tabeli wielu typów danych takich jak:

- String - łańcuch znaków
- Int - liczba całkowita
- Float - liczba zmiennoprzecinkowa
- Double - liczba zmiennoprzecinkowa
- Boolean - wartość boolowaska prawda/fałsz

- Date - obiekt przechowujący date
- Binary Data - dane w formacie binarnym
- URI - adresy zasobów
- Transformable - wartość do przechowywana niestandardowych danych

Jedną z ważnych cech tej bazy danych jest możliwość graficznego definiowania schematu bazy, pełne wsparcie zapewnia środowisko programistyczne XCode. Na rysunku poniżej został przedstawiony edytor bazy danych.



Rysunek 2.1: Edytor Core Data w XCode

Dodatkowo XCode pozwala generować klasy dla obiektów stworzonych za pomocą edytora, które wykorzystywane są później do tworzenia obiektów przed zapisem do bazy. Istnieją też inne narzędzia, które umożliwiają wygenerowanie klas dla stworzonego diagramu. Jednym z najpopularniejszym jest Mogenerator. Zapewnia on wsparcie dla języka Swift i Objective -C.

2.3 FMDB

FMDB jest biblioteką opakowującą SQLite. Można więc stwierdzić że FMDB i SQLite służą temu samemu celowi - umożliwiają efektywne zarządzanie danymi aplikacji. Lecz sposoby w jaki się je używa znacząco się od siebie różnią. FMBD oferuje interfejs wyższego poziomu ukrywając wszystkie szczegóły SQL takie jak połączenie i komunikacja z bazą, ale dalej oferuje dokładną obsługę danych.

FMDB zapewnia funkcje SQLite, więc podczas implementacji nie trzeba zajmować się połączeniami, a także pisaniem i odczytywaniem danych do i z bazy. Biblioteka ta jest dobrym rozwiązaniem dla programistów, którzy chcą wykorzystać swoją wiedzę SQL i pisać własne zapytania, ale bez potrzeby pisania własnego menadżera SQLite. FMDB działa w dwóch językach: Swift i Objective-C oraz bardzo szybko integruję się z projektem iOS. Jej twórcy zapewniają, że jest ona zaimplementowana tak aby zapewnić najwyższą wydajność SQLite, lepszą od prostych implementacji menadżerów dla SQLite.

Biblioteka jest bardo prosta w użyciu. Możemy w niej wyróżnić trzy główne klasy:

- FMDatabase - Reprezentuje pojedynczy obiekt bazy danych. Używana jest do wykonywania instrukcji SQL.
- FMResultSet - Klasa przetrzymująca rezultaty zapytań SQL wykonanych na bazie danych.
- FMDatabaseQueue - Klasa która jest używana podczas wykonywania kwerend i aktualizacji danych w bazie przy użyciu wielu wątków.

Jako, że FMBD oferuje interfejs wyższego poziomu do SQLite dodana została obsługa typów danych takich jak:

- Bool - wartość boolowaska prawda/fałsz
- Date - obiekt przechowujący date

Obsługa tych dwóch typów danych wymagała w SQLite dodatkowych operacji. W przypadku wartości prawda/fałsz należało zapisywać w tabeli zero lub jeden i odpowiednio konwertować wartości liczbowe do wartości Bool. Sytuacja podczas zapisu daty w SQLite wygląda podobnie, należy najpierw wartość daty skonwertować do wartości tekstowej a podczas odczytu danych wartość tekstową skonwertować do obiektu przechowującego datę.

FMBD w znaczącym stopniu ułatwia obsługę tak często używanych w bazach danych wartości. Dodatkowo warto wspomnieć, iż FMDB będąc biblioteką opakowującą SQLite daje możliwość korzystania z dokumentacji SQLite, która jest bardzo dokładna i ciągle rozwijana. Trudno więc zaprzeczyć, że biblioteka ta stanowi dobre rozwiązanie zastępujące standardowy SQLite.

2.4 Realm

Realm różni się od poprzednio opisywanych baz SQL. Biblioteka ta jest obiektową bazą danych NoSQL. Przechowuje dane w postaci obiektów. Realm jest rozwiązaniem open-source więc jest w pełni darmową baza danych, dedykowaną do aplikacji mobilnych. Dodatkowo jest dostępna na systemy Android i iOS, co znacząco ułatwia prace nad aplikacjami dostępnymi nie tylko na jedną platformę. Współpracuje z projektami pisanymi w: Swift, Objective-C, Java, React Native, Xamarin i innymi. Biblioteka ujrzała światło dzienne w 2016 roku a pierwsza stabilna wersja wydana została w styczniu 2017 roku. Jedną z wyróżniających ją funkcji jest możliwość dwukierunkowej synchronizacji pomiędzy bazą serwera a bazą aplikacji.

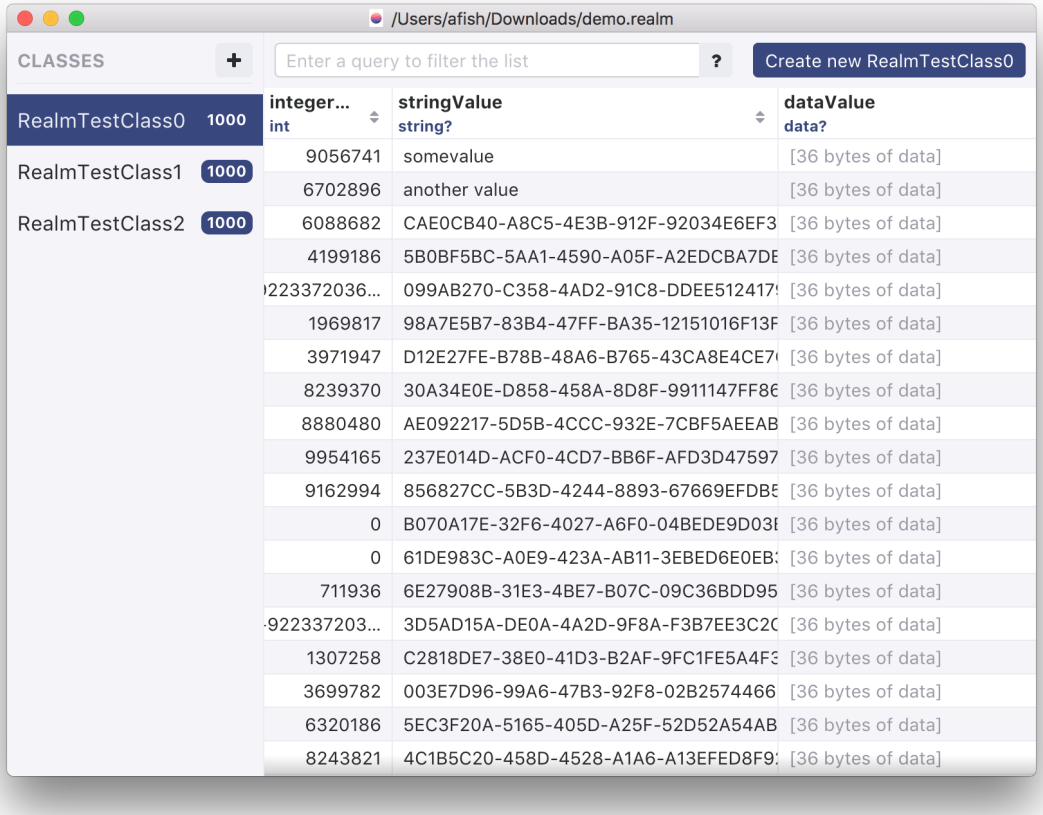
Realm swoją popularność zyskał poprzez szybkość, łatwość integracji z projektem oraz łatwością posługiwania się nim. Od bibliotek opakowujących SQLite oraz całego framework-u Core Data odróżnia go to, że większość typowych funkcji, takich jak odpytywanie bazy danych składa się z pojedynczych linii kodu. Dzięki temu używając biblioteki Realm uzyskuje się bardziej zwięzły kod i poprawia się jego czytelność. Podczas używania Realm nie potrzebna jest dobra znajomość SQL do zarządzania bazą i tworzenia zapytań. Nie wymagana jest też dobra znajomość Core Data, która jest zaawansowanym narzędziem i nie wiele osób posiada umiejętności pozwalające wydaje zarządzać tą bazą danych. W Realm dane są przechowywane jako obiekty, dzięki czemu przy odczytywaniu czy zapisywaniu danych w bazie nie ma potrzeby stosowania żadnej biblioteki ORM (Object Relation Mapping), pozwalającej na konwersje danych pomiędzy obiektem programu a rekordem tabeli. Dzięki temu nie występują problemy z wydajnością dodatkowych bibliotek ORM. Interfejs biblioteki ogranicza się do trzech klas:

- Realm - Główna klasa, pozwalająca na dostęp do bazy i wykonywanie operacji zapisu/odczytu.

- Object - Klasa reprezentująca model danych bazy.
- Result - Obiekt reprezentujący listę dane odczytane z bazy przy użyciu zapytania.

Realm tak jak Core Data czy FMDB zapewnia wsparcie dla wszystkich typów danych. Na uwagę zasługują tutaj relacje pomiędzy tabelami. Biblioteka jest obiektową bazą danych, nie ma więc w niej tabel i zwykłych relacji pomiędzy nimi. W Realm relacje pomiędzy obiektami tworzone są za pomocą linkowana obiektów. Każdy link tworzy link zwrotny, jako relację odwrotną do dowolnego obiektu łączącego się z bieżącym obiektem obiektem. Za pomocą Reaml Studio możliwe jest także stworzenie pliku bazy danych na podstawie dokumentu CSV.

Dodatkowo Realm dostarcza również narzędzie Realm Studio, widoczne na rysunku 2.2. Pozwala ono na łatwe zarządzanie bazą danych. Umożliwia otwieranie, przeglądanie i edytowanie lokalnych plików Realm.



Rysunek 2.2: Narzędzie Realm Studio

2.5 User Defaults

Kolejnym z rozwiązań pozwalającym zapisywać i przechowywać dane w systemi iOS jest User Defaults - domyślna baza użytkownika. Dane w niej przechowywane są pary klucz - wartość. System pomimo, iż nie jest prawdziwą bazą danych a jedynie interfejsem, który pozwala na przechowywanie potrzebnych, niewielkich danych aplikacji, został wybrany w tej pracy do porównania ponieważ wielu programistów używa go zamiast innych rozwiązań bazodanowych. Rozwiązanie to jest szybkie w implementacji i pozwala zapisywać niestandardowe obiekty, jeżeli implementują protokół NSCoding i zostaną skonwertowane do prostej formy Data. Więcej informacji na temat sposobu zapisu i konwersji danych znajduję się w rozdziale 6 .

User Defaults pozwala zapisywać wartości takie jak: Bool, Integer, Float, Double, String, Data, URL, Array, Dictionary. Zapis danych wykonuje się za pomocą funkcji set(:forKey:) a odczyt value(forKey:).

3 Przegląd literatury - dostępne porównania baz danych

W rozdziale zostały przedstawione dostępne porównania szybkości niektórych baz danych. Przytoczone porównania są w pewnym stopniu punktem odniesienia do uzyskanych w pracy wyników. W każdym z przypadków uzyskane wyniki są inne, testy były przeprowadzane na różnych danych. Różna była też ilość danych i ich złożoność a także sposób implementacji bibliotek.

3.1 Core Data i SQLite - porównanie wydajności

Porównanie zaczerpnięte z witryny internetowej http://www.drdobbs.com. Autor stworzył przykładową aplikacje pozwalająca przeprowadzić operacje na głównych bazach danych iOS. Testy zostały przeprowadzone na iPhone 5s.

Do testów został użyty prosty zestaw danych, który jest widoczny na rysunku 3.1 znajdującym się poniżej.

Car	Type	Details	Manufacturer
Pinto, $16,000	Compact	AC, Automatic transmission	Ford, Dearborn MI, 100,000 employees
Tahoe, $45000	SUV	Leather, Power Windows	GM, Dearborn, MI, 100,00 employees
Ferrari, $15000	Sports Car	V12	Maranello, Italy

Rysunek 3.1: Przykład danych użytych do porównania wydajności Core Data i SQLite

Autor przeprowadził testy pokazujące zajętość pamięci obiektu bazy na dysku urządzenia, testy użycia pamięci aplikacji używającej bazy oraz testy szybkości bazy podczas pobierania danych z bazy. Poniższe rysunki 3.2, 3.3, 3.4 prezentują ich rezultaty.

Record Type	50,000 Recs	100,000 Recs	200,000 Recs
Core Data	6532 KB (5s) 6412 KB (4s)	13296 KB (5s) 13180 KB (4s)	27004 KB (5s) 2,912 KB (4s)
SQLite	1676 KB (5s) 1676 KB (4s)	3364 KB (5s) 3364 KB (4s)	6824 KB (5s) 6824 KB (4s)

Rysunek 3.2: Rezultat testów porównania Core Data i SQLite - wielkość pliku bazy

Record Type	50,000 Recs	100,000 Recs	200,000 Recs
Core Data	32 MB (5s) 24 MB (4s)	53 MB (5s) 38 MB (4s)	91 MB (5s) 67 MB (4s)
SQLite	21 MB (5s) 16 MB (4s)	29 MB (5s) 22 MB (4s)	46 MB (5s) 36 MB (4s)

Rysunek 3.3: Rezultat testów porównania Core Data i SQLite - pamięć aplikacji

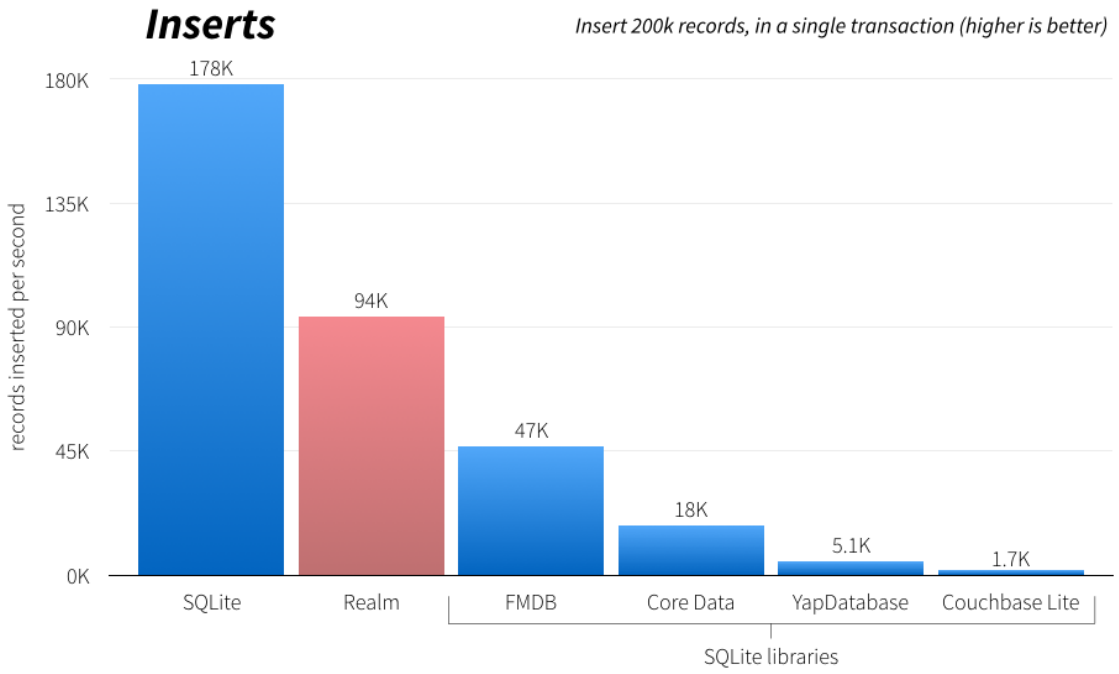
Record Type	Number of Records Fetched		
	50,000	100,000	200,000
Core Data	107 msec (5s) 397 msec (4s)	230 msec (5s) 850 msec (4s)	475 msec (5s) 1644 msec (4s)
SQLite	140 msec (5s) 730 msec (4s)	280 msec (5s) 1447 msec (4s)	580 msec (5s) 3077 msec (5s)

Rysunek 3.4: Rezultat testów porównania Core Data i SQLite - prędkość odczytu danych

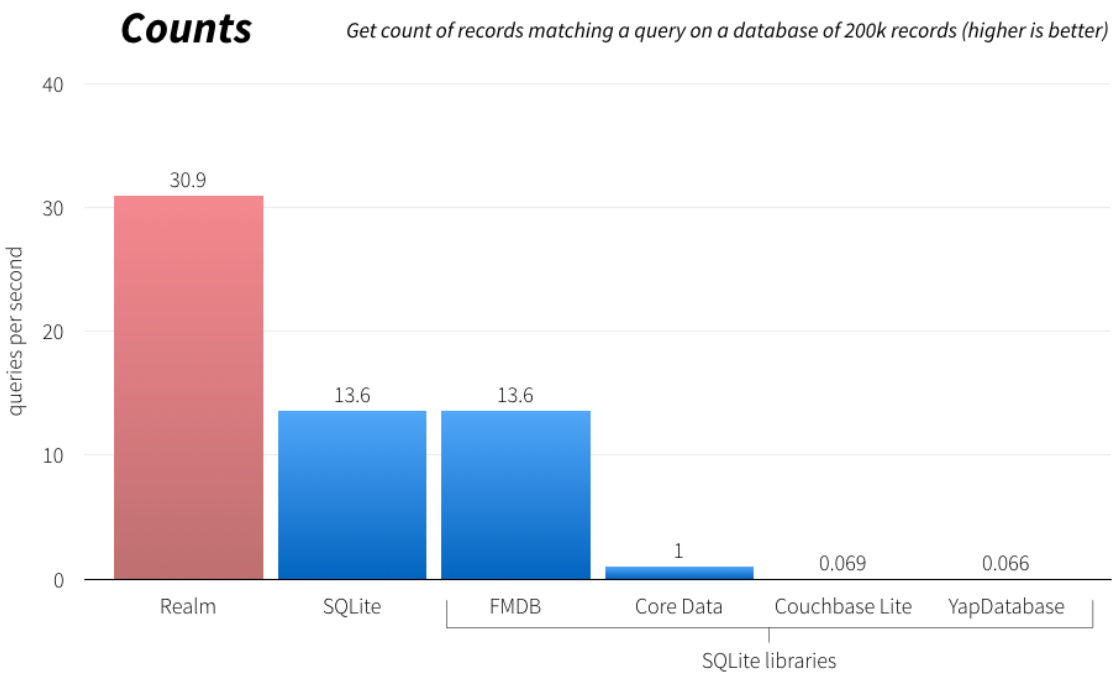
Wyniki wielkości pliku bazy pokazują że plik bazy Core Data zajmuje około cztery razy więcej miejsca niż plik bazy SQLite. Na rysunku 3.3 widać, że Core Data zużywa też więcej pamięci operacyjnej podczas działania aplikacji. Autor tłumaczy że wynika to ze sposobu w jaki działa Core Data. Tworzy ona w pierwszej kolejności obiekty w pamięci a dopiero później następuje zapis do bazy, przez co Core Data zużywa od 40% do 100% pamięci więcej niż SQLite. Rysunek 3.4 przedstawia czasy w jakich następuje kolejno odczyt 50 000, 100 000, 200 000 tysięcy rekordów z baz. W tym porównaniu znacząco dominuje Core Data. Na iPhone 4s osiąga dwu krotną szybkość odczytu danych.

3.2 Realm

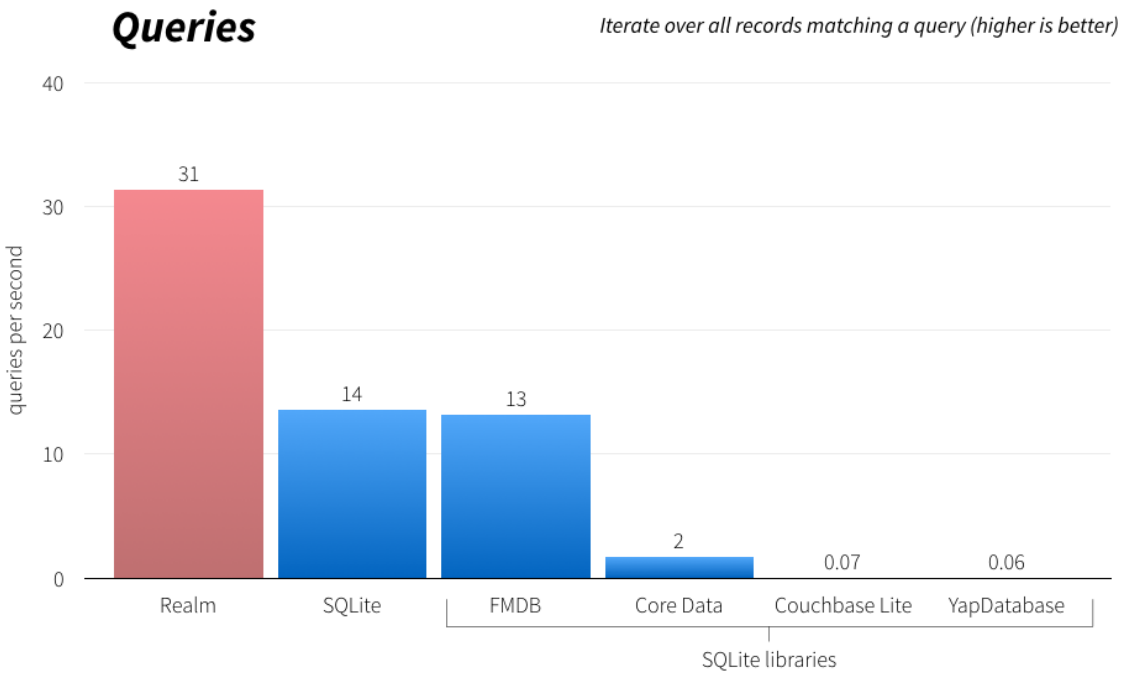
Wydawca Realm udostępnia na swojej stronie porównanie swojego rozwiązania z bazami takimi jak: SQLite, FMDB, Core Data, Couchbase Lite, YapDatabase. W testach zostały porównane prędkości zapisu danych, wykonania czasu zapytania oraz zliczenia rekordów. Poniższe rysunki pokazują ich wyniki.



Rysunek 3.5: Rezultat testów Realm - zapis danych do bazy



Rysunek 3.6: Rezultat testów Realm - zliczanie rekordów



Rysunek 3.7: Rezultat testów Realm - odczyt danych

Realm w opisach zapewnia, że ich produkt posiada doskonałą wydajność. Szybszą nawet w niektórych przypadkach od SQLite. Testy wydają się potwierdzać ich słowa. SQLite wypadł gorzej od Realm tylko w jednym teście, które przedstawili. Okazał się on najszybszy przy zapisie danych do bazy, osiągając blisko dwukrotnie wyższy wynik od Realm wynoszący 178 000 zapisanych rekordów w ciągu sekundy, rezultat widoczny jest na rysunku 3.5. Kolejne testy pokazują dominacje Realm. W teście zliczania rekordów SQLite i FMDB uzyskują taki sam wynik - 13.6 operacji na sekundę zaś Realm wykonuje aż 30.9 operacji co jest wynikiem ponad dwukrotnie wyższym. W przypadku odczytu danych Realm też jest najszybszy. Osiąga wynik 31 operacji na sekundę a SQLite i FMDB jedynie 13-14 operacji. Wynik kolejny raz jest ponad dwukrotnie lepszy.

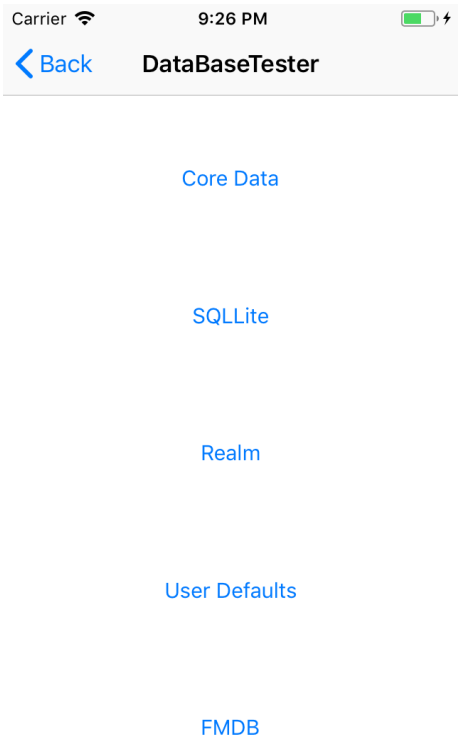
Wynik testów są bardzo imponujące i dowodzą, że Realm jest jedną z najszybszych baz danych. Lecz autorzy nie pokazują sposobu testowania baz ani nie ujawniają na danych jakiego typu odbyły się testy.

4 Narzędzie do porównywania wydajności i model bazy

W rozdziale zostanie przedstawiona aplikacja umożliwiająca przeprowadzenie testów wybranych baz danych. Pokazana zostanie struktura testowej bazy danych oraz opisane zostaną przeprowadzane testy.

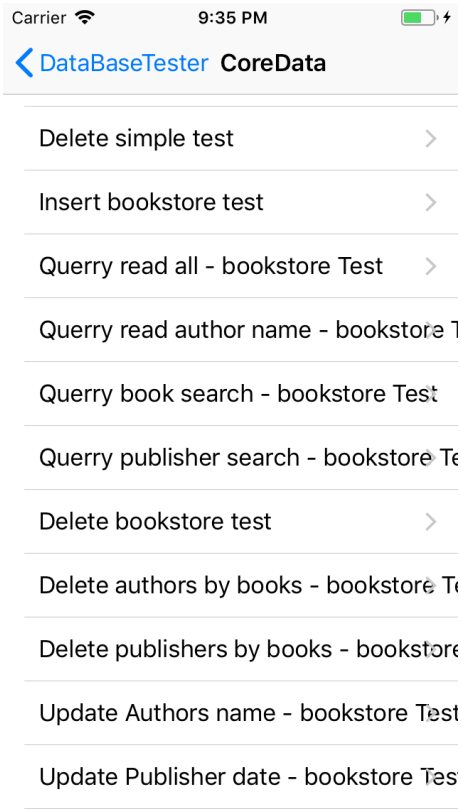
4.1 Aplikacja

W celu przeprowadzenia testów wybranych baz danych stworzona została aplikacja umożliwiająca przetestowanie szybkości baz danych w systemie iOS. Na rysunku 4.1 znajdującym się poniżej pokazany został pierwszy ekran aplikacji umożliwiający wybranie baz danych do testów.



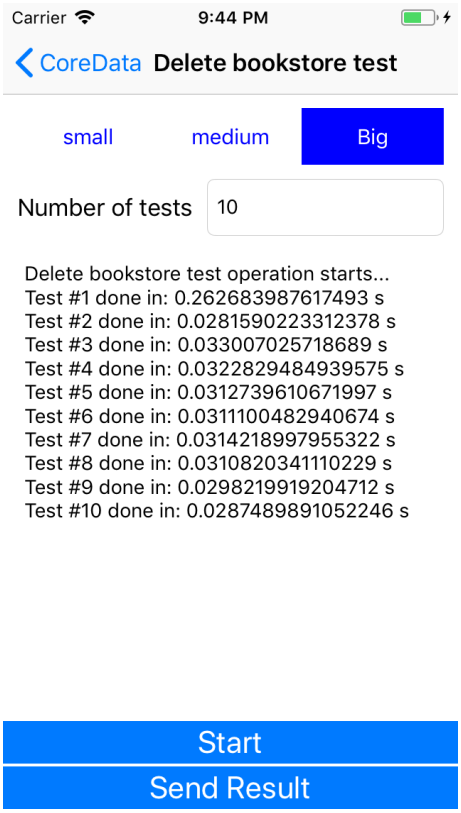
Rysunek 4.1: Ekran wyboru bazy danych do testów

Po wyborze bazy danych użytkownik zostaje przeniesiony do kolejnego ekranu aplikacji w którym należy wybrać jeden z dostępnych testów. Ekran widoczny jest na rysunku 4.2 znajdującym się poniżej.



Rysunek 4.2: Ekran wyboru testu bazy danych

Po wyborze jednego z testów następuje przejście do kolejnego ekranu widocznego na rysunku 4.3. Jest to ekran wyboru parametrów testu takich jak wielkość zestawu danych testowych i ilości powtórzeń testu.



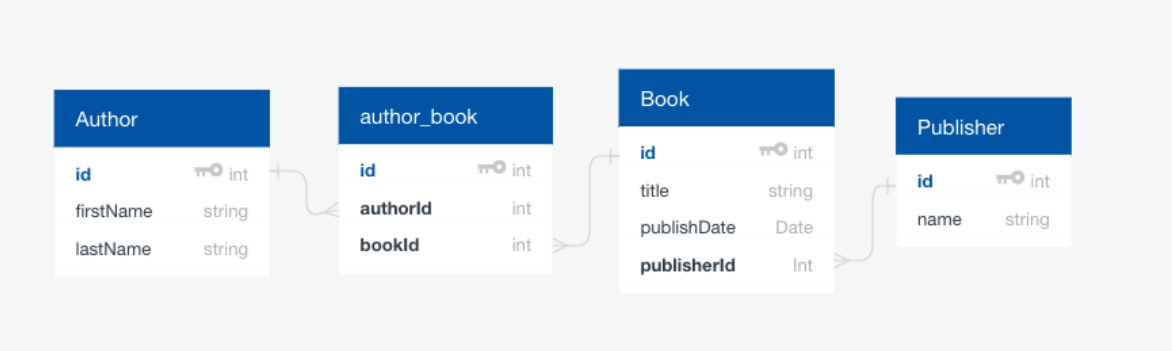
Rysunek 4.3: Ekran wykonywania testu

Na ekranie znajduje się przycisk „Start” który uruchamia test. Podczas wykonywania testu na ekranie wypisywane są czasy w których zakończone zostają pojedyncze iteracje. Kiedy test dobiegnie końca za pomocą przycisku „Send Result"możliwa jest generacja pliku CSV zawierającego wszystkie uzyskane wyniki takie jak czasy pojedynczych operacji, użycie pamięci operacyjnej aplikacji podczas trwania testu czy też użycie procesora podczas każdej z przeprowadzanych operacji.

4.2 Schemat bazy danych

Do przeprowadzenia testów zaprojektowany został prosty schemat bazy danych. Widoczny jest na rysunku 4.4. Zawiera on następujące tablice: Autor, Książka, Wydawnictwo. Autor może mieć wiele książek zaś książka wielu autorów - użyta została tu relacja wiele do wielu. Wydawnictwo może wiele książek ale jedna książka może mieć tylko jedno wydawnictwo - zastosowana została relacja jeden do wielu.

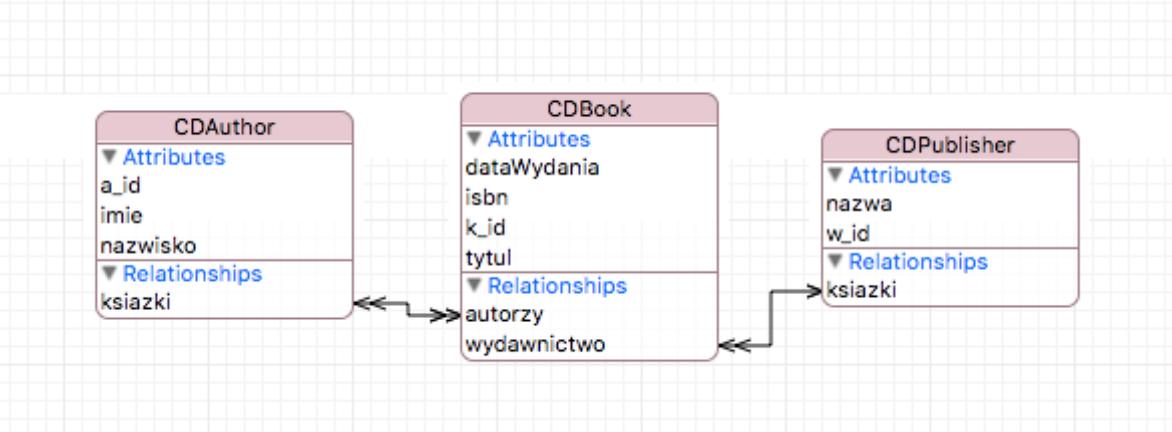
Zaprojektowany schemat bazy umożliwia przetestowanie działania baz danych przy pracy na tabelach w których są różne typy relacji a także zapewnia możliwość testowania odczytu i zapisu daty. Data jest jednym z ważniejszych elementów w mobilnych bazach da-



Rysunek 4.4: Schemat bazy danych SQL

nych ponieważ niektóre rozwiązania bazodanowe nie zapewniają wsparcia dla tego formatu danych o czym była mowa podczas opisu wybranych baz danych.

W pracy testom zostały poddane także bazy NoSQL. Dla Realm i Core Data stworzony został schemat bazy widoczny na rysunku 4.5, nie posiadający tablicy pośredniej dla relacji wiele do wielu (Autor - Książka). Tablica ta jest zbędna dla tego typu baz danych ponieważ relacja wiele do wielu przechowywana jest za pomocą listy książek w tabeli autor i listy autorów w tablicy książka.



Rysunek 4.5: Schemat basy danych dla Realm i Core Data

Dla domyślnej bazy użytkownika - User Defaults struktura zapisywanych obiektów wygląda tak samo jak schemat dla baz NoSQL widoczny na rysunku 4.5.

5 Opis przeprowadzonych testów

W rozdziale zostały opisane testy przeprowadzone w ramach pracy. Przedstawiono kroki potrzebne do otrzymania poprawnych wyników w zależności od typu testu oraz poka-

zano w jaki sposób został mierzony czas uzyskania rezultatu każdej operacji.

5.1 Testowane operacje

W ramach analizy zostały przetestowane operacje CRUD - zapis, odczyt, uaktualnienie i usuwanie danych. W celu uzyskania jak najbardziej poprawnych wyników testy zostały przeprowadzone stukrotnie a wyniki przedstawiane w analizie są uśrednieniem wszystkich stu operacji dla danego testu. Przeprowadzenie operacji więcej niż jeden raz ma ogromne znaczenie ponieważ otrzymane czasy operacji zależne są od obciążenia urządzenia w danym czasie. Wszystkie testy zostały przeprowadzone na urządzeniu iPhone 6s.

Testy były przeprowadzone dla następujących operacji:

- Zapis
  - Zapisz wszystkich danych do bazy
- Odczyt
  - Odczyt wszystkich danych z tabel
  - Wyszukanie wszystkich autorów o imieniu "Diena"
  - Wyszukanie 2 książek z największa liczbą autorów
  - Odczyt maksymalnie 20 wydawnictw z największą liczba wydanych książek i posortowanie wyniku rosnąco
- Edycja
  - Edycja imienia autora "Dienańa Ąlona"
  - Edycja daty wydania książek na obecna datę
- Usuwanie
  - Usunięcie wszystkich danych
  - Usunięcie wszystkich autorów którzy wydali 3 książki
  - Usunięcie wydawnictw które wydały książki o tytułach Ąnnie Oakleyłub "Tokyo Zombie (Tky zonbi)"

W celu otrzymania dokładnych wyników świadczących o wydajności bazy danych testy były projektowane tak aby operowały one na wszystkich danych zawartych w tabelach a także na niektórych z wybranych podczas zapytania. Zapewnia to lepszy obraz wydajności bazy danych kiedy istnieje potrzeba przetworzenia wszystkich danych w danej tabeli lub kiedy należy przeprowadzić operacje na kilku rekordach.

5.2 Dane testowe

Jak wspomniano w rozdziale 4 podczas opisu aplikacji, przed wykonaniem testu możliwy jest wybór zestawu danych testowych. Zestaw duży to tysiąc danych, zestaw średni to sto danych testowych a zestaw mały to 10 danych. Liczba danych dla każdej z tabel (Autor, Książka, Wydawnictwo) w zależności od wielkości zestawu danych przedstawiony został w tabeli 1.

Tabela 1: Liczba rekordów w zestawach danych testowych

	Mały	Średni	Duży
Wydawnictwo	1	10	100
Książka	6	60	600
Autor	3	30	300

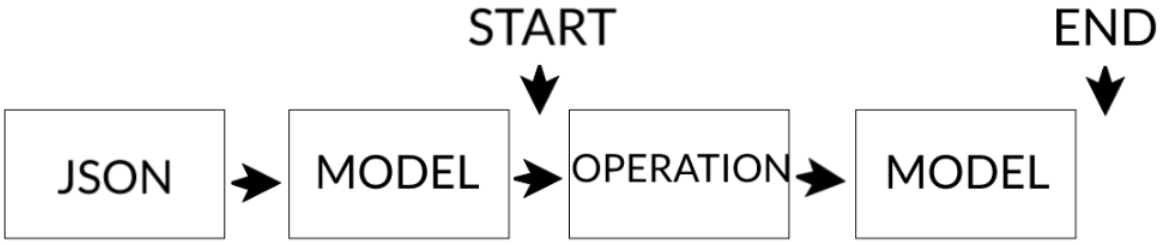
Różna ilości danych ma na celu pokazanie podczas testów zmian wydajności każdej z baz. Każda z baz inaczej będzie sobie radzić z różną ilością danych. Dane testowe zostały wygenerowane za pomocą serwisu internetowego MOCKAROO.

5.3 Przebieg testów

W pracy skupiono się na wydajności baz danych, ale z powodu testowania różnych typów baz należało przyjąć odpowiednie zasady testów.

Bazy Core Data czy Realm w wyniku zapytania zwracają obiekty, różniące się od siebie pod względem implementacji ale zawierające te same pola zawarte w modelu bazy. Różnice te wynikają z różnego sposobu działania baz danych, przedstawione one zostały w kolejnych rozdziałach pracy. Bazy SQLite i FMBD w rezultacie zapytania zwracają surowe dane a nie obiekty. Jasne staje się że czasy otrzymania obiektu względem otrzymania surowego rezultatu będą różne. Aby otrzymać miarodajne rezultaty testów założono więc, że

wynikiem końcowym każdego z testów odczytu będzie otrzymanie listy obiektów programu a nie samych surowych danych.



Rysunek 5.1: Przebieg testów

Cały proces testów przedstawia rysunek 5.1. Pierwszym etapem jest przekształcenie plików w formacie JSON na obiekty programu, których typ jest zależny od wybranej bazy danych. Kolejny z etapów to przekazanie obiektów do odpowiedniej bazy danych i zaczęcie wykonywania operacji, od tego momentu mierzony jest czas operacji. Po wykonaniu operacji otrzymany rezultat operacji przekształcany jest ponownie na obiekty programu i po zakończeniu tej operacji następuje koniec pomiaru czasu operacji.

W zależności od typu badanej operacji zastosowane zostały różne scenariusze testów. Zależnie od operacji można rozróżnić trzy typy scenariuszy:

- Zapis danych - w pętli wykonywane są następujące operacje:
    1. Wygenerowanie danych
    2. Wyczyszczenie bazy danych
    3. Wykonanie zapisu danych do bazy
- Odczyt danych:
    1. Wygenerowanie danych
    2. Wyczyszczenie bazy danych
    3. Wykonanie zapisu danych do bazy
    4. W pętli wykonywana zostaje operacja odczytania danych i pomiar czasów
- Usuwanie i edycja danych - w pętli wykonywane są następujące operacje:

1. Wygenerowanie danych
2. Wyczyszczenie bazy danych
3. Wykonanie zapisu danych do bazy
4. Wykonanie operacji usuwania lub edycji danych i pomiar czasów

Ilość wykonywania powtórzeń operacji poprzez pętle zostaje zdefiniowana przez użytkownika w ekranie aplikacji widocznym na rysunku 4.3, poprzez wprowadzenie liczby testów w polu tekstowym. Jak można zauważyć najbardziej czasochłonnymi testami jest pomiar czasu edycji i usuwania danych. Podczas tego testu wielokrotnie jest powtarzana operacja zapisu do danych.

6 Różnice implementacji baz danych

W rozdziale zostały pokazane różnice i podobieństwa w implementacji poszczególnych rozwiązań bazodanowych. Przedstawiono fragmenty kodów wykonujące te same operacje przy użyciu różnych baz danych. Opisany też został stopień skomplikowania każdej z operacji.

6.1 Modele danych

Modele danych definiowane są różnie w zależności od wykorzystanej bazy danych. W przypadku Core Data używany jest edytor w środowisku programistycznym XCode opisany w drugim rozdziale pracy. Stworzony schemat bazy widoczny jest na rysunku 4.5. Do poprawnego działania bazy Core Data wymagane jest wygenerowanie klas reprezentujących zaprojektowane tabele. Klasy te mogą zostać stworzone na kilka sposobów: ręcznie, za pomocą wbudowanego narzędzia w XCode lub za pomocą dodatkowego programu typu „mogenerator".

Bazy SQL przedstawione w pracy używają standardowych poleceń SQL do stworzenia tabel i relacji pomiędzy nimi. Przykład kodu źródłowego 6.1 prezentuje zapytania użyte w stworzonym do testów programie.

```
1    let createBookTableString = "CREATE TABLE IF NOT EXISTS Book (id
     INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT, isbn TEXT, publishDate
     TEXT, publisher_id INTEGER, FOREIGN KEY(publisher_id) REFERENCES
     Publisher(id))"
2    let createAuthorTableString = "CREATE TABLE IF NOT EXISTS Publisher (
     id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT)"
3    let createPublisherTableString = "CREATE TABLE IF NOT EXISTS
     Publisher (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT)"
```

Kod źródłowy 6.1: Polecenia tworzenia tabel w SQLite i FMDB

W przypadku dokumentowej bazy Realm kolekcje tworzone są na podstawie obiektów dziedziczących z klasy *Object* biblioteki Realm. Kod źródłowy 6.2 pokazuje przykład klasy.

```
1  class RAuthor: Object {
2
3      @objc dynamic var id: Int = 0
4      @objc dynamic var firstName = ""
5      @objc dynamic var lastName = ""
6      var books = List<RBook>()
7
8      override static func primaryKey() -> String? {
9          return "id"
10     }
11 }
```

Kod źródłowy 6.2: Przykład obiektu bazy Realm

W liniach 3-5 przedstawione są zmienne przechowujące dane obiektu, linia 6 pokazuje listę obiektów *RBook*. Lista ta jest odwzorowaniem relacji pomiędzy tablicami. Metoda *primaryKey()* znajdująca się w linii numer 10 oznacza pole odpowiadające za klucz główny danej klasy.

Domyślna Baza Użytkownika jak wspomniane zostało w rozdziale 2.5 do konwersji obiektów wymaga rozszerzenia obiektów o protokół *NSCoding*. Przykład kodu klasy Domyślnej Bazy Użytkownika reprezentuje kod źródłowy 6.3. Tak jak w w przypadku Realm linie 2-4 reprezentują pola przechowujące dane za w linii 5 przedstawiona jest lista identyfikatorów książek wydanych przez danego autora. Lista ta reprezentuje relacje pomiędzy obiektami. Metoda *encode(with aCoder: NSCoder)* potrzebna jest do poprawnego zapisu obiektu w pamięci zaś funkcja *convenience init(coder aDecoder: NSCoder)* odpowiada za odczyt obiektu.

```
1  class UDAuthor: NSObject, NSCoding {
2      var authorId: Int
3      var firstName: String
4      var lastName: String
5      var books: [Int]
6
7      init(authorId: Int, firstName: String, lastName: String, books: [Int
       ]) {
8          self.authorId = authorId
9          self.firstName = firstName
10         self.lastName = lastName
11         self.books = books
12     }
13
14     func encode(with aCoder: NSCoder) {
15         aCoder.encode(authorId, forKey: "authorId")
16         aCoder.encode(firstName, forKey: "firstName")
17         aCoder.encode(lastName, forKey: "lastName")
18         aCoder.encode(books, forKey: "books")
19     }
20
21     required convenience init(coder aDecoder: NSCoder) {
22         let authorId = aDecoder.decodeInteger(forKey: "authorId")
23         let firstName = aDecoder.decodeObject(forKey: "firstName") as!
       String
24         let lastName = aDecoder.decodeObject(forKey: "lastName") as!
       String
25         let books = aDecoder.decodeObject(forKey: "books") as! [Int]
26
27         self.init(authorId: authorId, firstName: firstName, lastName:
       lastName, books: books)
28     }
29 }
```

Kod źródłowy 6.3: Przykład obiektu Domyślnej Bazy Użytkownika

Można zauważyć znaczące różnice w tworzeniu struktury danych w każdej z przedstawionych baz. Core Data dzięki dobremu wsparciu ze strony XCode może wydawać się prosta w zaimplementowaniu lecz wymaga dobrej znajomosci całego framework-a. W celu

szybszej implementacji klas reprezentujących tabele trzeba skorzystać z oddzielnych narzędzi co może przysporzyć dodatkowych problemów w dalszych procesach utrzymywania oprogramowania. SQLite i FMDB wymagają dobrej znajomości języka SQL, ale dobrze napisane polecenia są w stanie posłużyć przez wiele lat gdyż język SQL jest jednym z najstabilniejszych i najpopularniejszych rozwiązań bazodanowych. Realm wymaga stosunkowo małej ilości kodu i umiejętności aby stworzyć strukturę bazy. Klasy są bardzo przejrzyste i zbliżone do zwykłych klas obiektów w języku Swift. Domyślna Baza Użytkownika poza standardową deklaracją pól i konstruktorów wymaga jeszcze zaimplementowania metod protokołu *NSCoding*. Dodatkowo wymagane jest dodanie odpowiednich kluczy dla każdego z pól obiektu. Ilość kodu i kluczy uzależniona jest tutaj od ilości pól obiektu. Przy wielu tabelach może to stanowić znaczące problemy w implementacji oraz utrzymaniu kodu.

6.2 Zapis danych

Zapis danych w przypadku Core Data przedstawiony jest w przykładzie 6.4 znajdującym się poniżej.

```
1     private func insertPublishers(list: [Publisher]) {
2         list.forEach { (publisher) in
3             let publisherObject = CDWydawnictwo(context: stack.
    managedObjectContext)
4             publisherObject.fill(publisher: publisher)
5             guard let booksList = publisher.books else { return }
6             booksList.forEach({ (bookId) in
7                 guard let book = featchBook(bookId: bookId) else { return
     }
8                 book.wydawnictwo = publisherObject
9                 publisherObject.ksiazki.adding(book)
10            })
11        }
12        stack.saveContext()
13    }
```

Kod źródłowy 6.4: Przykład zapisu obiektu Core Data

Dla każdego obiektu z listy tworzony jest obiekt *CDWydawnictwo* będący reprezentacją rekordu w tabeli wydawnictwo. W linii za pomocą funkcji *fill* uzupełniane są pola

obiektu. Następnie dla każdej książki wydanej przez wydawnictwo w pętli pokazanej w linii 8 odczytywany jest obiekt *CDKsiazka* wcześniej już zapisany w bazie. Po odczytaniu obiektu następuję ustawienie relacji książka - wydawnictwo co reprezentują linie 10 i 11 w przedstawionym przykładzie. Po wykonaniu opisanych operacji w linii numer 15 następuję zapis stanu bazy Core Data.

Podczas użycia baz SQLIte i FMDB aby zapisać obiekt do bazy należy w pierwszej kolejności zdefiniować odpowiednie zapytania. Aby poprawnie wprowadzać dane do przedstawionej w pracy bazie testowej zostały użyte zapytania SQL widoczne w przykładzie 6.5.

```
1    private let insertBookDataQueryString = "INSERT INTO Book (title,
     isbn, publishDate) VALUES (?, ?, ?)"
2    private let insertAuthorDataQueryString = "INSERT INTO Author (
     firstName, lastName) VALUES (?, ?)"
3    private let insertPublisherDataQueryString = "INSERT INTO Publisher (
     name) VALUES (?)"
4    private let insertAuthorsBooksDataQueryString = "INSERT INTO
     authors_books (book_id, author_id) VALUES (?, ?)"
```

Kod źródłowy 6.5: Zapytania SQL do wprowadzania danych

Kolejnym krokiem jest odpowiednie użycie przedstawionych powyżej zapytań oraz przypisanie odpowiednich pól obiektu do zapytania. Czynności te dla bazy SQLite zostały pokazane w fragmencie kodu 6.6. Przykład pokazuję zapis obiektu *Książka*. Aby zapisać obiekt niezbędna jest zmienna typu *OpaquePointer* reprezentująca obiekt zapytania do bazy danych. Za pomocą funkcji widocznej w 4 linii *sqlite3_prepare_v2* zapytanie z formy ciągu znaków konwertowane jest na obiekt. Następnie w liniach 8-11 za pomocą funkcji *sqlite3_bind_text* pola obiektu *Książka* przypisywane są do zapytania. W linii 13 przy użyciu *sqlite3_step* następuję wykonanie zapytania i zapis obiektu do tablicy. Funkcja *sqlite3_reset* czyści przypisane w zapytaniu zmienne i umożliwia ponowne załadowane nowych danych do zapytania. W linii 20 przykładu następuję zapis stanu bazy danych przy użyciu *sqlite3_finalize*.

```
1     private func insertBooks(list: [Book]) {
2         var insertDataStatement: OpaquePointer?
3
4         if sqlite3_prepare_v2(db, insertBookDataQueryString, -1, &
      insertDataStatement, nil) != SQLITE_OK {
5             print("error create database statement")
6         }
7
8         list.forEach { (book) in
9             sqlite3_bind_text(insertDataStatement!, 1, (book.title as!
      NSString).utf8String, -1, nil)
10            sqlite3_bind_text(insertDataStatement!, 2, (book.isbn as!
      NSString).utf8String, -1, nil)
11            sqlite3_bind_text(insertDataStatement!, 3, (book.stringDate
      as NSString).utf8String, -1, nil)
12
13            if sqlite3_step(insertDataStatement!) != SQLITE_DONE {
14                print("Could not insert row into Book table")
15            }
16
17            sqlite3_reset(insertDataStatement!)
18        }
19
20        sqlite3_finalize(insertDataStatement)
21    }
```

Kod źródłowy 6.6: Przykład zapisu obiektu SQLIte

Przy użyciu biblioteki FMDB użycie SQLite jest znacznie łatwiejsze. Bliźniacza metoda do zapisu obiektów *Książka* została przedstawiona w przykładzie 6.7. Proces zapisu obiektu jest znacząco ułatwiony. W linii 2 poprzez funkcje *open()* otwierane jest połączenie z bazą danych. Następnie w pętli wykonywana jest operacja *executeUpdate*, która przyjmuje jako parametr zapytanie do bazy i listę pól obiektu. Po zapisaniu wszystkich obiektów do bazy w linii numer 15 wywoływana jest funkcja *close* służąca do zamknięcia połączenia z bazą danych.

```
1     private func insertBooks(list: [UDBook]) {
2         guard database.open() else {
3             print("Unable to open database")
4             return
5         }
6
7         list.forEach({ (book) in
8             do {
9                 try! database.executeUpdate(insertBookDataQueryString,
      values: [book.title, book.isbn, book.stringDate])
10            } catch {
11                print("failed: error.localizedDescription")
12            }
13        })
14
15        database.close()
16    }
```

Kod źródłowy 6.7: Przykład zapisu obiektu FMDB

Jedną z najprostszych implementacji zapisu danych posiada Domyślna Baza Użytkownika. Kod wykonujący to samo zadanie co przytoczone wcześniej przykłady reprezentuje fragment kodu 6.8. Dzięki implementacji protokołu *NSCoding* możliwe jest szybkie kodowanie i dekodowanie obiektów. W celu uzyskania formatu danych pozwalającego na zapis w Domyślnej bazie użytkownika użyta została metoda *NSKeyedArchiver.archivedData* przekształcająca listę obiektów na typ *Data* a następnie dane zostały zapisane za pomocą funkcji *userDefaults.set* a w ostatniej linii za pomocą metody *synchronize* zapisany został stan bazy.

```
1     private func insertBooks(list: [UDBook]) {
2         let encodedData = NSKeyedArchiver.archivedData(withRootObject:
      list)
3         userDefaults.set(encodedData, forKey: booksKey)
4         userDefaults.synchronize()
5     }
```

Kod źródłowy 6.8: Przykład zapisu obiektu User Defaults

Biblioteka Realm także posiada prosty interfejs zapisu danych. Metoda wykonująca

to zadanie przedstawiona została poniżej w przykładzie 6.9. Cała operacja sprowadza się do stworzenia bloku zapisu *write* i wywołaniu metody *add* z listą obiektów przeznaczoną do zapisu.

```
1     private func insertBooks(list: [RBook]) {
2         try! realm.write {
3             realm.add(list)
4         }
5     }
```

Kod źródłowy 6.9: Przykład zapisu obiektu Realm

Można zauważyć, że najmniej wymagający interfejs do zapisu danych posiada biblioteka Realm. Składa się on praktycznie z dwóch metod. Bardzo prosty zapis danych posiada także Domyślna Baza Użytkownika lecz w jej przypadku należy dodatkowo w każdym obiekcie implementować protokół *NSCoding*. Jedną z bardziej wymagających bibliotek jest Core Data. Zapis danych wymaga tutaj większej ilości kodu a także konwersji zapisywanego obiektu do obiektu reprezentującego tablice bazy. Należy też zadbać o zapisywanie kontekstu bazy w odpowiednich momentach. Najbardziej wymagające są bazy SQL. SQLite i FMDB wymagają sformułowania odpowiednich zapytań SQL, dodatkowo w przypadku SQLite implementacja wymaga pojedynczego przypisywania pól obiektu do stworzonego zapytania. Problem ten znika przy użyciu FMDB lecz w większości przypadków to rozwiązanie pomimo ułatwień w implementacji okazuję się działać znacznie wolniej od SQLite.

6.3 Odczyt danych

Odczyt danych zostanie porównany na podstawie funkcji użytej podczas testu przedstawionego w rozdziale 6.2 polegającego na wyszukaniu wszystkich autorów o imieniu „Diena".

Wybrany przykład funkcja dla Core Data został przedstawiony poniżej w kodzie źródłowym 6.10. Do odczytu danych należy zainicjalizować obiekt *NSFetchRequest* z odpowiednią nazwą tablicy na której przeprowadzona ma zostać operacja. Następnie tworzony jest *NSPredicate*, który odpowiada za odpowiednie wyszukanie rezultatu zapytania, w tym przypadku wybraniu autorów o konkretnym imieniu. Po prawidłowej konfiguracji operacja jest wykonywana za pomocą funkcji *executeFetchRequest*, która jako parametr przyjmuje

NSFetchRequest i kontekst bazy danych.

```
1     func getAuthorsByName() {
2         let name = "Diena"
3
4         let request = NSFetchRequest<NSFetchRequestResult>(entityName:
      CDAuthor.entityName())
5         let predicate = NSPredicate(format: "name == %@", name)
6
7         request.predicate = predicate
8
9         guard let result = stack.executeFetchRequest(request: request,
      inContext: stack.managedObjectContext) else { return }
10    }
```

Kod źródłowy 6.10: Przykład odczytu danych Core Data

Inny sposób przeszukiwania danych oferuje Domyślna Baza Użytkownika. Funkcja wykonująca te same zadanie została przedstawiona w przykładzie 6.11.

```
1     func getAuthorsByName() {
2         let name = "Diena"
3         var result = [UDAuthor]()
4
5         if  let authorsData = userDefaults.data(forKey: authorsKey) {
6             let authorsList = NSKeyedUnarchiver.unarchiveObject(with:
      authorsData) as! [UDAuthor]
7             result = authorsList.filter { firstName == name }
8         }
9     }
```

Kod źródłowy 6.11: Przykład odczytu danych User Defaults

W 5 linii za pomocą zdefiniowanego klucza dla tabeli odczytywane są wszystkie dane dla tablicy Autor. Następnie w linii numer 6 następuję konwersja surowego typu danych na obiekty. Kolejnym krokiem jest przeszukanie listy i wybranie autorów o podanym imieniu.

Bazy SQL do odczytania danych wymagały zdefiniowana zapytania widocznego poniżej. Zapytanie te zostało użyte dla bazy SQLite i FMDB.

```
1 let selectAuthorByNameQueryString = "SELECT id, firstName, lastName FROM
    Author a~WHERE a.firstName = ?"
```

Kod źródłowy 6.12: Zapytanie SQL do odczytu danych

W przypadku użycia SQLite odczyt danych przeprowadzony był poprzez poprzez funkcje widoczna w fragmencie kodu 6.13

```
1     func getAuthorsByName() {
2         let name = "Diena"
3         var result = [Author]()
4         var stmtAuthors: OpaquePointer?
5
6         if sqlite3_prepare(db, selectAuthorByNameQueryString, -1, &
    stmtAuthors, nil) != SQLITE_OK {
7             let errmsg = String(cString: sqlite3_errmsg(db)!)
8             print("error preparing insert: errmsg")
9             return
10        }
11
12        sqlite3_bind_text(stmtAuthors!, 1, (name as! NSString).utf8String
    , -1, nil)
13
14        while(sqlite3_step(stmtAuthors) == SQLITE_ROW) {
15            let id = Int(sqlite3_column_int(stmtAuthors, 0))
16            let firstName = String(cString: sqlite3_column_text(
    stmtAuthors, 1))
17            let lastName = String(cString: sqlite3_column_text(
    stmtAuthors, 2))
18
19            result.append(Author(authorId: id, firstName: firstName,
    lastName: lastName))
20        }
21    }
```

Kod źródłowy 6.13: Przykład odczytu danych SQLite

Ponownie w przypadku SQLite wcześniej zdefiniowane zapytanie zostaje przypisane do obiektu typu *OpaquePointer* za pomocą funkcji *sqlite3_prepare*. Następnie w linii 12 do

zapytania zostaje przypisany parametr oznaczony za pomocą "?"w zapytaniu. W linii numer 14 za pomocą funkcji *sqlite3_step* w pętli odczytywane są kolejne rekordy będące wynikiem zapytania. Każde z pól obiektu musi zostać odczytane i sformatowane na odpowiedni typ.

Operacja odczytu została ułatwia FMDB w stosunku do SQLite. Funkcja wykonująca tą samą operacje przy użyciu bazy FMDB została przedstawiona poniżej.

```
1  func getAuthorsByName() {
2          let name = "Diena"
3          var authorsList = [Author]()
4
5          guard database.open() else {
6              print("Unable to open database")
7              return
8          }
9
10         do {
11             let result = try database.executeQuery(
       selectAuthorByNameQueryString, values: [name])
12
13             while(result.next()) {
14                 let id = Int(result.int(forColumnIndex: 0))
15                 let firstName = result.string(forColumnIndex: 1)
16                 let lastName = result.string(forColumnIndex: 2)
17
18                 authorsList.append(Author(authorId: id, firstName:
       firstName, lastName: lastName))
19             }
20         } catch {
21             print("failed: (error.localizedDescription)")
22         }
23
24         database.close()
25     }
```

Kod źródłowy 6.14: Przykład odczytu danych FMDB

FMDB po otworzeniu połączenia z bazą umożliwia wykonanie zapytania za pomocą funkcji *executeQuery* i podaniu argumentów zapytania jako parametry funkcji. Następnie

można iterować po kolejnych rekordach będących wynikiem zapytania. Rezultaty nie wymagają ponownego parsowania na odpowiednie typy. Po zakończeniu operacji należy pamiętać o zamknięciu połączenia z bazą danych za pomocą funkcji *close*.

Jedną z najprostszych implementacji posiada biblioteka Realm. Przykład reprezentuje fragment kodu 6.15.

```
1     func getAuthorsByName() {
2         let name = "Diena"
3         let predicate = NSPredicate(format: "firstName = %@", name)
4         let result = realm.objects(RAuthor.self).filter(predicate)
5     }
```

Kod źródłowy 6.15: Przykład odczytu danych Realm

Realm tak samo jak Core Data używa *NSPredicate* do sformułowania zapytania odczytu danych. W linii 4 za pomocą funkcji *object()*, podając odniesienie do klasy, której rezultat chcemy otrzymać istnieje możliwość odczytania wszystkich rekordów. Następnie za pomocą funkcji *filter*, której argumentem jest wcześniej zdefiniowany predykant następuje filtrowanie wyniku.

Odczyt danych jest inny dla każdej z bibliotek. Core Data pomimo dość prostej składni tworzenia zapytań posiada często problematyczne w użyciu sformułowania predykantów. Domyślna Baza Użytkownika operuje jedynie na funkcjach filtrowania oferowanych przez środowisko przez co często operacje są znacznie wolniejsze niż w przypadku innych rozwiązań bazodanowych. Dużym problemem jest też sytuacja kiedy wyszukanie rezultatu wymaga przeszukania relacji lub filtrowania wielu parametrów. W takich przypadkach wymagane jest stworzenie wielu funkcji co niesie za sobą duży nakład kodu. SQLite posiada problematyczne przypisywanie argumentów do zapytania a także wymaga parsowania odczytanych pól na odpowiednie typy. FMDB rozwiązuje te problemy lecz ilość kodu potrzebnego na zaimplementowanie jest zbliżona do implementacji SQLite. Jeden z najprostszych sposobów implementacji posiada Realm. Odczytanie danych sprowadza się jedynie do kilku linii kodu. Oferuje on także możliwość używania predykatów tak samo jak w przypadku Core Data a także zwykłego filtrowania wyniku jak Domyślna Baza Użytkownika.

6.4 Usuwanie danych

Usuwanie danych zostanie porównane na podstawie funkcji użytej podczas testu przedstawionego w rozdziale 6.3 polegającego na usunięciu wszystkich autorów, którzy wydali 3 książki.

Kod źródłowy 6.16 przedstawia przykład usuwania danych przy użyciu biblioteki Core Data. W usuwaniu danych tak samo jak podczas odczytu tworzony jest *NSFetchRequestResult* za pośrednictwem którego odczytywane są dane z określonej tabeli. Następnie otrzymany rezultat jest filtrowany. Po przefiltrowaniu w linii 9 wykonywana jest operacja usuwania wybranych rekordów. Standardowo po wszystkich operacjach następuje zapis stanu bazy danych.

```
1     func removeAuthorsByBooks() {
2         let requestAuthors = NSFetchRequest<NSFetchRequestResult>(
      entityName: CDAutor.entityName())
3
4         guard var result = stack.executeFetchRequest(request:
      requestAuthors, inContext: stack.managedObjectContext) as? [CDAutor]
      else { return }
5         result = result.filter { 0.ksiazki.count == 3 }
6
7         result.forEach { (author) in
8             stack.managedObjectContext.delete(author)
9         }
10
11        stack.saveContext()
12    }
13
```

Kod źródłowy 6.16: Przykład usuwania danych Core Data

Kod źródłowy 6.17 reprezentuje usuwanie danych w Domyślnej Bazie Użytkownika. W przykładzie widać słabe strony tego rozwiązania bazodanowego. W linii 4 następuje filtrowanie listy zapisanych rekordów w taki sposób aby wyszukać identyfikatory autorów których należy usunąć. W linii 5 wykonywane jest ponowne przeszukanie listy rekordów i usunięcie wybranych danych. W przypadku złożonych operacji podczas używania Domyślnej Bazie Użytkownika często istnieje potrzeba przeszukiwania kilkukrotnie więcej niż

jednej listy. Często jest to problematyczne w zaimplementowaniu oraz znacząco spowalnia aplikacje.

```
1          func removeAuthorsByBooks() {
2          if  let authorsData = userDefaults.data(forKey: authorsKey) {
3              let authorsList = NSKeyedUnarchiver.unarchiveObject(with:
       authorsData) as! [UDAuthor]
4              let resultIdsToRemove = authorsList.filter { 0.books.count ==
        3 }.map { 0.authorId }
5              let result = authorsList.filter { !resultIdsToRemove.contains
       (0.authorId) }
6              let encodedData = NSKeyedArchiver.archivedData(withRootObject
       : result)
7              userDefaults.set(encodedData, forKey: authorsKey)
8              userDefaults.synchronize()
9          }
10      }
```

Kod źródłowy 6.17: Przykład usuwania danych User Defaults

Bazy SQL do usuwania danych wymagały zdefiniowana zapytania widocznego poniżej. Zapytanie te zostało użyte dla bazy SQLite i FMDB.

```
1 let deleteAuthorsByBooks = "DELETE FROM Author WHERE EXISTS (SELECT
      authorId FROM ( SELECT Author.id as authorId, COUNT(authors_books.
      book_id) as booksCount FROM Author, authors_books WHERE Author.id =
      authors_books.author_id GROUP BY Author.id) WHERE booksCount = 3)"
```

Kod źródłowy 6.18: Zapytanie SQL do usuwania danych

Przykłady kodów źródłowych 6.19 (SQLite) i 6.20 (FMDB) pokazują, że implementacja tego samego zadania nie różni się znacząco. Największą różnice pomiędzy nimi stanowi składnia. SQLite posiada składnie pokrewną dla języka C zaś biblioteka FMDB pomimo wykorzystywania SQLite udostępnia wszystkie funkcja charakterystyczne dla języka Swift.

```
1     func removeAuthorsByBooks() {
2         var stmtAuthors: OpaquePointer?
3
4         if sqlite3_prepare(db, deleteAuthorsByBooks, -1, &stmtAuthors,
      nil) != SQLITE_OK {
5             let errmsg = String(cString: sqlite3_errmsg(db)!)
6             print("error preparing insert: (errmsg)")
7             return
8         }
9
10        sqlite3_step(stmtAuthors!)
11        sqlite3_finalize(stmtAuthors)
12    }
```

Kod źródłowy 6.19: Przykład usuwania danych SQLite

```
1     func removeAuthorsByBooks() {
2         guard database.open() else {
3             print("Unable to open database")
4             return
5         }
6
7         do {
8             try database.executeStatements(deleteAuthorsByBooks)
9         } catch {
10            print("failed: (error.localizedDescription)")
11        }
12
13        database.close()
14    }
```

Kod źródłowy 6.20: Przykład usuwania danych FMDB

Najmniej wymagający interfejs posługiwania się bazą danych zaprezentowany został w przykładzie 6.21. Biblioteka Realm wymaga jedynie przefiltrowania listy rekordów a następnie przekazanie rezultatu jako argument funkcji *delete*.

```
1      func removeAuthorsByBooks() {
2          try! realm.write {
3              let authors = realm.objects(RAuthor.self).filter { 0.books.
       count == 3 }
4              realm.delete(authors)
5          }
6      }
```

Kod źródłowy 6.21: Przykład usuwania danych Realm

Przykłady usuwania danych bardzo dobrze pokazują w jakim stopniu różni się sposób implementacji operacji na danych w przypadku różnych bibliotek. Core Data posiada swój charakterystyczny sposób przeprowadzania operacji za pomocą *NSFetchRequest*. Implementacja SQLite i FMDB są do siebie zbliżone. W przypadku SQLite implementacja może wydawać się trudniejsza ze względu na interfejs C biblioteki. Domyślna Baza Użytkownika jest doskonałym rozwiązaniem do prostych operacji. Podczas skomplikowanych operacji na danych ilość kodu i czas poświęcony na implementacje wzrasta. Najmniej wymagającą biblioteką jest dokumentowa baza Realm. Nie wymaga ona znaczących umiejętności od programisty a także przeprowadzenie operacji na danych sprowadza się najczęściej do wywołania paru metod udostępnionych przez bibliotekę.

7 Analiza

W rozdziale zostały przedstawione wyniki przeprowadzonych testów. Dane zostały przedstawione w tabelach oraz za pomocą wykresów. Każdy z rezultatów testów został zanalizowany i opisany.

7.1 Testy zapisu danych

W podrozdziale zostały przedstawione wyniki testów zapisu danych. Dodatkowo przedstawiono ilość wykorzystanej pamięci operacyjnej podczas operacji oraz stopień wykorzystania procesora. W systemie iOS nie istnieje możliwość odczytania zajętości pamięci operacyjnej i zużycia procesora jedynie dla danej operacji. Przedstawione dane pokazują więc parametry dla całej działającej aplikacji. Wyniki te są jak najbardziej miarodajne, gdyż każdy z testów odbywał się na ”świeżo„ uruchomionej aplikacji i użycie pamięci operacyjnej i procesora zawsze miało stały punkt startowy.

7.1.1 Mały zestaw danych

Prezentacja danych w formie tabel:

Tabela 2: Czasy zapisu danych do bazy - mały zestaw danych

Baza danych	Czas zapisu [ms]
User Defaults	2,07
Realm	5,52
Core Data	16,05
FMDB	173,63
SQLite	175,45

Tabela 3: Rozmiar pliku wynikowego bazy danych - mały zestaw danych

Baza danych	Rozmiar pliku bazy danych [kb]
User Defaults	0,04
Realm	16,38
FMDB	61,23
SQLite	61,32
Core Data	80,04

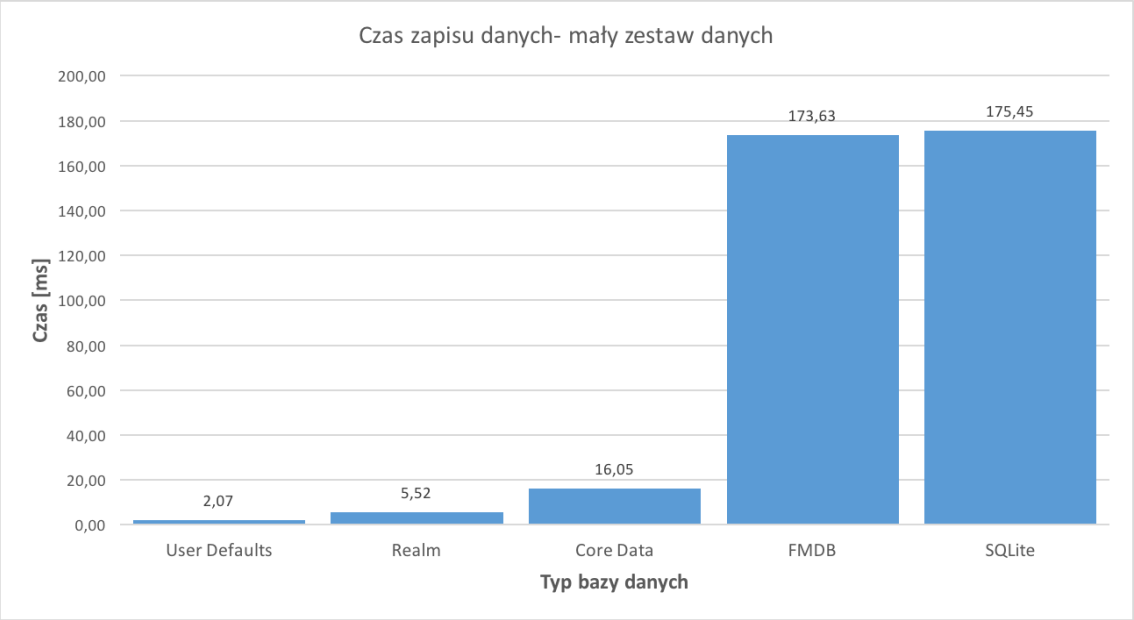
Tabela 4: Użycie procesora podczas operacji - mały zestaw danych

Baza danych	Użycie procesora [%]
SQLite	18
FMDB	19
Realm	47
Core Data	83
User Defaults	88

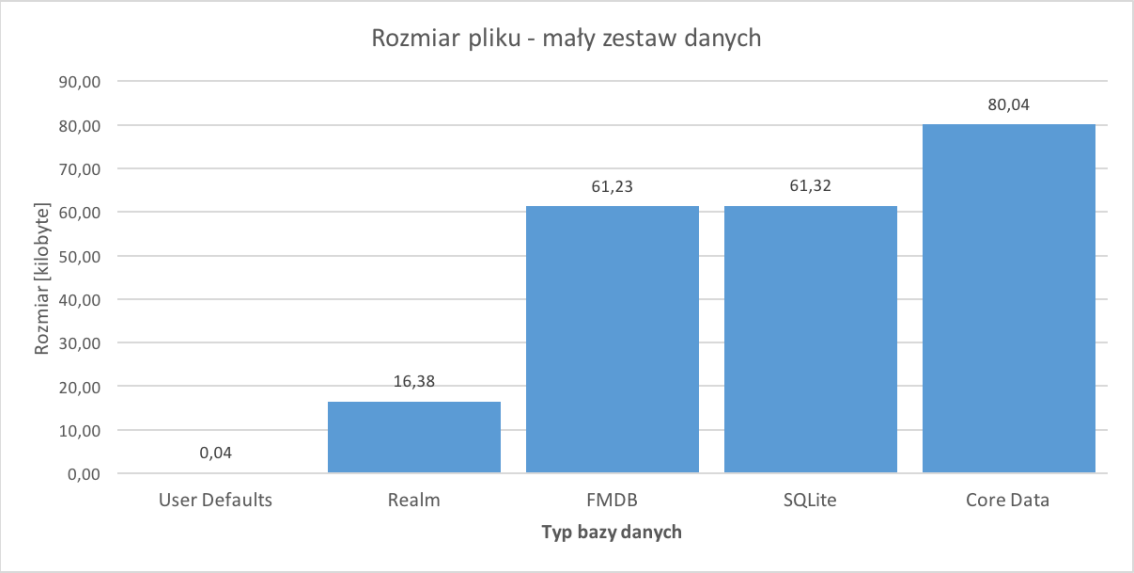
Tabela 5: Użycie pamięci operacyjnej podczas operacji - mały zestaw danych

Baza danych	Użycie pamięci RAM [mb]
User Defaults	86,4
FMDB	86,4
SQLite	86,5
Core Data	88,6
Realm	89,7

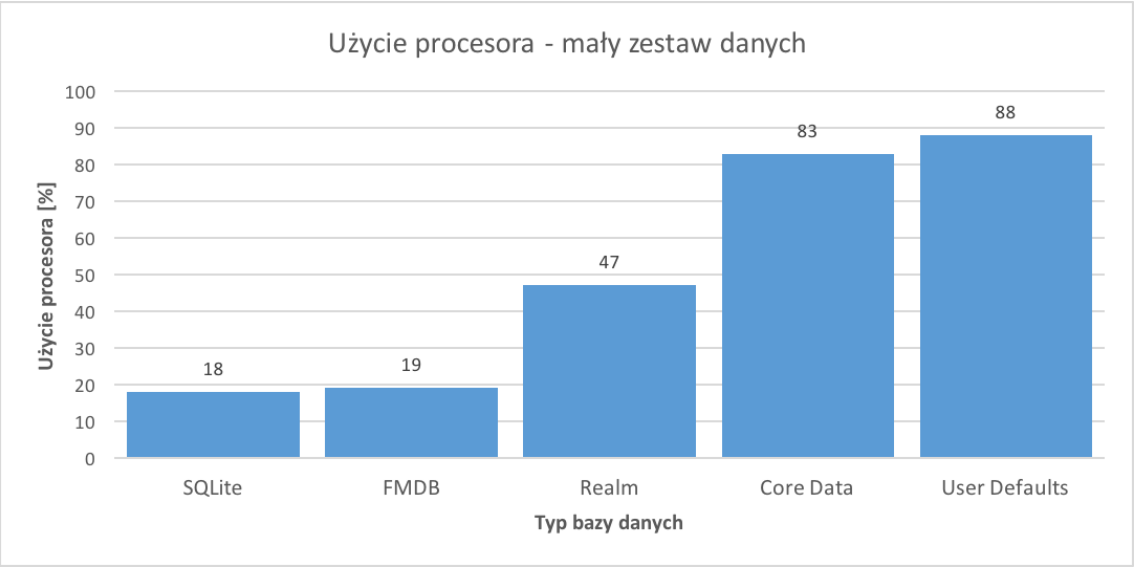
Prezentacja wyników w formie wykresów:



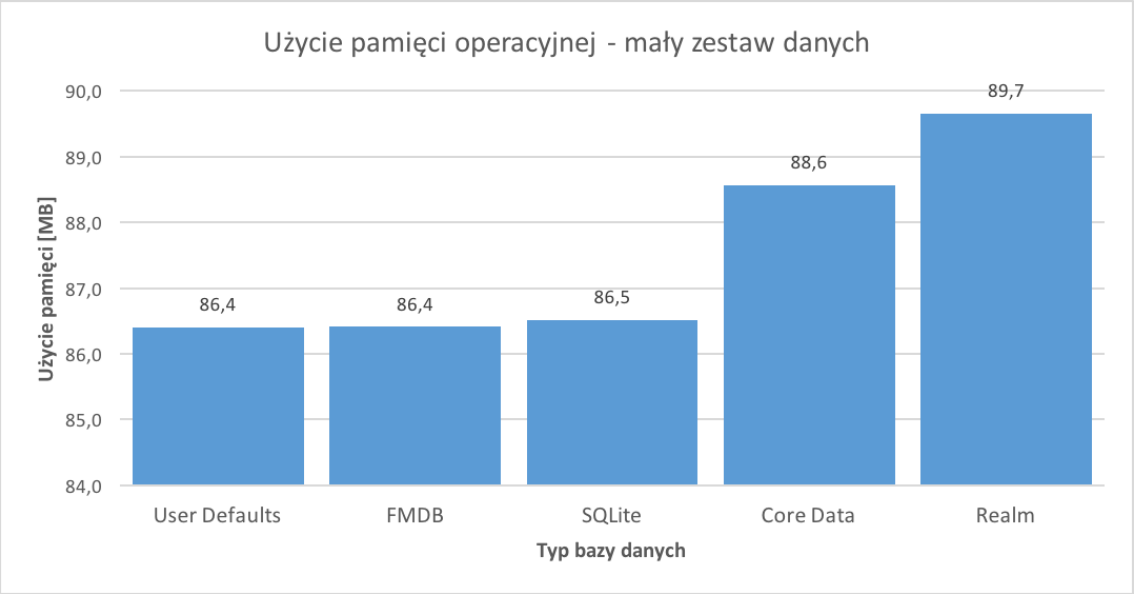
Rysunek 7.1: Czasy zapisu danych do bazy - mały zestaw danych



Rysunek 7.2: Rozmiar pliku wynikowego bazy danych - mały zestaw danych



Rysunek 7.3: Użycie procesora podczas operacji - mały zestaw danych



Rysunek 7.4: Użycie pamięci operacyjnej podczas operacji - mały zestaw danych

Czasy zapisu danych widoczne na wykresie 7.1 i w tabeli 2 pokazują że najszybszy okazała się Domyślna Baza Użytkownika, wynika to z zasady jej działania. Zapisywane są w niej listy obiektów, nie zachodzi potrzeba ustawiania relacji pomiędzy tabelami. Z czasem ponad dwukrotnie wyższym wynoszącym 5,52 ms drugi w kolejności jest Realm. Core Data uzyskała czas trzykrotnie wyższy niż Realm wynoszący 16.05 ms. Najwolniejsze okazały się bazy SQLite i FMDB uzyskując zbliżone czasy 173,63 ms i 175,45 ms.

Rozmiary plików baz widoczne na wykresie 7.2 i w tabeli 3 pokazują, że Domyślna Baza Użytkownika posiada najmniejszy rozmiar wynoszący 0.04 kb, przechowuje dane binarne w wysokim stopniu kompresji. Baza danych Realm posiada plik dużo większy, jego rozmiar wynosi 16,38 kb. SQLite i FMDB uzyskały niemal równe rozmiary plików. Zaś plik wynikowy Core Data uzyskał rozmiar największy 80 kb.

Dane pokazujące użycie procesora podczas operacji zapisu widoczne w tabeli 4 i na wykresie 7.3 dowodzą, że SQLite i FMDB używają jedynie 18-19% zasobów procesora urządzenia. Realm uzyskuje wynik ponad 50% wyższy wynoszący 47%, zaś framework Core Data osiąga aż 83%. Domyślna Baza Użytkownika mimo iż uzyskuje najlepszy czas zapisu podczas wykonywania operacji używa aż 88% zasobów procesora co jest najgorszym wynikiem.

Podczas testu przy użyciu najmniejszego zestawu danych użycie pamięci operacyjnej urządzenia jest zbliżone dla wszystkich baz danych. Rezultaty widoczne w tabeli 5 oscylują w granicach od 86 mb do 89 mb. Przy tak małej ilości danych różnice te można przyjąć za mało istotne. Kolejne testy pokażą znaczne rozbieżności wykorzystania pamięci RAM pomiędzy testowanymi bazami.

7.1.2 Średni zestaw danych

Prezentacja danych w formie tabel:

Tabela 6: Czasy zapisu danych do bazy - średni zestaw danych

Baza danych	Czas zapisu [ms]
User Defaults	4,48
Realm	36,46
Core Data	150,01
FBDB	1872,36
SQLite	1877,78

Tabela 7: Rozmiar pliku wynikowego bazy danych - średni zestaw danych

Baza danych	Rozmiar pliku bazy danych [kb]
User Defaults	0,08
Realm	106,99
SQLite	318,34
FBDB	326,34
Core Data	592,94

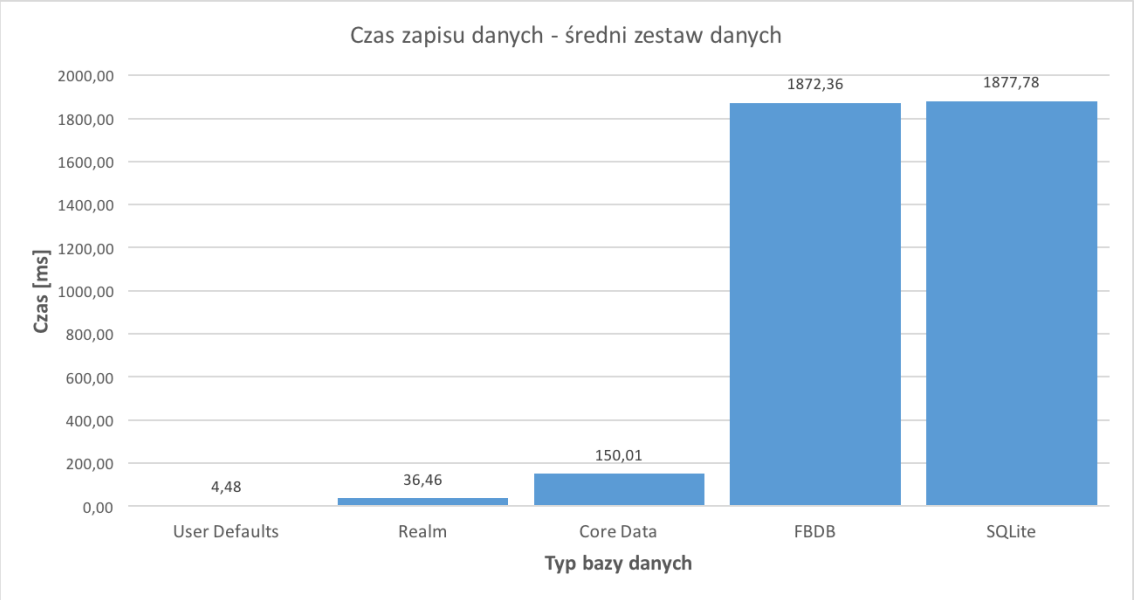
Tabela 8: Użycie procesora podczas operacji - średni zestaw danych

Baza danych	Użycie procesora [%]
SQLite	16
FMDB	19
Realm	77
User Defaults	96
Core Data	97

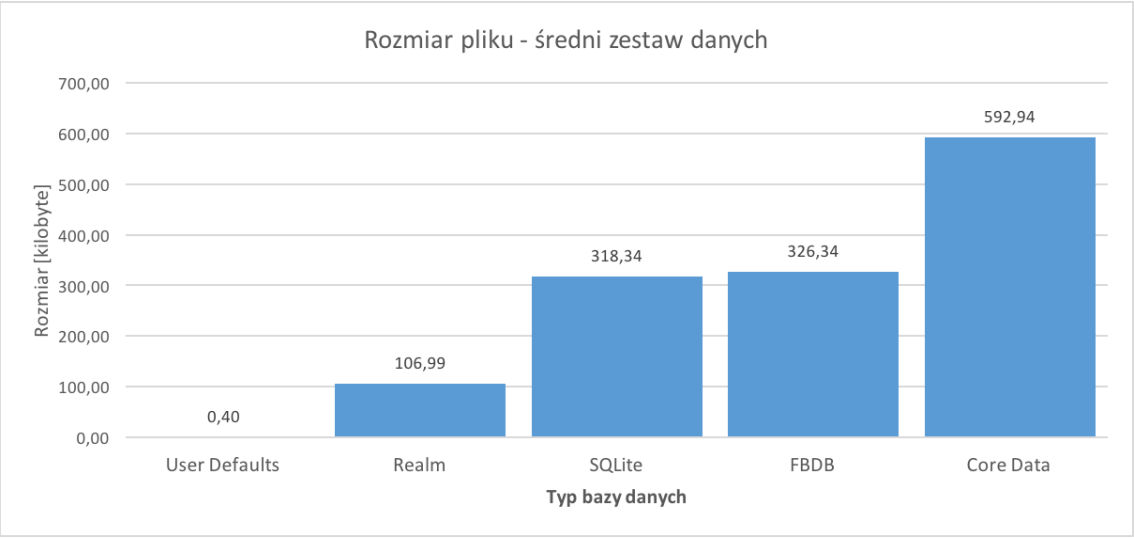
Tabela 9: Użycie pamięci operacyjnej podczas operacji - średni zestaw danych

Baza danych	Użycie pamięci RAM [mb]
User Defaults	91,3
FMDB	91,4
SQLite	91,4
Realm	94,8
Core Data	102,7

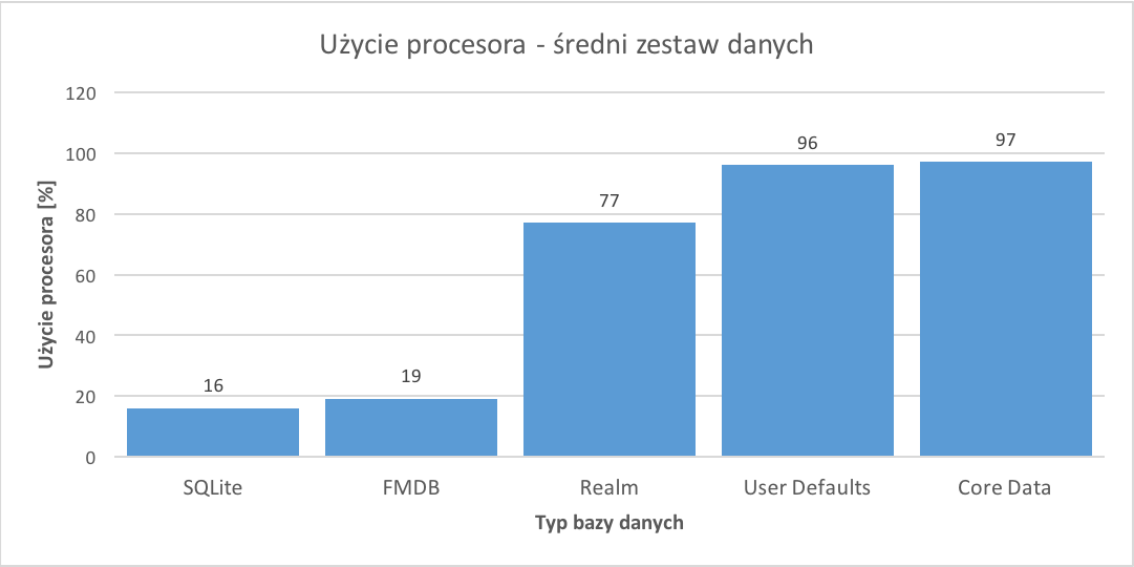
Prezentacja wyników w formie wykresów:



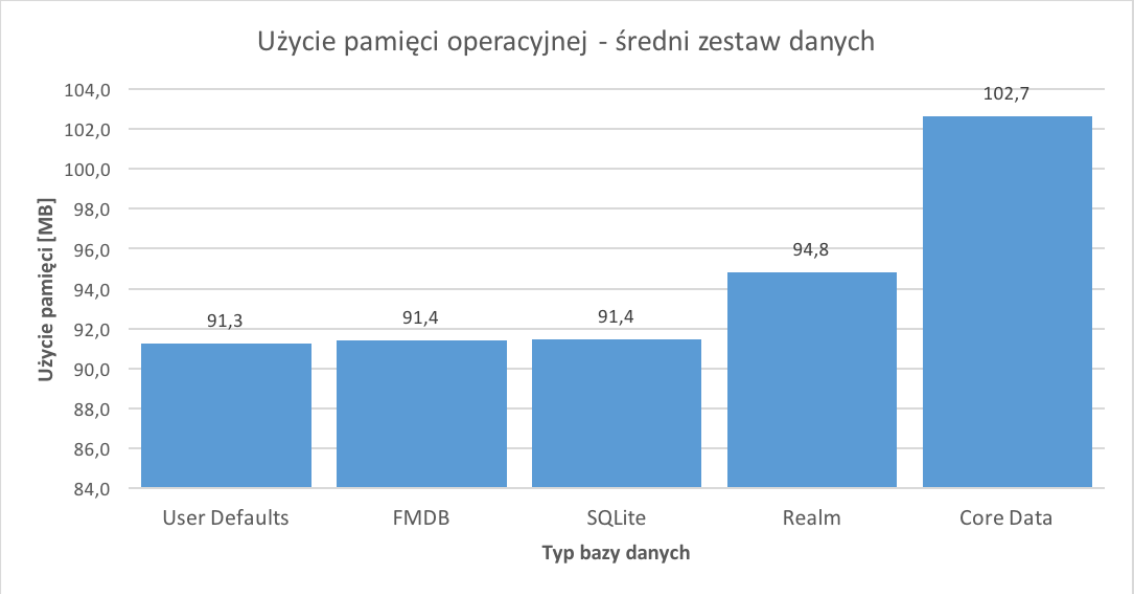
Rysunek 7.5: Czasy zapisu danych do bazy - średni zestaw danych



Rysunek 7.6: Rozmiar pliku wynikowego bazy danych - średni zestaw danych



Rysunek 7.7: Użycie procesora podczas operacji - średni zestaw danych



Rysunek 7.8: Użycie pamięci operacyjnej podczas operacji - średni zestaw danych

Czasy zapisu danych podczas testu z średnim zestawem danych widoczne na wykresie 7.5 pokazały tą samą klasyfikacje baz danych co test z użyciem małego zestawu danych. Czas potrzebny na zapis danych wzrósł o ponad 50% dla Domyślnej Bazy Użytkownika, 80% dla Realm, 85% dla Core Data. SQLite i FMDB potrzebowały ponad 90% czasu więcej aby zapisać dane średniego zestawu.

Podobna sytuacja występuje przy wynikach wielkości plików wynikowych zaprezentowanych na wykresie 7.6. Klasyfikacja baz jest identyczna jak w poprzednim teście. Plik Domyślnej Bazy Użytkownika osiągnął rozmiar o 90% większy. Realm, SQLite, FMDB i Core Data osiągnęły podobny przyrost pliku wynoszący 85%.

Użycie procesora podczas testu z średnim zestawem danych widoczne na wykresie 7.7 w dalszym ciągu dowodzi, że pomimo wzrostu ilości danych SQLite i FMDB zużywa najmniej zasobów procesora. W przypadku Realm wykorzystanie CPU wzrosło o 40%. Core Data i Domyślna baza użytkownika przy większej ilości danych używały procesora w 96-97%.

Wykorzystanie pamięci operacyjnej dla SQLite, FMDB i Domyślnej Bazy Użytkownika wzrosło o 5-6% co pokazuje wykres 7.8. Realm uzyskał lepszy rezultat względem poprzedniego testu zmniejszając wykorzystanie pamięci RAM o 5%. Core Data wypadła najgorzej osiągając 103 MB.

7.1.3 Duży zestaw danych

Prezentacja danych w formie tabel:

Tabela 10: Czasy zapisu danych do bazy - duży zestaw danych

Baza danych	Czas zapisu [ms]
User Defaults	30,48
Realm	201,93
Core Data	24912,73
FMDB	34217,17
SQLite	35044,62

Tabela 11: Rozmiar pliku wynikowego bazy danych - duży zestaw danych

Baza danych	Rozmiar pliku bazy danych [kb]
User Defaults	4,00
Realm	2588,67
SQLite	3177,88
FMDB	3264,30
Core Data	7240,79

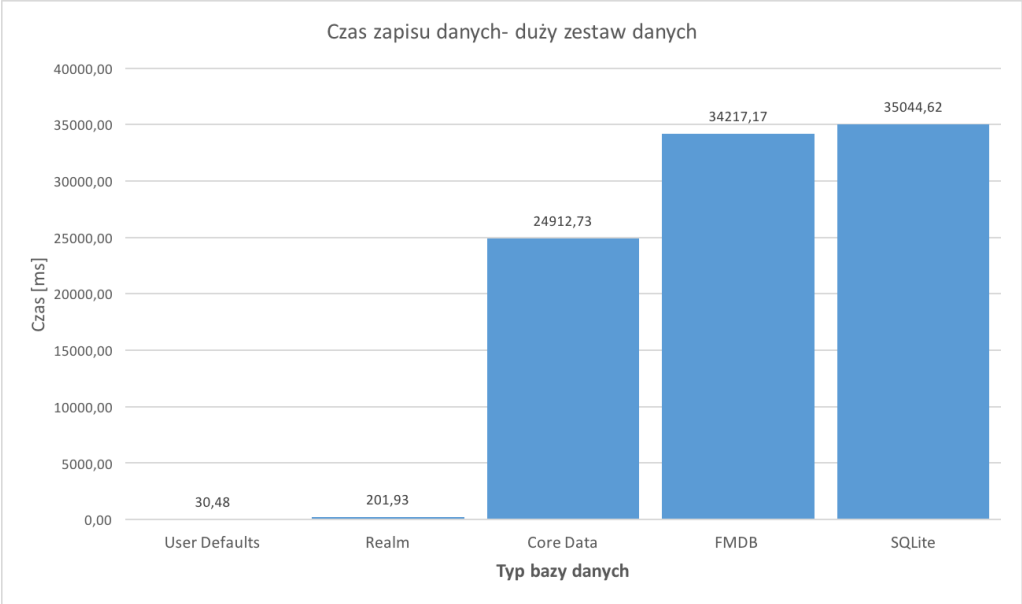
Tabela 12: Użycie procesora podczas operacji - duży zestaw danych

Baza danych	Użycie procesora [%]
SQLite	16
FMDB	19
Realm	88
Core Data	98
User Defaults	97

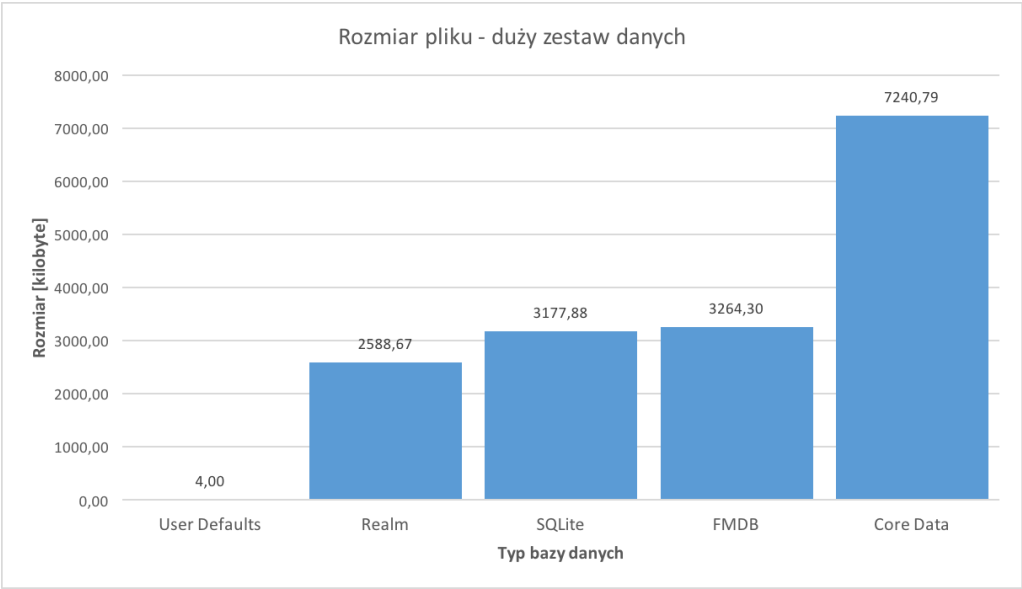
Tabela 13: Użycie pamięci operacyjnej podczas operacji - duży zestaw danych

Baza danych	Użycie pamięci RAM [mb]
SQLite	133,2
FMDB	134,9
Realm	141,6
User Defaults	145,7
Core Data	258,5

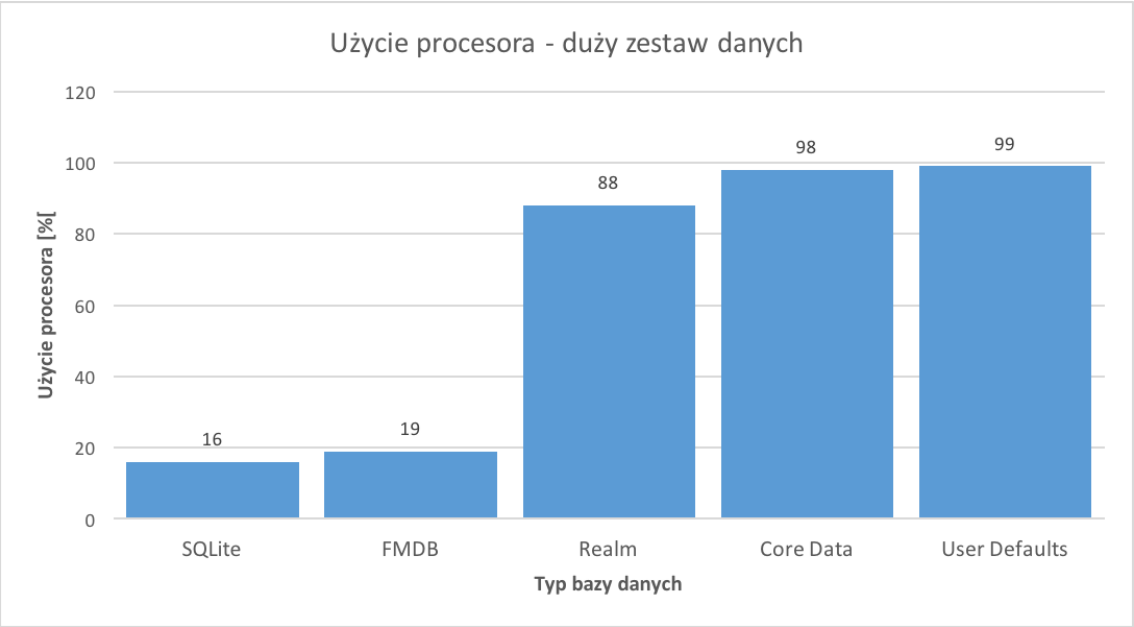
Prezentacja wyników w formie wykresów:



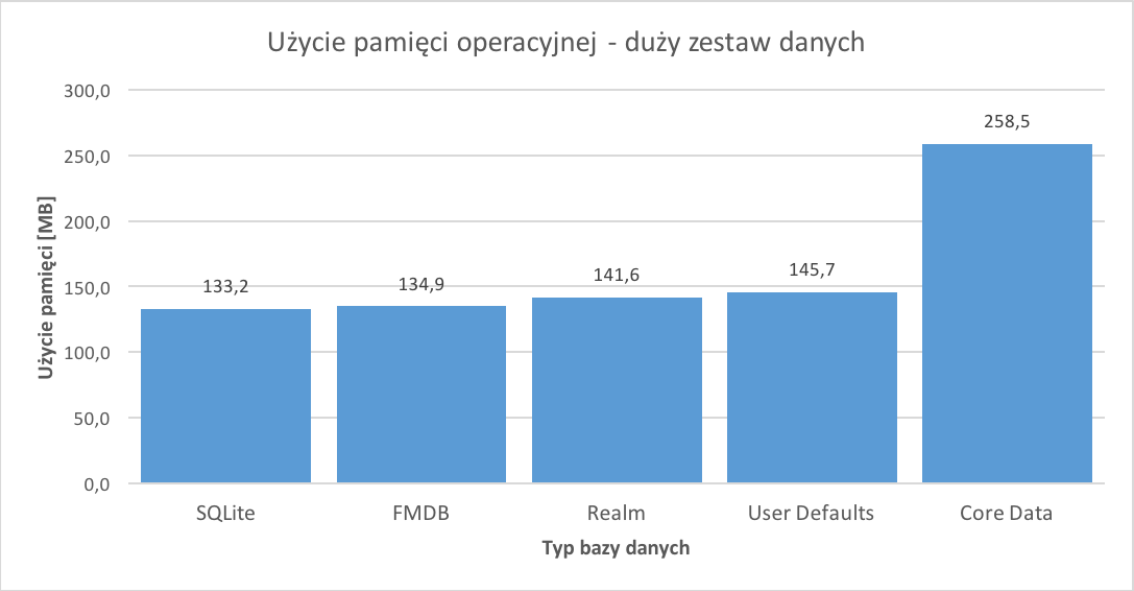
Rysunek 7.9: Czasy zapisu danych do bazy - duży zestaw danych



Rysunek 7.10: Rozmiar pliku wynikowego bazy danych - duży zestaw danych



Rysunek 7.11: Użycie procesora podczas operacji - duży zestaw danych



Rysunek 7.12: Użycie pamięci operacyjnej podczas operacji - duży zestaw danych

Czasy zapisu danych pokazane na wykresie 7.9 udowadniają, że wraz ze wzrostem ilości danych Domyślna Baza Użytkownika i Realm mają stały przyrost czasu względem dwóch poprzednich testów wynoszący około 85%. Sytuacja zmienia się w przypadku pozostałych baz danych. Core Data w teście z największym zestawem danych wypada o wiele gorzej niż w poprzednich testach. Czas zapisu danych sięga tu 25 sekund. Gorzej względem poprzednich testów wypada SQLite i FMDB osiągając czasy 34-35 sekund.

Rozmiary plików wynikowych w teście z największą ilością danych widoczny na wykresie 7.10 pokazują, że Domyślna Baza Użytkownika zwiększa rozmiar wykładniczo względem poprzednich testów. Plik bazy Realm osiąga rozmiar 2588 kb jest to 25% więcej niż w teście ze średnim zestawem danych. FMDB i SQLite zwiększyły swój rozmiar pliku o 10% zaś Core Data zwiększyła rozmiar pliku o 12

Użycie procesora podczas testu z wielkim zestawem danych pokazane na wykresie 7.11 pokazuje, że w dalszym ciągu SQLite i FMDB nie zwiększają zapotrzebowania na moc obliczeniową. Realm w tym teście potrzebował 10% wykorzystania procesora, zaś Domyślna Baza Użytkownika i Core Data pokazały największe zapotrzebowanie wynoszące 98-99%.

W przypadku użycia pamięci RAM widocznego na wykresie 7.12 można zauważyć, że SQLite i FMDB posiadają najmniejsze wartości wynoszące 133-135 MB. Realm zachowuje zbliżony rezultat wynoszący 141 MB. Zapotrzebowanie zwiększyła Domyślna Baza Użytkownika zajmując przedostatnie miejsce z wartością 145 MB. Core Data zaś potrzebowała aż 258 MB pamięci operacyjnej do operacji zapisu największego zestawu danych, jest to 150% więcej niż w poprzednim teście.

7.1.4 Podsumowanie - zapis danych

Z testów wynika, że najszybszy zapis danych możliwy jest przy wykorzystaniu Domyślnej Bazy Użytkownika. Realm posiada we wszystkich testach drugi rezultat. Core Data świetnie radzi sobie przy zapisie małej i średniej ilości danych, natomiast czas zapisu drastycznie rośnie w przypadku wielkiej liczby danych. SQLite i FMDB osiągają najgorsze rezultaty, wynikać to może z surowej implementacji obsługi konwersji danych podczas zapisu.

W wielkości pliku wynikowego także dominuje Domyślna Baza Użytkownika a zaraz po niej jest Realm. SQLite i FMDB posiadają pliki niemal tej samej wielkości. Core Data

ze względu na zapis do swojego pliku nie tylko samych danych a także danych przechowujących graficzną reprezentacje bazy oraz wszelkich konfiguracji wykorzystywanych przed środowisko programistyczne XCode osiąga wielkie rozmiary plików wynikowych.

Najmniejsze zużycie procesora wykazują bazy SQLite i FMDB. W przypadku Realm wykorzystanie mocy CPU zależne jest od ilości przetwarzanych danych i rośnie wraz z ich ilością. Core Data i Domyślna Baza Użytkownika wymagają najwięcej zasobów procesora do przeprowadzenia operacji zapisywania danych.

Wraz ze wzrostem ilości danych rośnie też zapotrzebowanie na pamięć operacyjna urządzenia. W przypadku małej ilości danych wszystkie bazy wykazują niewielkie zapotrzebowanie na pamięć RAM. Największe różnice widoczne są przy wielkiej ilości danych. Najlepiej w tej sytuacji radzą sobie bazy SQL, drugie miejsce zajmuje Realm. Core Data wymaga największej ilości pamięci operacyjnej.

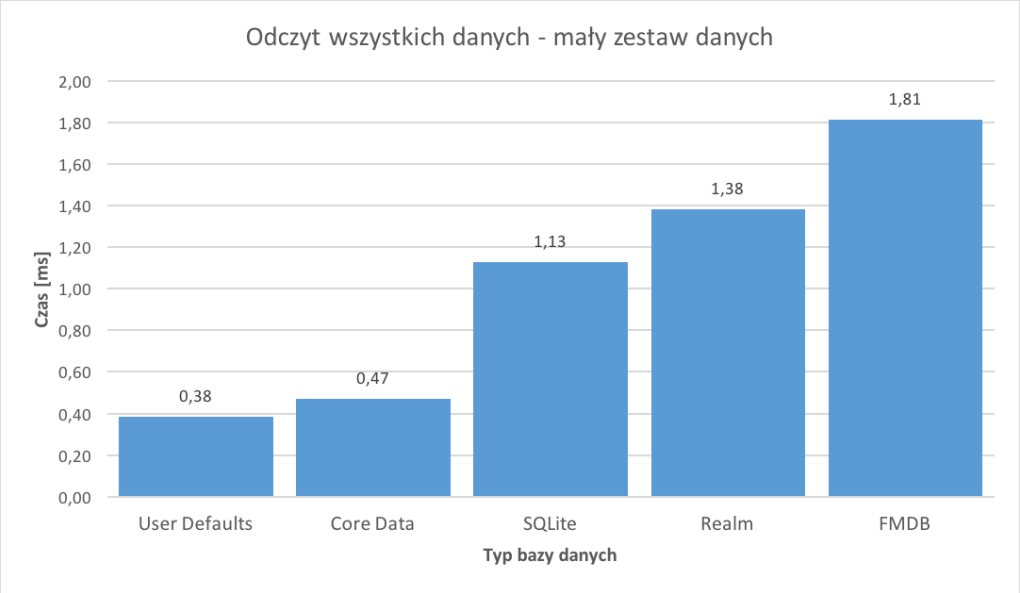
7.2 Testy odczytu danych

Podrozdział przedstawia testy odczytu danych w kliku różnych scenariuszach:

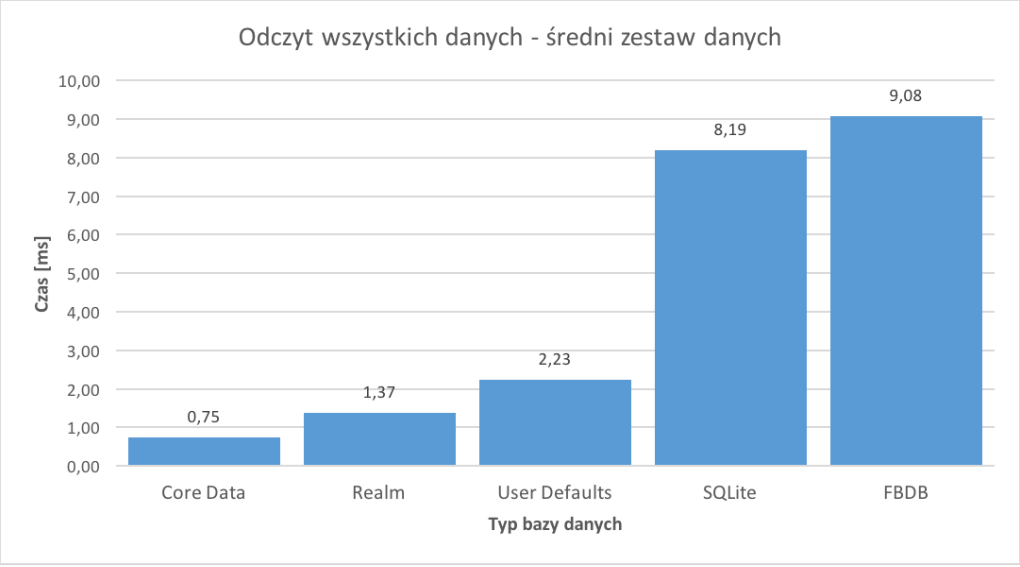
- Odczyt wszystkich danych z tabel.
- Wyszukanie wszystkich autorów o imieniu "Diena".
- Wyszukanie 2 książek z największa liczbą autorów.
- Odczyt maksymalnie 20 wydawnictw z największą liczba wydanych książek i posortowanie wyniku rosnąco.

Zastosowanie różnych operacji odczytywania danych ma na celu sprawdzenie jak testowane bazy danych radzą sobie podczas wykonywania zapytań. Wykonanie jedynie jednej operacji nie pozwoliło by uzyskać zadowalających rezultatów gdyż wiadomo iż każda z baz danych inaczej będzie wykonywać każdą z przeprowadzanych operacji. Tak samo jak w przypadku testowania zapisu danych każda z operacji zostały przeprowadzone na trzech różnych zestawach danych oraz każdy z testów wykonany został sto razy a prezentowany wynik jest średnią z wszystkich stu operacji.

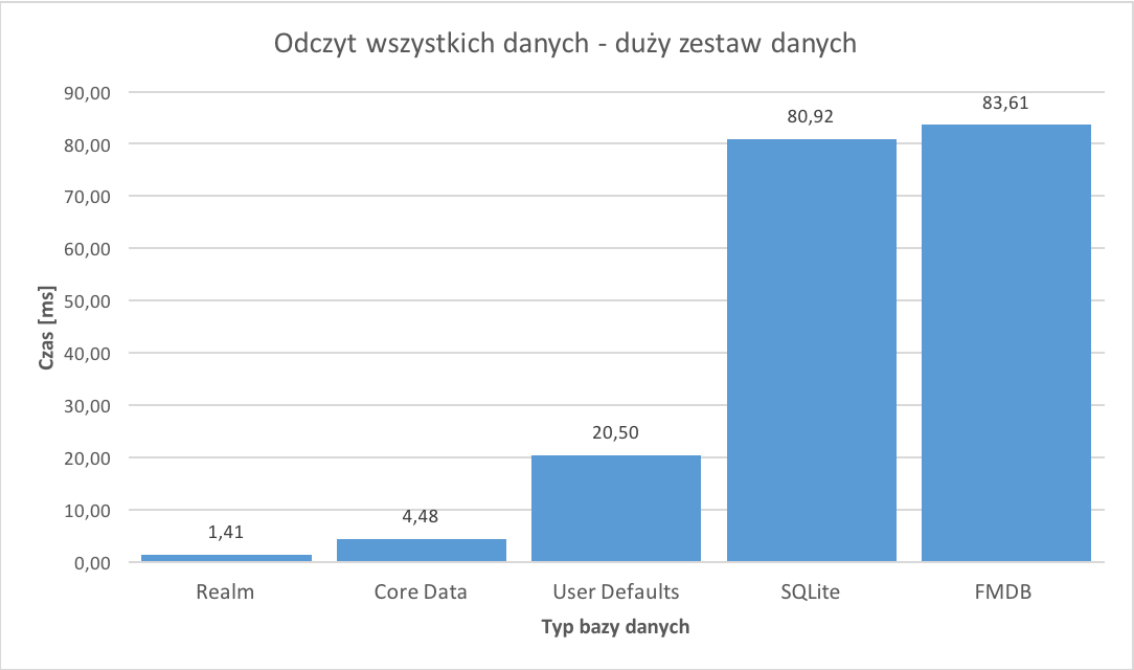
7.2.1 Odczyt wszystkich danych



Rysunek 7.13: Czas odczytu wszystkich danych - mały zestaw danych



Rysunek 7.14: Czas odczytu wszystkich danych - średni zestaw danych



Rysunek 7.15: Czas odczytu wszystkich danych - duży zestaw danych

Rezultaty odczytu wszystkich danych z baz prezentują różne wyniki zależne od ilości danych. W przypadku małego zestawu danych najmniejszy czas równy 0.38 ms uzyskała Domyślna Baza Użytkownika. Zaraz po niej z czasem 0.47 ms znajduje się Core Data. SQLite uzyskał czas 1.13 ms. Baza danych Realm odczytała wszystkie dane w czasie 1.38 ms. Najwolniejsza okazała się FMDB dając rezultat 1.81 ms.

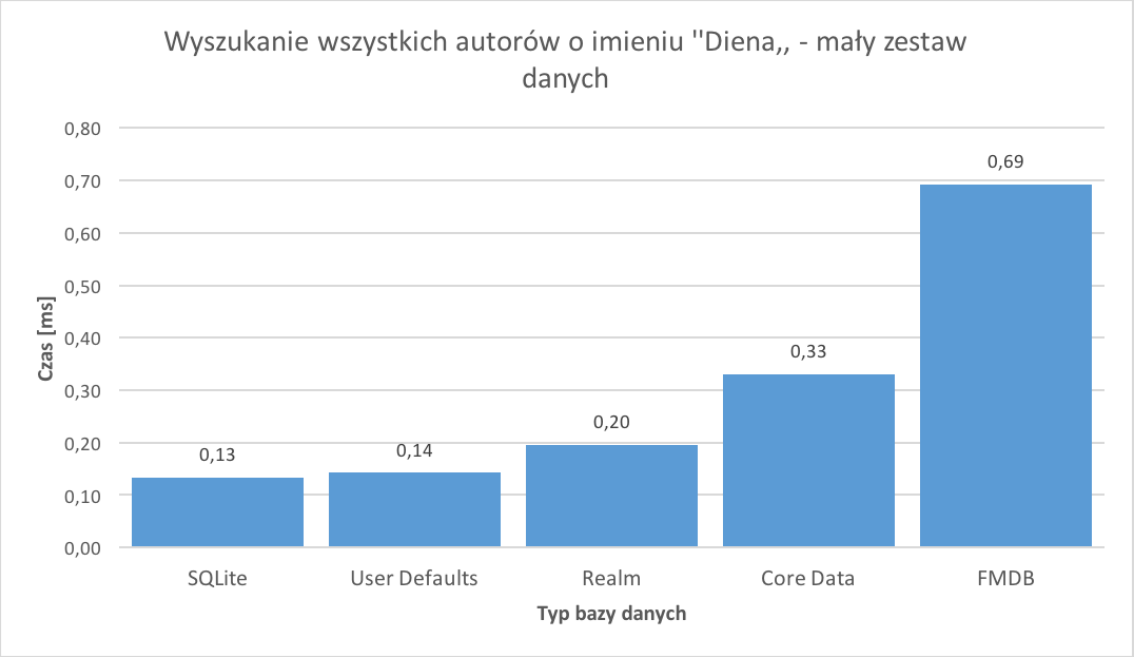
W przypadku odczytu średniego zestawu danych najszybsza okazała się Core Data uzyskując czas 0.75 ms. Realm zaś odczytał dane zestawu średniego w czasie 1.37 ms, czas ten różnie się zaledwie o 0.01 ms od wyniku uzyskanego przez tą bazę danych przy odczycie małego zestawu danych. Domyślna Baza Użytkownika uzyskała czas 2.23 ms co jest rezultatem gorszym aż o 1.85 względem testu z użyciem małego zestawu danych. Najwolniejsze okazały się SQLite z czasem 8,19 ms i FMDB z czase 9.08 ms.

Testy z użyciem dużego zestawu danych pokazują, że Realm jest najszybszą bazą danych jeżeli chodzi o odczyt wszystkich danych znajdujących się w bazie. Czas odczytu największej ilości danych to zaledwie 1.41 ms. Core Data uzyskała drugi rezultat z czasem 4.48 ms. Domyślna Baza Użytkownika odczytała dane w czasie 20.50 ms. SQLite i FMDB odczytały dane w najdłuższym czasie wynoszącym ponad 80 ms.

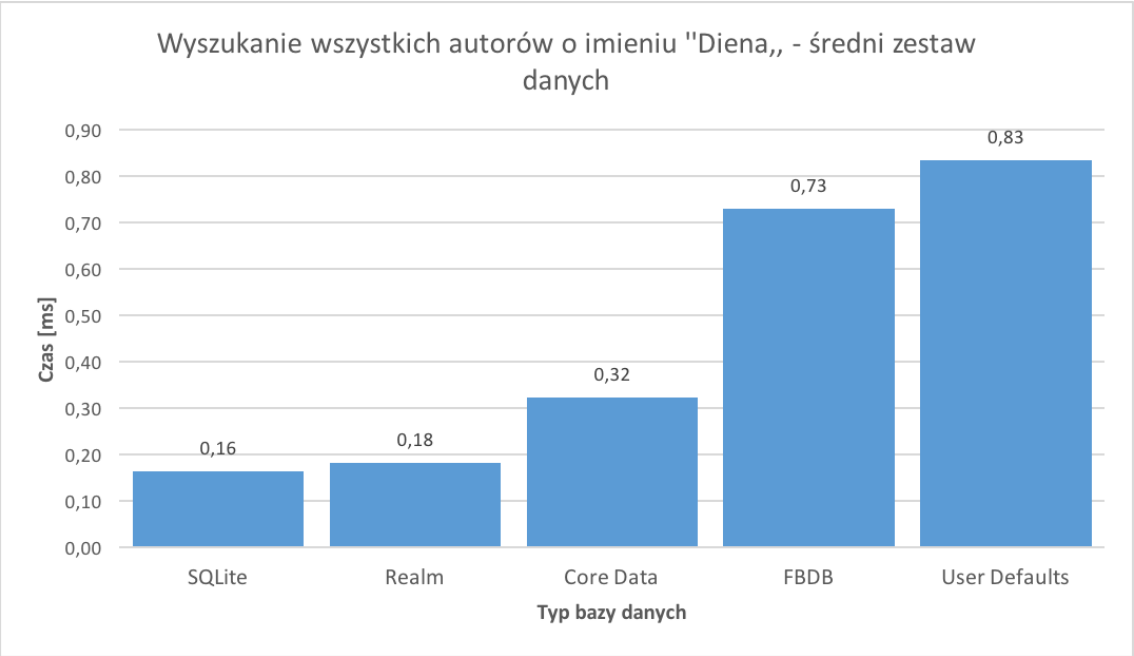
Można więc zaważyć, że Realm mimo słabych rezultatów w testach z małym i średnim zestawem danych uzyskał doskonały rezultat podczas największego testu. Różnice w cza-

sach odczytu w zależności od ilości danych przy użyciu Realm są bardzo niewielkie, wahają się w granicach 0.04 ms. Domyślna Baza Użytkownika znacząco zwiększa czas odczytu danych wraz ze wzrostem ilości zapisanych danych. Czas odczytu danych w przypadku użycia Core Data wzrasta wraz ze wzrostem ilości danych lecz przyrost czasu jest nie duży względem reszty testowanych baz danych. Najwolniejsze i z największym przyrostem czasowym okazały się bazy SQLite i FMDB.

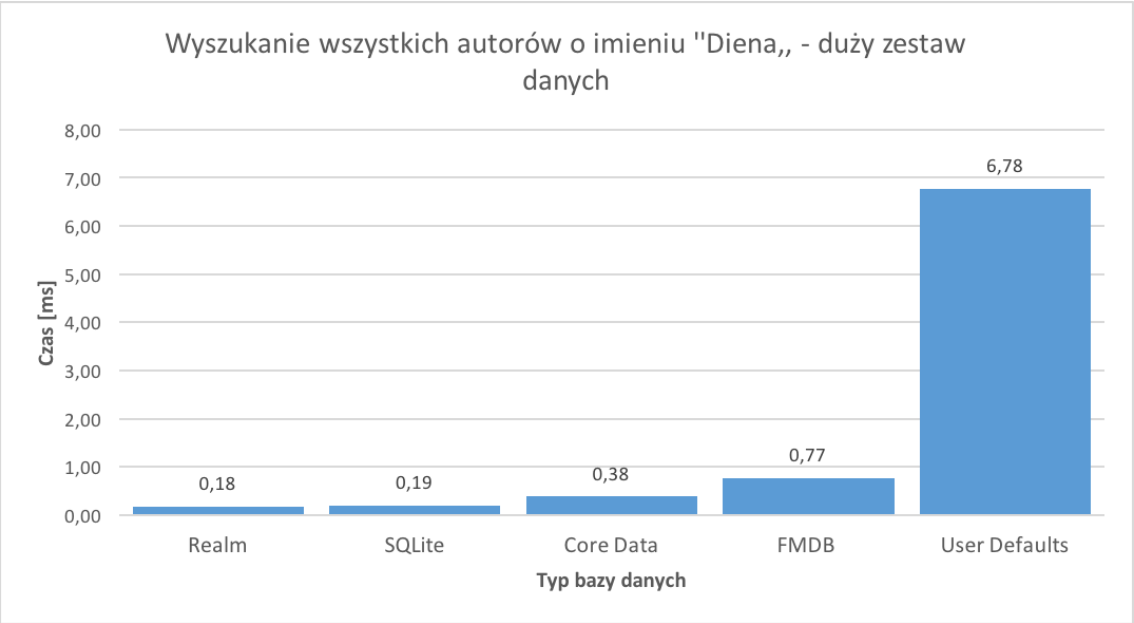
7.2.2 Wyszukanie wszystkich autorów o imieniu ”Diena,,



Rysunek 7.16: Wyszukanie wszystkich autorów o imieniu ”Diena,, - mały zestaw danych



Rysunek 7.17: Wyszukanie wszystkich autorów o imieniu ”Diena,, - średni zestaw danych



Rysunek 7.18: Wyszukanie wszystkich autorów o imieniu ”Diena,, - duży zestaw danych

Test odczytu autorów o imieniu ”Diena,, przedstawia następujące wyniki. W przypadku małego zestawu danych okazał się SQLite z wynikiem 0.13 ms, na drugim miejscu znalazła się Domyślna Baza Użytkownika uzyskując czas 0.14 ms. Rezultat Realm widoczny na wykresie 6.16 wynosi 0.20 ms, baza ta jest więc wolniejsza o 0.07 ms od najszybszej SQLite. Core Data wyszukała dane w czasie 0.33 ms. Dobrze widoczna jest różnica w cza-

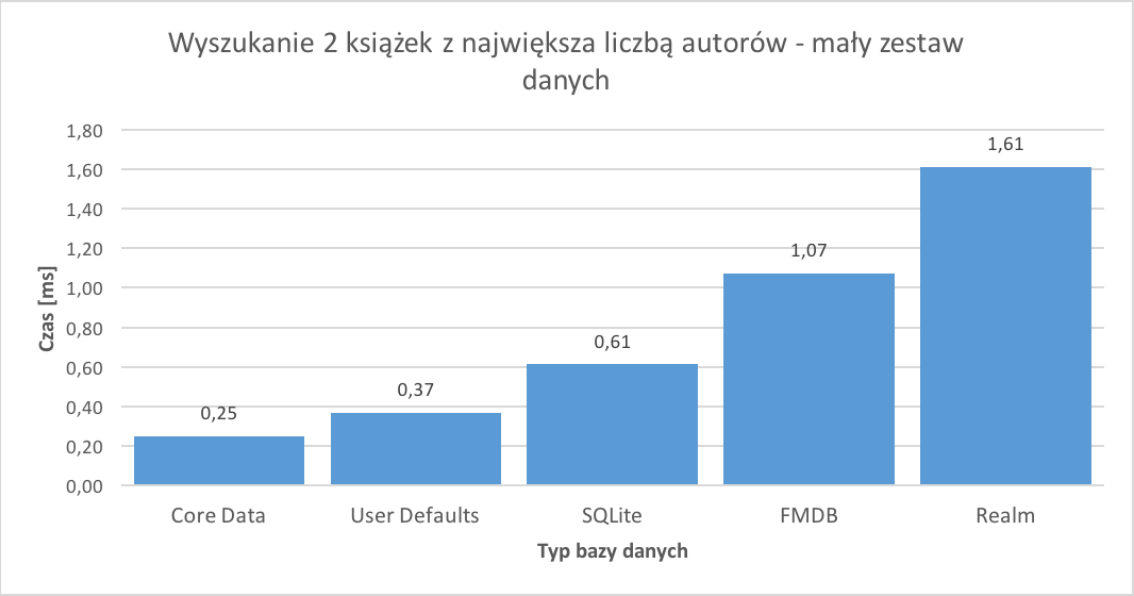
sie FMDB od czystej bazy SQLite. Wraper odczytał dane w czasie 0.69 ms, jest to rezultat gorszy o 0.56 ms od SQLite.

Odczyt w przypadku średniego zestawu danych pokazuję, że dziesięciokrotnie większa ilość rekordów względem poprzedniego testu nie wpływa znacząco na bazy Realm, SQLite, Core Data i FMDB. Rezultaty widoczne na wykresie 7.17 są większe o 0.02 - 0.04 ms. Znaczącą różnice pokazuję zaś Domyślna Baza Użytkownika, odczytała ona dane w czasie 0.83 ms co jest rezultatem o 85% gorszym niż w poprzednim teście.

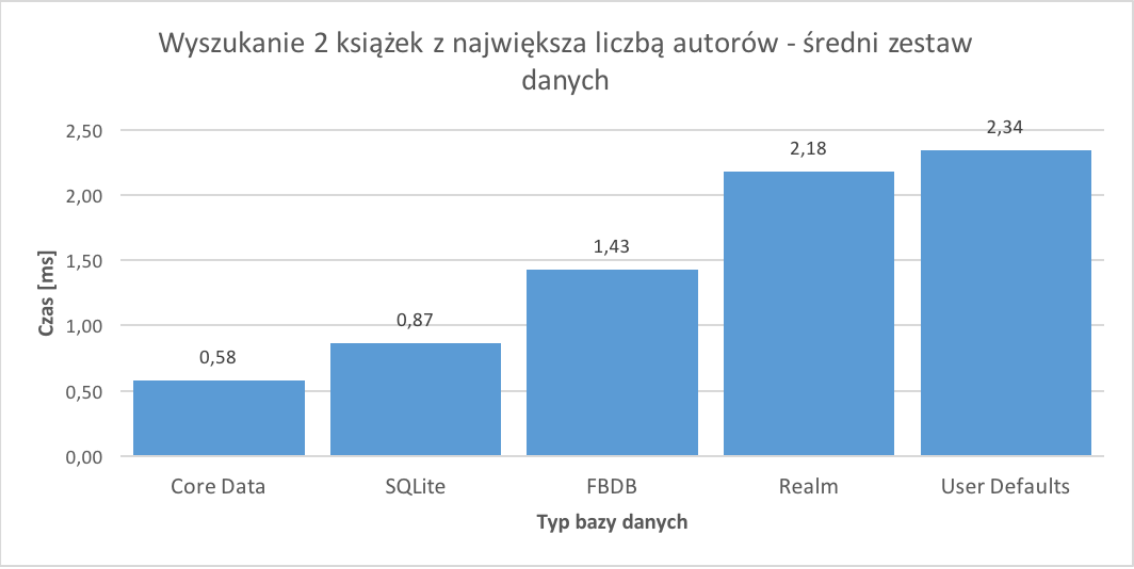
Także odczyt dużego zestawu danych nie wpłyną znacząco na uzyskane czasy otrzymania rezultatów. Na wykresie 7.18 widać, że tak jak poprzednio czasy dla baz Realm, SQLite, Core Data i FMDB są większe o 0.02 - 0.04 ms w porównaniu to poprzedniego testu. Domyślna Baza Użytkownika tak samo zwiększyła swój czas odczytu do 6.78 ms i jest to rezultat także o 85% większy niż w poprzednim teście.

Przedstawiona operacja odczytu autorów o wybranym imieniu nie wymagała przetwarzania relacji w bazach. Można tutaj zauważyć, które z baz danych dobrze radzą sobie z prostym przeszukiwaniem zapisanych danych. Realm i SQLite przetwarzając różne ilości danych potrzebują bardzo zbliżonego czasu na znalezienie rezultatu zapytania. Core Data pomimo, iż jest blisko dwukrotnie wolniejsza od Realm i SQLite także nie zwiększa znacząco czasu odczytu względem ilości danych. FMDB mimo iż jest wraperem SQLite okazuje się jednym z najwolniejszych rozwiązań. Domyślna Baza Użytkownika radzi sobie znakomicie z małą ilością danych. Wraz ze wzrostem liczby rekordów czas rośnie o 85% przy zwiększaniu ilości danych dziesięciokrotnie.

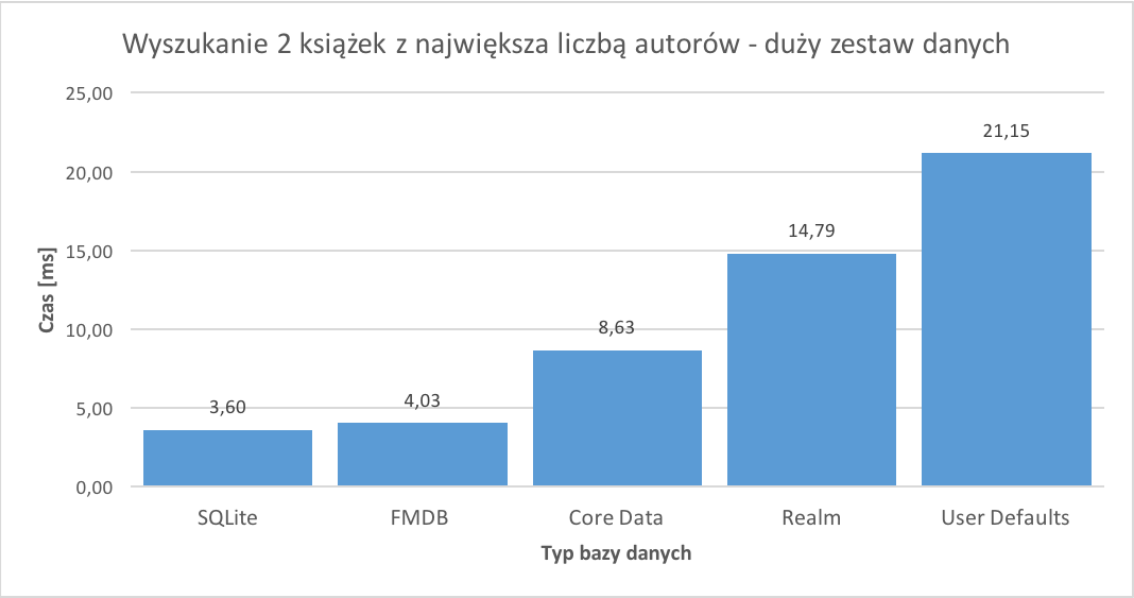
7.2.3 Wyszukanie 2 książek z największa liczbą autorów



Rysunek 7.19: Wyszukanie 2 książek z największa liczbą autorów - mały zestaw danych



Rysunek 7.20: Wyszukanie 2 książek z największa liczbą autorów - średni zestaw danych



Rysunek 7.21: Wyszukanie 2 książek z największa liczbą autorów - duży zestaw danych

Test wyszukania książek z największą liczbą autorów wymagał wspierania się relacjami pomiędzy tabelami książka i autor. Wyniki testów pokazują znaczące różnice w szybkości działania baz w porównaniu do testu przeszukiwania jednej tabeli.

Wykres 7.19 przedstawia rezultaty wyszukanie 2 książek z największa liczbą autorów w przypadku użycia małego zestawu danych. Najszybsza okazała się Core Data uzyskując czas równy 0.25 ms. Z małą ilością danych dobrze poradziła sobie też Domyślna Baza Użytkownika uzyskując rezultat 0.37 ms. Trzeci w kolejności jest SQLite z czasem 0.61 ms. Ponownie wolniejszy od SQLite okazał się FMDB, zwrócił on rezultat w czasie 1.07 ms. Najwolniejsza jest baza Realm, uzyskała ona czas 1.61 ms.

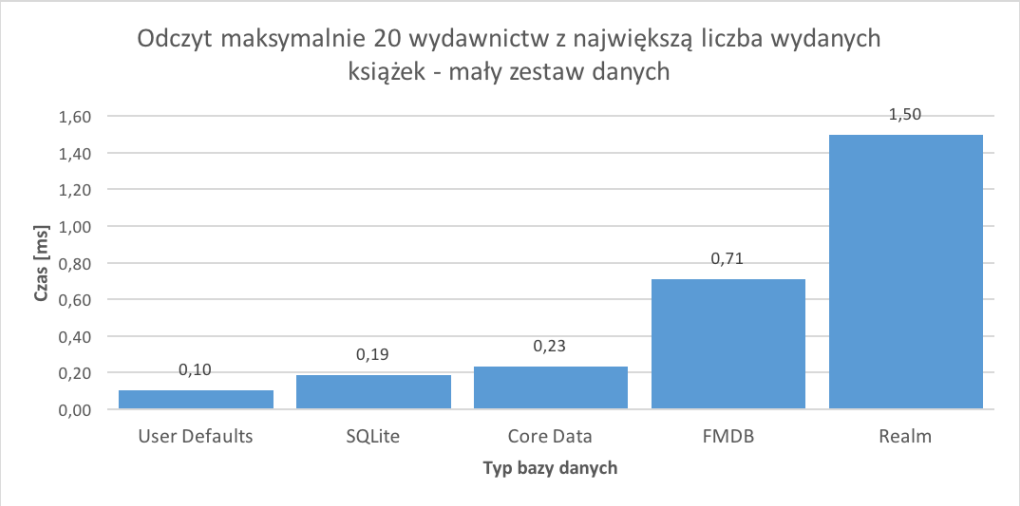
Dane przedstawione na wykresie 7.20 pokazują rezultatu testu z użyciem średniego zestawu danych. Core Data uzyskała najmniejszy czas 0.58 ms jest to rezultat o 55% większy od testu z małym zestawem danych. SQLite zakończył zadanie w czasie 0.87 ms, zaś FMDB ponownie okazał się wolniejszy od SQLite uzyskując czas 1.43 ms. Przedostatni rezultat uzyskał Realm, czas operacji wyniósł 2.18 ms. Domyślna Baza Użytkownika po raz kolejny podczas zwiększenia ilości danych uzyskuję znacznie gorsze wyniki. Baza zwróciła rezultat w czasie 2.34 ms i kolejny raz jest to rezultat większy o 85% od testu na małym zestawie danych.

Test z użyciem dużego zestawu danych, którego wyniki widoczne są na wykresie 7.21 pokazuje przewagę SQL-a w operacjach na dużej liczbie danych. Najszybsze okazują się

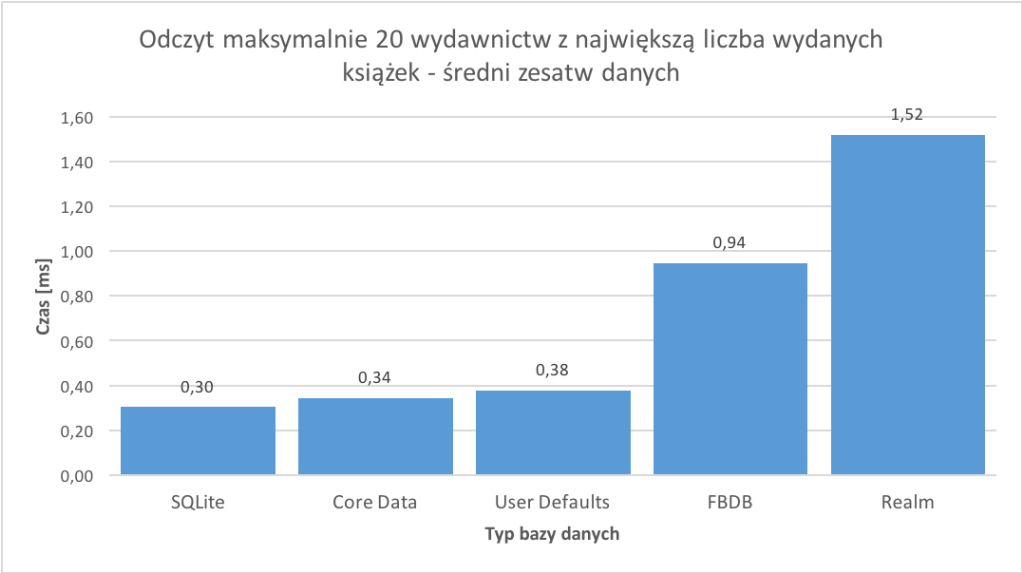
tu bazy SQLite i FMDB. SQLite uzyskał czas 3.60 ms zaś FMDB 4.03 ms. Trzeci rezultat uzyskała Core Data, potrzebowała ona 90% czasu więcej niż w teście poprzednim lecz w dalszym ciągu ukończyła zadanie z dobrym czasem wynoszącym 8.63 ms. Jedną z najwolniejszych baz okazał się Realm, zrealizował on zapytanie w czasie 14.79 ms. Najwolniejsza okazała się ponownie Domyślna Baza Użytkownika uzyskując czas 21.15 ms.

Powyższy test pokazuję przewagę rozwiązań SQL w bardziej skomplikowanych operacjach niż przeszukanie jednej tabeli. Rozwiązania takie jak SQLite i FMDB okazują się wolniejsze w przypadku małej i średniej liczby danych lecz zyskują wiele w przypadku dużej ilości danych. Core Data uzyskuję czasy coraz większe wraz z liczbą rekordów lecz nie jest to tak znacząca różnica jak w przypadku Realm czy Domyślnej Bazy Użytkownika, która okazała się najwolniejsza w przedstawionym teście niezależnie od zestawu danych.

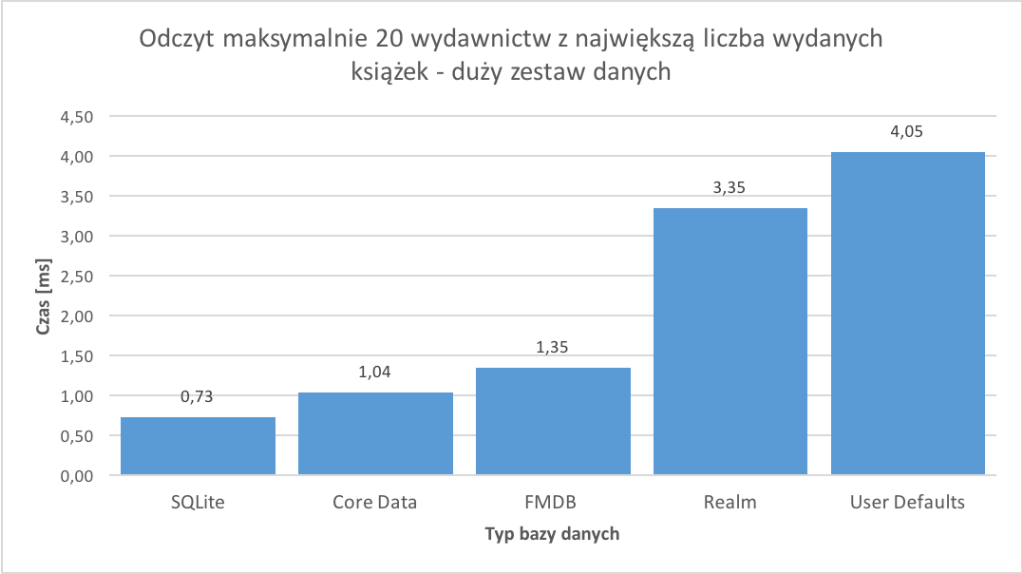
7.2.4 Odczyt maksymalnie 20 wydawnictw z największą liczba wydanych książek oraz posrotowanie wyniku rosnąco



Rysunek 7.22: Odczyt wydawnictw z największą liczba wydanych książek - mały zestaw



Rysunek 7.23: Odczyt wydawnictw z największą liczba wydanych książek - średni zestaw



Rysunek 7.24: Odczyt wydawnictw z największą liczba wydanych książek - duży zestaw

W przedstawionym teście zostało dodatkowo dodane sortowanie otrzymanego wyniku. Przy użyciu małego zestawu danych najszybszy czas otrzymania rezultatu uzyskała Domyślna Baza Użytkownika, skończyła wykonywać operacje w czasie 0.10 ms. Drugi w kolejności czas należy do SQLite i wynosi 0.19 ms. O 0.04 ms wolniejsza okazała się Core Data, wykonując operacje w czasie 0.23 ms. Jednym ze znacznie wolniejszych rozwiązań okazał się FMDB, uzyskał on czas 0.71 ms. Najwolniejszy w teście okazał się Realm wykonując zadanie w 1.50 ms.

Przy użyciu średniego zestawu danych najszybsza jest baza SQLite, wynik otrzymany zostaje w czasie 0.30 ms. Kolejny raz Core Data jest o 0.04 ms wolniejsza, rezultat zostaje zwrócony po 0.34 ms. Domyślna Baza Użytkownika uzyskała gorszy rezultat niż w przypadku małego zestawu danych, uzyskała czas 0.38 ms. Przedostatnie miejsce kolejny raz zajmuje FMDB, wykonując operacje w czasie 0.94 ms. Kolejny raz najwyższy czas należy do Realm wynosi on 1.51 ms i jest on jedynie o 0.01 ms większy niż w poprzednim teście.

Podczas testów z wykorzystaniem największego zestawu danych najlepszy rezultat ponownie należy do SQLite, uzyskał on czas 0.73 ms. Core Data kolejny raz jest za SQLite z czasem 1.04 ms. Lepszy wynik prezentuje FMDB, wykonując operacje odczytu w czasie 1.35 ms. Jednym z wolniejszych rozwiązań jest Realm, zakończenie odczytu nastąpiło w czasie 3.35 ms. Najwolniejsza jest Domyślna Baza Użytkownika kończąc operacje w czasie 4.05 ms.

Prezentowane wyniki pokazują, że w przypadku małej ilości danych w tego typu operacji dobrze spisuję się Domyślna Baza Użytkownika lecz traci ona swoją wydajność w przypadku dużych ilości danych. Przy większej ilości danych i dodatkowych operacjach takich jak sortowanie wyniku znacznie lepiej spisują się rozwiązania SQL (SQLite i FMDB) a także Core Data. Dokumentowa baza Realm okazała się najmniej wydajna w przedstawionym teście.

7.3 Testy usuwania danych

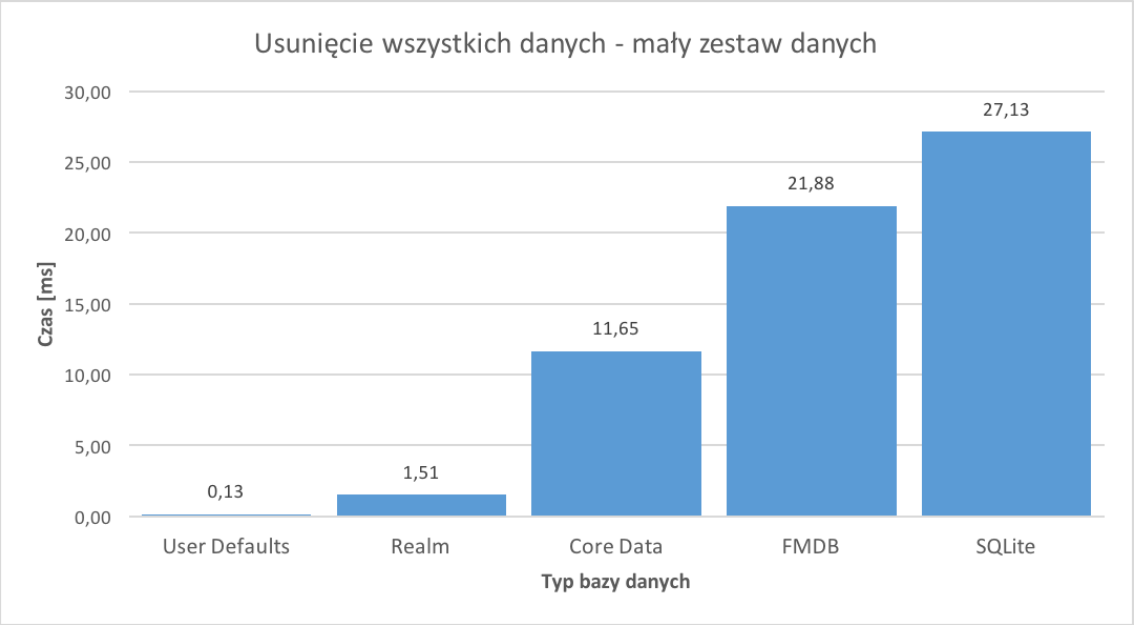
Podrozdział przedstawia testy usuwania danych w kilku różnych scenariuszach:

- Usunięcie wszystkich danych.
- Usunięcie wszystkich autorów którzy wydali 3 książki.
- Usunięcie wydawnictw które wydały książki o tytułach Ąnnie Oakleyłub "Tokyo Zombie (Tky zonbi)".

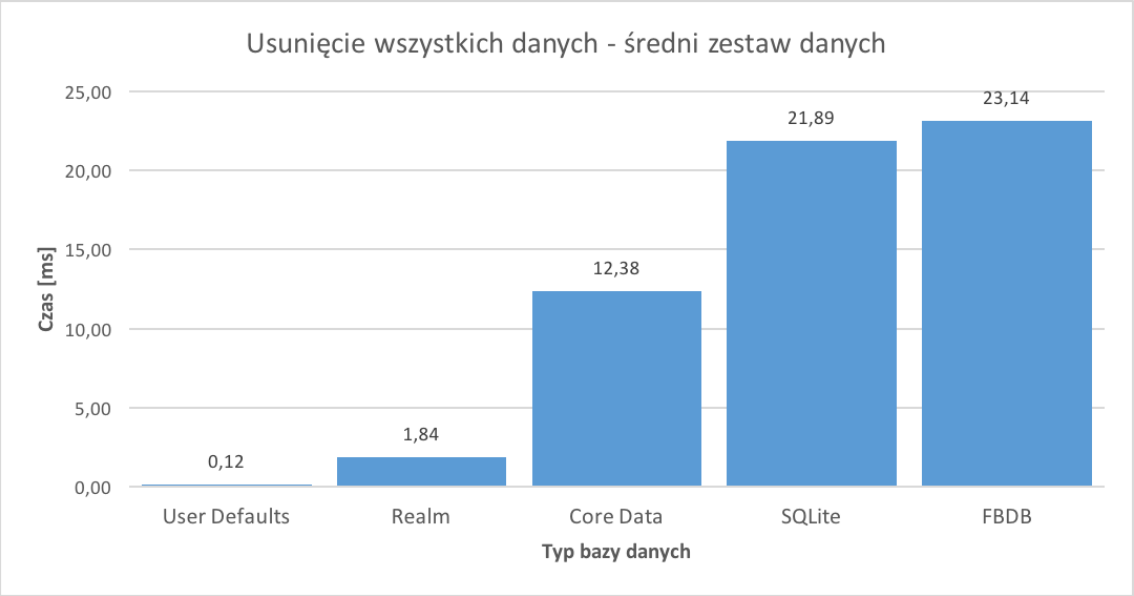
Testu usuwania danych przedstawiają różne stopnie skomplikowania operacji. Usunięcie wszystkich danych jest standardowym czyszczeniem bazy danych podczas którego czyszczone są wszystkie rekordy we wszystkich tablicach. Usunięcie wszystkich autorów którzy wydali 3 książki jest to operacja wymagająca zliczenia książek i wybraniu odpowiednich rekordów do usunięcia. Usunięcie wydawnictw które wydały książki o tytułach

Ąnnie Oakleyłub "Tokyo Zombie (Tky zonbi)"wymaga przeszukania relacji jeden do wielu wydawnictwo - książka, porównaniu nazw i usunięciu odpowiednich rekordów z tablicy wydawnictwo.

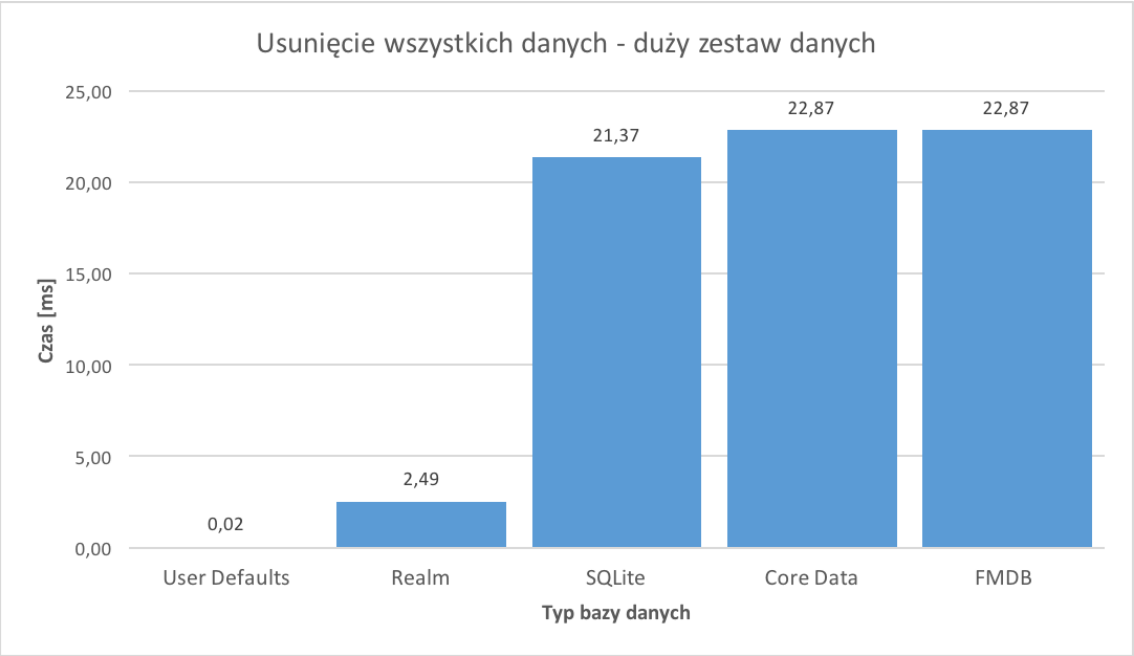
7.3.1 Usunięcie wszystkich danych



Rysunek 7.25: Usunięcie wszystkich danych - mały zestaw



Rysunek 7.26: Usunięcie wszystkich danych- średni zestaw



Rysunek 7.27: Usunięcie wszystkich danych - duży zestaw

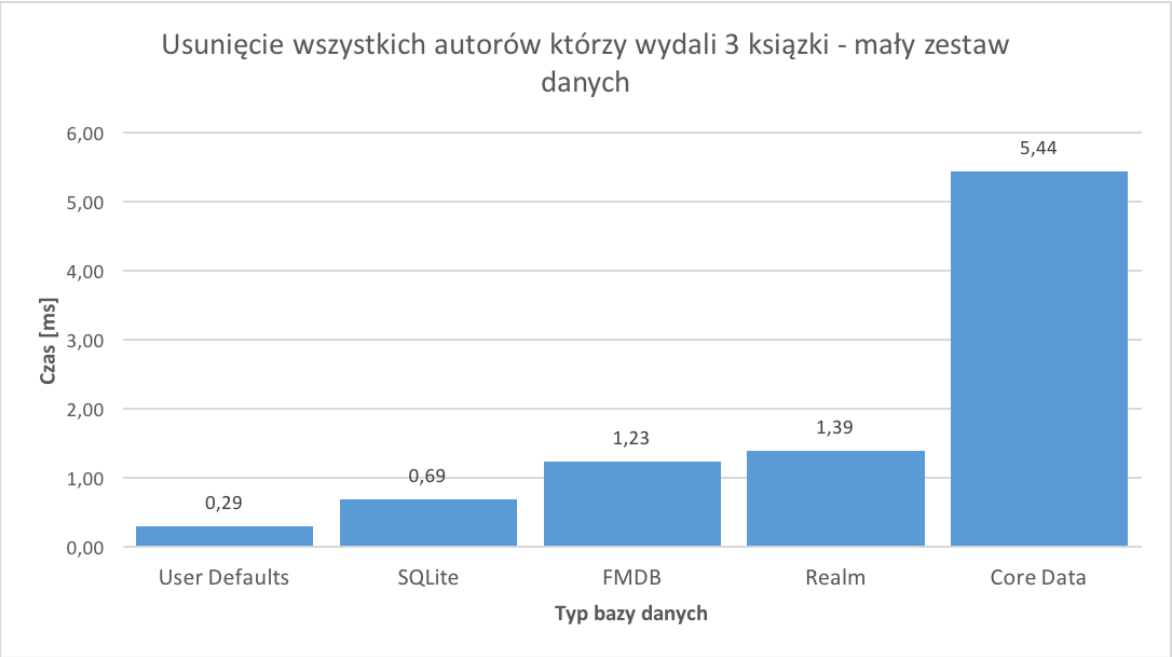
Usunięcie wszystkich danych przy małym zestawie danych pokazuje, że najszybsza jest Domyślna Baza Użytkownika operacja zakończyła się po 0.13 ms. Realm uzyskał drugi czas wynoszący 1.51 ms. Core Data uzyskała rezultat znacznie wyższy od poprzednich dwóch baz danych i usunęła dane w czasie 11.65 ms. Najgorzej wypadły bazy SQL, FMDB wyczyścił tablice w czasie 21.88 ms. Najwolniejszy był SQLite uzyskując czas 27.13 ms.

Test przeprowadzony na średnim zestawie danych kolejny raz pokazuję, że najszybsza jest Domyślna Baza Użytkownika uzyskała ona czas 0.12 ms. Drugi czas ponownie osiąga Realm i jest on równy 1.84 ms. Core Data także w tym teście znacząco wolniej usuwa dane, czas wyniósł 12.38 ms. SQLite wraz ze wzrostem ilości danych osiągnął rezultat lepszy niż w poprzednim teście i ukończył operacje w 21.89 ms. Najwolniejszy tym razem jest FMDB uzyskując czas 23.14 ms.

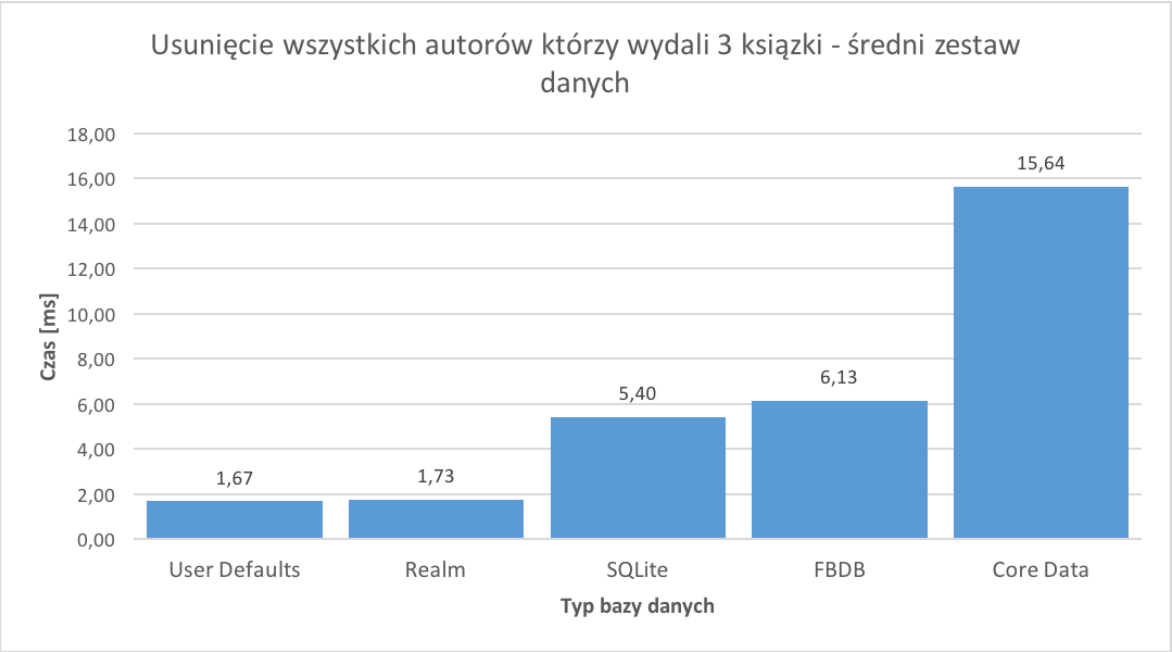
W przypadku dużego zestawu danych ponownie najlepszy rezultat należy do Domyślnej Bazy Użytkownika wynosi on 0.02 ms. Ponownie też Realm jest na drugim miejscu uzyskując czas 2.49 ms. SQLite w porównaniu do poprzedniego testu wykonał operacje szybciej pomimo większej ilości danych. Czas uzyskany przez SQLite wynosi 21.37 i jest o 0.51 mniejszy niż przy użyciu średniego zestawu danych. Różnica ta wynika z obecnego stanu urządzenia, można więc przyjąć, że operacja wykonana została w takim samym czasie. Core Data i FMDB uzyskały takie same czasy wynoszące 22.87 ms.

Test pokazuje, że niezależnie od zestawu danych najszybsza podczas usuwania wszystkich danych jest Domyślna Baza Użytkownika. Szybki jest też Realm jego czasy w stosunku do zwiększającej się ilości danych nie rosną znacząco. SQLite ponownie pokazuję, że doskonale radzi sobie podczas dużej ilości danych, zaś jego wydajność podczas niewielkiej ilości rekordów nie jest zadowalająca. Rezultaty Core Data są zależne od ilości danych i wydajność bazy spada wraz ze wzrostem ilości rekordów. Najwolniejszy w teście okazał się wraper FMDB, zaprezentował on najmniejszą wydajność podczas testów.

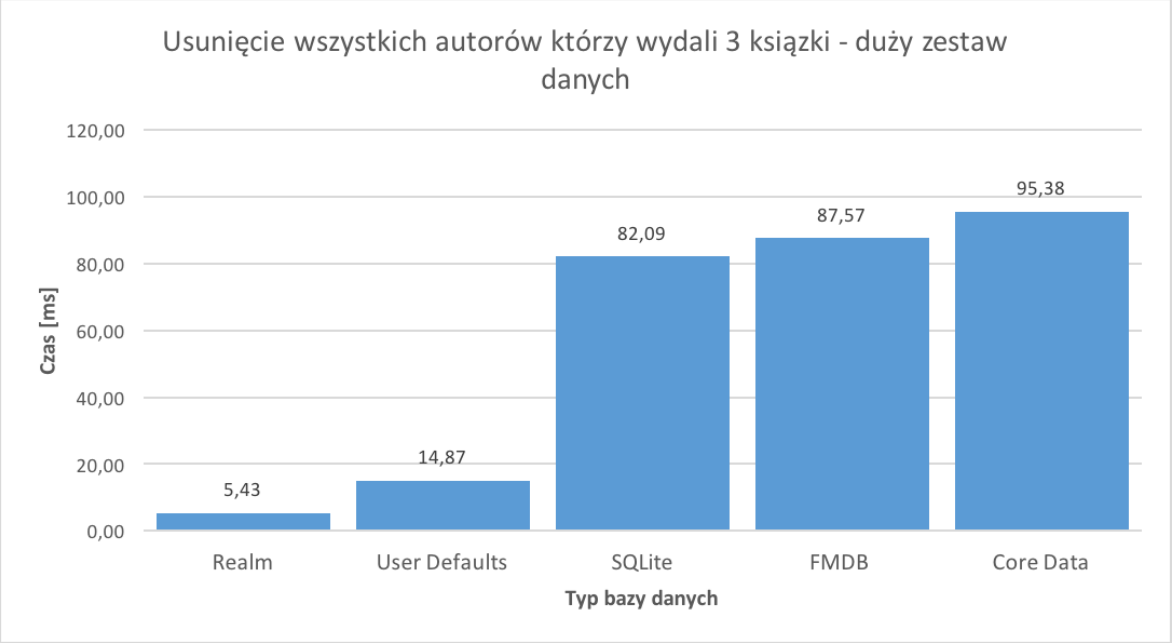
7.3.2 Usunięcie wszystkich autorów którzy wydali 3 książki



Rysunek 7.28: Usunięcie wszystkich autorów którzy wydali 3 książki - mały zestaw danych



Rysunek 7.29: Usunięcie wszystkich autorów którzy wydali 3 książki - średni zestaw danych



Rysunek 7.30: Usunięcie wszystkich autorów którzy wydali 3 książki - duży zestaw danych

Wyniki testów usunięcie wszystkich autorów którzy wydali 3 książki w przypadku użycia małego zestawu danych widoczne są na wykresie 7.28. Najniższy czas wynoszący 0.29 ms uzyskała Domyślna Baza Użytkownika. Drugi co do wielkości czas należy do SQLite, który uzyskał 0.69 ms. FMDB uzyskał blisko dwukrotnie wyższy czas wynoszący 1.23 ms. Realm zaś ukończył operacje z czasem gorszym o 0.16 ms od FMDB równym 1.39

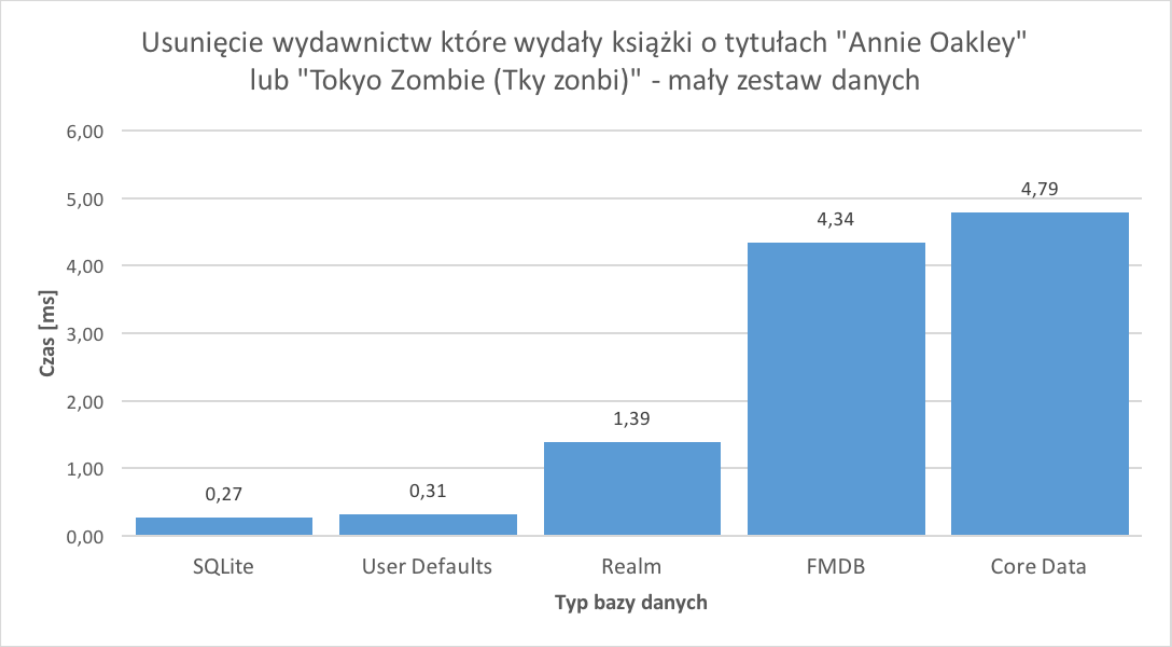
ms. Najwolniejsza okazała się Core Data, wykonała ona operacje po upływie 5.44 ms.

Wyniki testów z użyciem średniego zestawu danych widoczne są na wykresie 7.29. Najniższy czas równy 1.67 ms ponownie uzyskała Domyślna Baza Użytkownika. Wolniejszy o 0.06 ms okazał się Realm kończąc zadanie w czasie 1.73 ms. SQLite zakończył operacje w czasie 5.40 ms. FMDB zakończyło zadanie w czasie 6.13 ms. Najwolniejsza ponownie była Core Data uzyskując czas 15.64 ms.

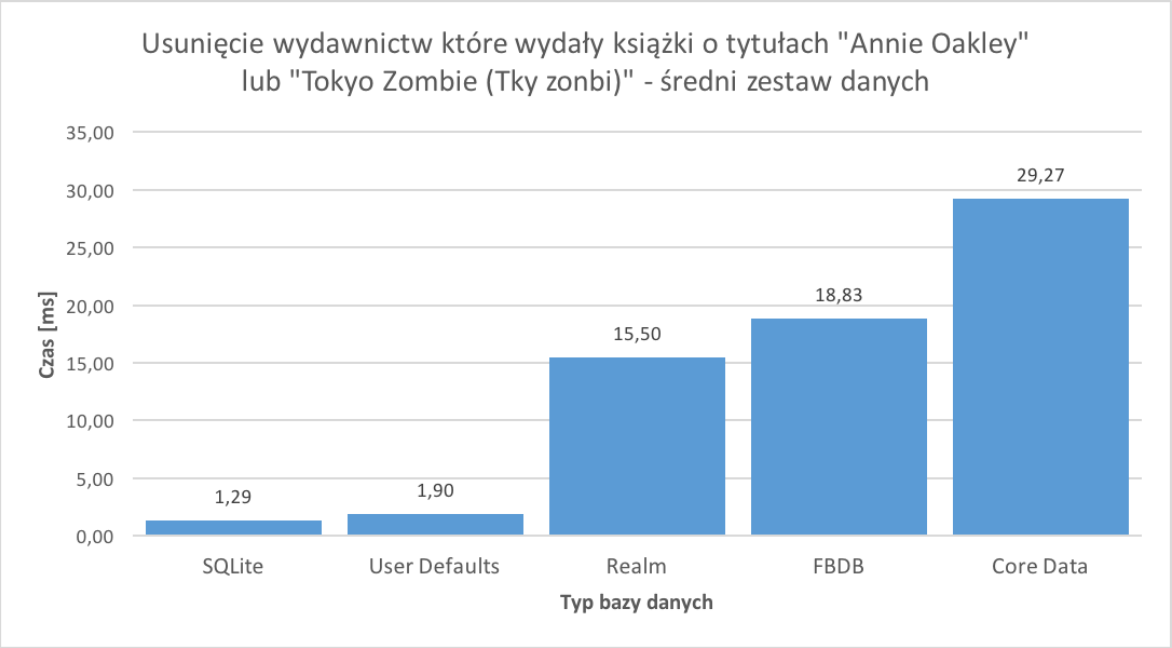
Rezultaty testów z wykorzystaniem dużego zestawu danych widoczne są na wykresie 7.30. Najszybszy okazał się Realm uzyskując czas 5.43 ms. Drugi co do wielkości rezultat wynoszący 14.87 ms należy do Domyślnej Bazy Użytkownika. SQlite ukończył wykonywanie zadanie w 82.09 ms. FMDB okazał się wolniejszy od SQLite o 5.48 ms i wykonał operacje w 87.57 ms. Ponownie najwolniejsza okazała się Core Data uzyskując czas 95.38 ms.

Przedstawiony test wymagał zliczenia wszystkich książek dla każdego z autorów a następnie wybraniu tych, którzy wydali trzy książki i usunięciu ich z bazy. Wyniki pokazują że przy małej i średniej ilości danych w tego typu operacjach dobrze spisuję się Domyślna Baza Użytkownika, która radzi sobie gorzej w przypadku dużej ilości danych. Realm w przypadku małej i średniej ilości danych wykonuję zadanie w podobnym czasie różnica wynosi jedynie 0.34 ms, zaś ta baza danych prezentuje doskonały rezultat w przypadku dużej ilości danych i jest najszybsza z testowanych baz. SQLite i FMDB nie pokazują zaskakującej wydajności w tego typu operacjach. FMDB przy użyciu różnych zestawów danych zawsze wypada gorzej od SQLite. Core Data okazała się najwolniejszym rozwiązaniem.

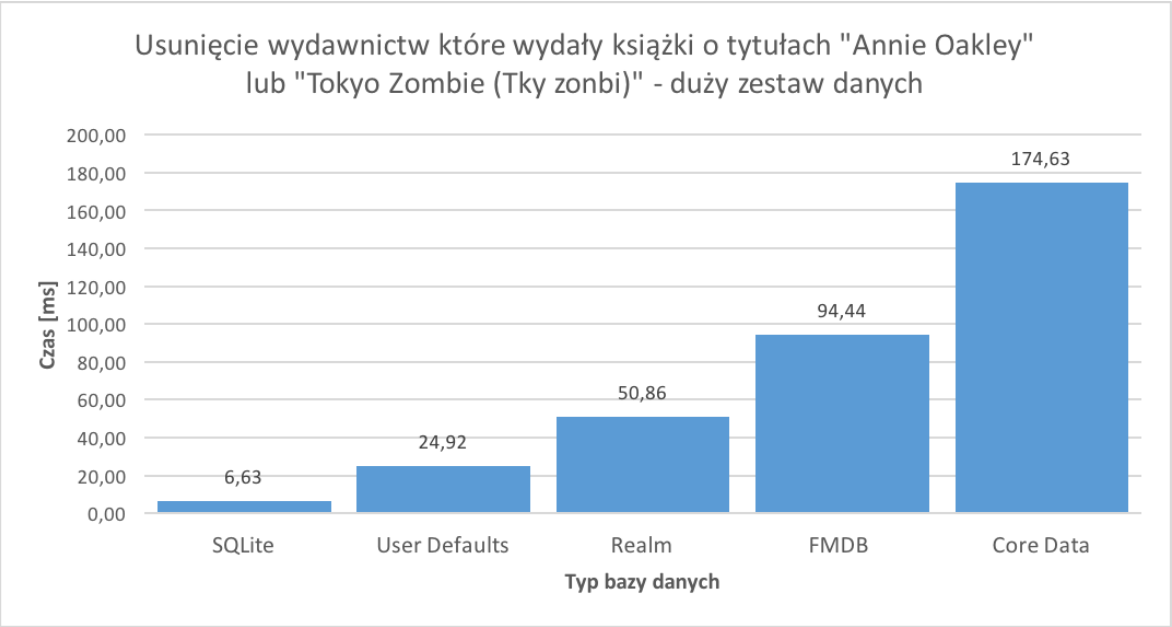
7.3.3 Usunięcie wydawnictw które wydały książki o tytułach „Annie Oakley” lub „Tokyo Zombie (Tky zonbi)”



Rysunek 7.31: Usunięcie wydawnictw które wydały książki o tytułach „Annie Oakley” lub „Tokyo Zombie (Tky zonbi)” - mały zestaw danych



Rysunek 7.32: Usunięcie wydawnictw które wydały książki o tytułach „Annie Oakley” lub „Tokyo Zombie (Tky zonbi)” - średni zestaw danych



Rysunek 7.33: Usunięcie wydawnictw które wydały książki o tytułach „Annie Oakley” lub „Tokyo Zombie (Tky zonbi)” - duży zestaw danych

Rezultaty testu usunięcia wydawnictw które wydały książki o tytułach „Annie Oakley” lub „Tokyo Zombie (Tky zonbi)” z użyciem małego zestawu danych widoczne na wykresie 7.31 pokazują, że najszybszy w tym przypadku okazał się SQLite. Uzyskał on czas równy 0.27 ms. Wolniej wykonała operacje Domyślna Baza Użytkownika kończąc zadanie w ciągu 0.31 ms. Trzeci rezultat uzyskał Realm wykonując operacje w 1.39 ms. FMDB ukończył zadanie w 4.34 ms, zaś Core Data okazała się najwolniejsza czas zakończenia operacji w jej wykonaniu wyniósł 4.79 ms.

W przypadku średniego zestawu danych ponownie najszybszy jest SQLite z czasem 1.29 ms. Ponownie nieznacznie wolniejsza w stosunku do SQLite okazała się Domyślna Baza Użytkownika wykonując operacje w 1.90 ms. Realm przedstawia znaczny spadek wydajności względem poprzedników kończąc zadanie po upływie 15.50 ms co jest o 13.9 ms wolniej niż Domyślna Baza Użytkownika. FMDB i Core Data ponowni okazały się najmniej wydajne. Czas wykonania testu dla FMDB wynosi 18.83 ms zaś dla Core Data równy jest on 29.27 ms.

Podczas wykorzystania dużego zestawu danych kolejność uzyskanych czasów przez bazy nie uległa zmianie. Ponownie najszybszy jest SQLite, czas wyniósł 6.63 ms. Znacznie wolniej operacje wykonała Domyślna Baza Użytkownika, zadanie wykonane zostało po upływie 24.92 ms. Realm zakończył operacje w 50.86 ms. FMDB zakończył usuwanie

wybranych wydawnictwa w czasie 94.44 ms. Core Data prezentuje znacznie wolniejsze działanie w tego typu zapytaniach, uzyskała największy czas równy 174.6 ms.

Test pokazuję, że w przypadku operacji wymagającej pracy z relacjami pomiędzy danymi najszybszym rozwiązaniem jest SQLite. Domyślna Baza Użytkownika ponownie prezentuje duży spadek wydajności wraz ze wzrostem ilości danych. Realm i FMDB prezentują zbliżone wyniki lecz kiedy operacje wykonywane są na dużej ilości danych zdecydowaną przewagę posiada Realm. Core Data niezależnie od ilości danych posiada najmniejszą wydajność z pośród testowanych baz danych niezależnie od ilości danych.

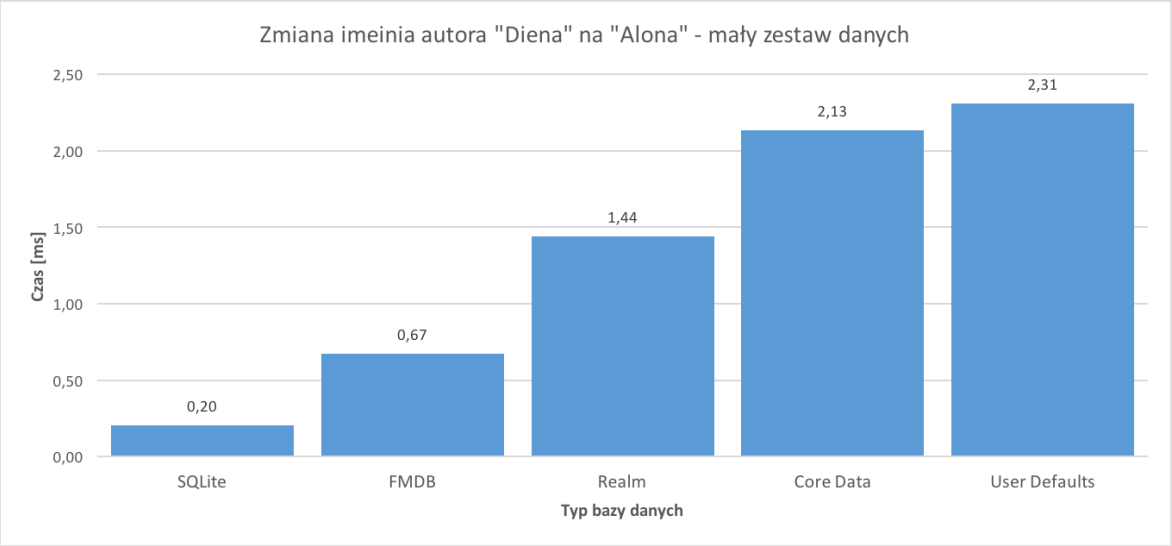
7.4 Testy edycji danych

Podrozdział przedstawia testy edycji danych w dwóch różnych opcjach:

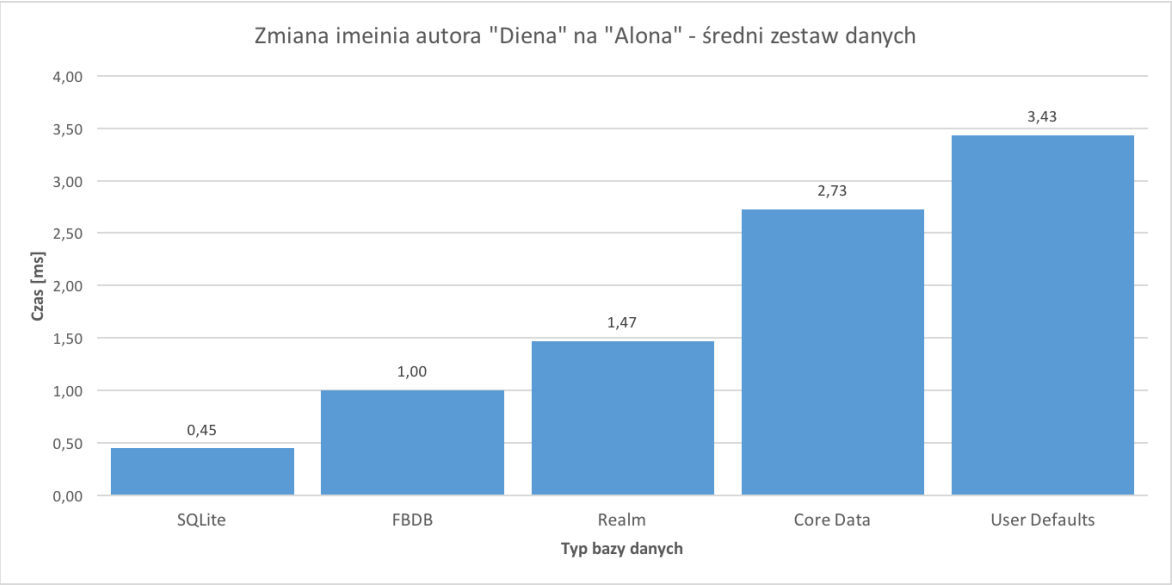
- Zmiana imienia autora „Diena” na „Alona”.
- Zmiana daty wydania książek na obecna datę.

Pierwszy z testów przedstawia operacje przeprowadzoną na wybranych rekordach z tabeli, wymaga on przeszukania bazy w celu odnalezienia odpowiedniego rekordu do edycji. Drugi test jest operacją przeprowadzaną na całej tablicy bez potrzeby wykonywania dodatkowych operacji wyszukiwania w obrębie danego rekordu.

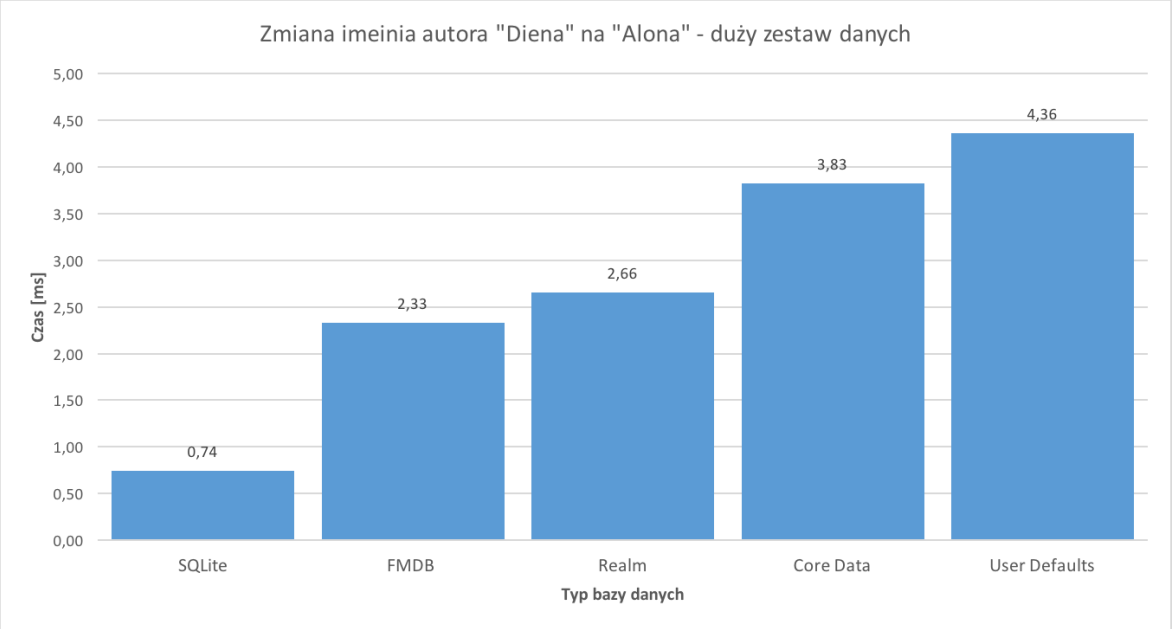
7.4.1 Zmiana imienia autora „Diena” na „Alona”



Rysunek 7.34: Zmiana imienia autora „Diena” na „Alona” - mały zestaw danych



Rysunek 7.35: Zmiana imienia autora „Diena” na „Alona” - średni zestaw danych



Rysunek 7.36: Zmiana imienia autora „Diena” na „Alona” - duży zestaw danych

Test zmiany imienia autora „Diena” na „Alona” z użyciem małego zestawu danych najszybciej wykonał SQLite uzyskując czas 0.20 ms. Wraper FMDB zakończył to zadanie w czasie 0.67 ms. Realm test zakończył po upływie 1.44 ms. Core Data pokazała niższa wydajność i skończyła zadanie w czasie 2.13 ms. Najwolniejsza okazała się Domyślna Baza Użytkownika kończąc operacje w czasie 2.31 ms.

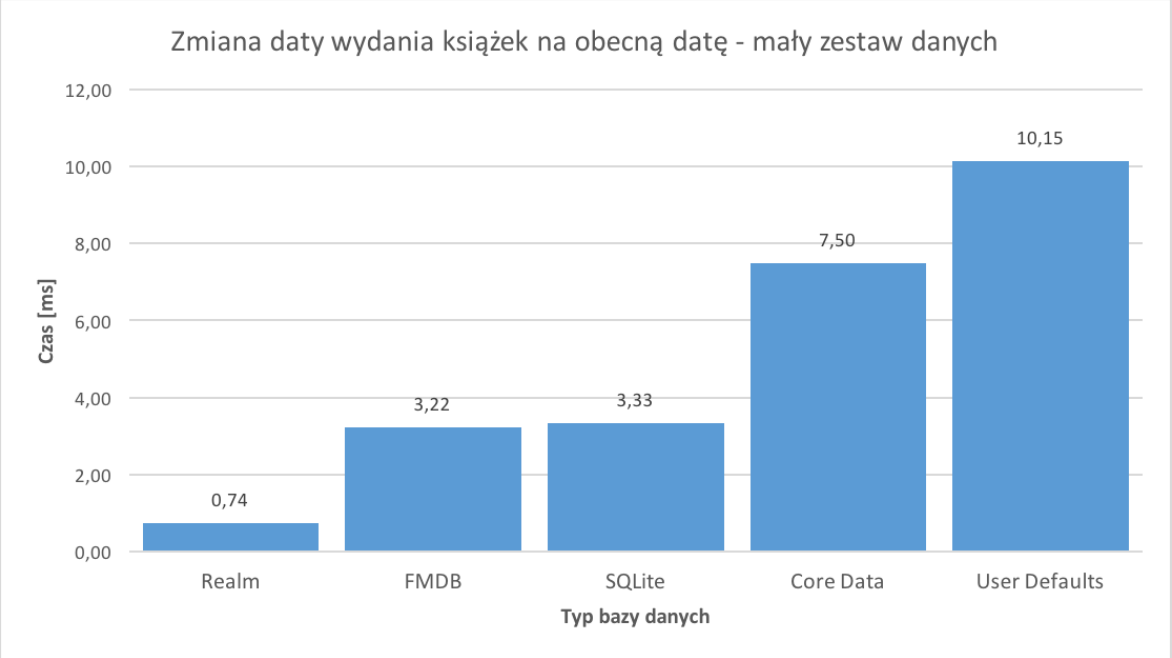
Zastosowanie średniego zestawu danych przyniosło następujące rezultaty. SQLite wy-

konał operacje w czasie 0.45 ms. FMDB okazał się wolniejszy o 0.55 ms i uzyskał czas 1 ms. Dla bazy Realm test zajął 1.47 ms. Core Data zakończyła zadanie w czasie 2.73 ms. Ponownie najmniej wydajna jest Domyślna Baza Użytkownika kończąc test w 3.43 ms.

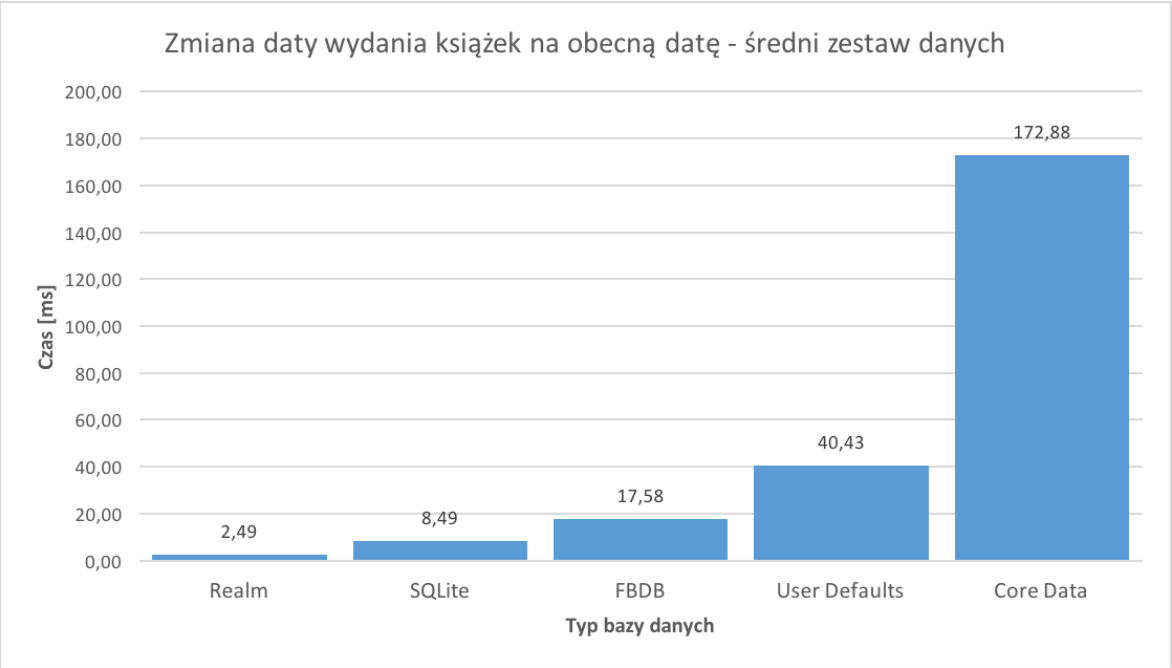
Użycie dużego zestawu danych pokazuje, że SQLite w dalszym ciągu jest bardzo wydajny, czas ukończenia operacji w tym przypadku wyniósł 0.74 ms. Znacznie wolniejszy jest FMDB, zakończył test w 2.33 ms. Realm przy zwiększonej ilości danych uzyskał czas 2.66 ms zaś Core Data potrzebowała 3.83 ms na edycje danych. Domyślna Baza Użytkownika po raz kolejny była najmniej wydajna potrzebowała aż 4.36 ms aby wykonać test.

Test pokazuję, że wybranie odpowiednich rekordów i ich edycja jest mocną stroną baz SQL, SQLite i FMDB pokazują największą wydajność w wykonanym teście niezależnie od ilości danych. Pozostałe bazy danych poddane testowi zwiększają swoje czasy wraz ze wzrostem ilości danych. Najmniej wydajną bazą w przypadku przedstawionego testu jest Domyślna Baza Użytkownika.

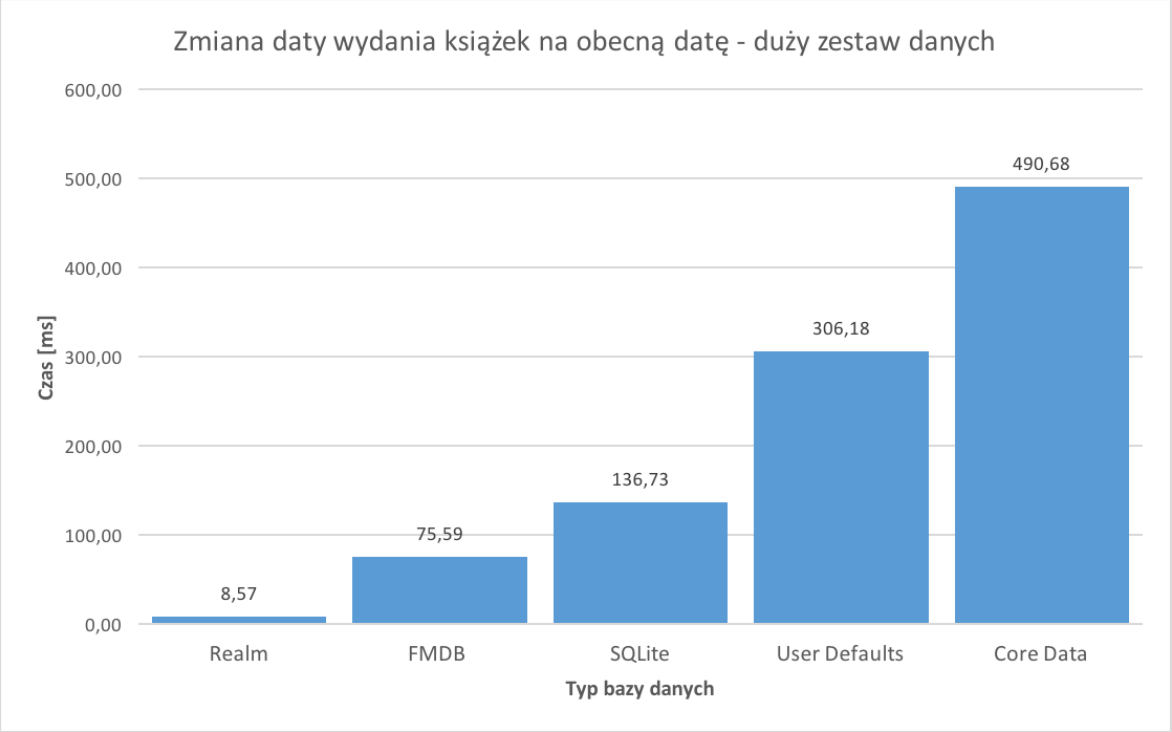
7.4.2 Zmiana daty wydania książek na obecna datę



Rysunek 7.37: Zmiana daty wydania książek na obecna datę - mały zestaw danych



Rysunek 7.38: Zmiana daty wydania książek na obecna datę - średni zestaw danych



Rysunek 7.39: Zmiana daty wydania książek na obecna datę - duży zestaw danych

Test zmiany daty wydania książek na obecna datę operował na wszystkich rekordach w tabeli Książka. Rezultaty w przypadku małego zestawu danych prezentują się następująco. Realm zakończył test w czasie 0.74 ms. FMDB i SQLite ukończyły operacje kolejno

w czasach 3.22 i 3.33 ms. Core Data okazała się znacznie wolniejsza od poprzedników i potrzebowała 7.50 ms aby zakończyć zadanie. Najwolniejsza jest Domyślna Baza Użytkownika uzyskując czas wykonania operacji równy 10.15 ms.

Średni zestaw danych w przedstawianym teście przedstawia bardzo wysoką wydajność bazy Realm, zakończył on operacje w 2.49 ms. SQLite na wykonanie testu potrzebował 8.49 ms. FMDB zakończył zadanie w czasie 17.58 ms. Znacznie wolniejsza od poprzednich baz jest Domyślna Baza Użytkownika, zakończyła test w 40.43 ms. Najwolniejsza jest Core Data potrzebowała aż 172.88 ms na uaktualnienie dat wydania książek.

Użycie dużego zestawu danych pokazało, że w dalszym ciągu Realm jest w podanym teście najszybszy, czas zakończenia operacji wyniósł 8.57 ms. Dużo wolniejszy jest FMDB potrzebował aż 75.59 ms. SQLite także potrzebował dużo czasu na zakończenie testu, operacja wymagała 136.73 ms. Domyślna Baza Użytkownika operacje zakończyła po upływie 306.18 ms. Core Data na wykonanie testu potrzebowała 490,68 ms.

Podczas edycji każdego z rekordów tabeli najwydajniejszą bazą okazał się Realm. Wolniejsze od dokumentowej bazy danych okazały się bazy SQL, edycja danych zajęła im więcej czasu. Najmniej wydajna jest Domyślna Baza Użytkownika oraz Core Data, której wydajność jest bardzo słaba w przedstawionej operacji.

8 Podsumowanie

Celem pracy było porównanie wydajności oraz sposobów implementacji baz danych w systemie iOS. Wykonane testy, przedstawione w pracy pozwoliły porównać wydajność opisanych baz danych. W pracy przedstawione i opisane zostały też fragmenty kodu dla każdej bazy danych potrzebne do wykonywania tych samych operacji. Przytoczone fragmenty kodu obrazują też różnice w skomplikowaniu i ilości potrzebnego kody do przeprowadzania tych samych zadań.

Stworzono aplikacje na system iOS, która pozwala na wybranie bazy danych do testu, wykonanie odpowiednich testów, określenie ilości powtórzeń danego testu a także wysłanie wyniku w formacie CSV na adres email.

Wykonano 10 różnych testów. Każdy z testów został przeprowadzony na trzech zestawach danych: małym - 10-elementowym, średnim - 100-elementowym i dużym - 1000-elementowym. Dodatkowo każdy z testów dla uzyskania wiarygodnych wyników został wykonany sto razy. W pracy testom zostały podane różne typy baz danych dlatego wynikiem końcowym każdego z testów był moment otrzymania identycznych obiektów, które zostały uprzednio poddane zapisowi w bazie.

Podczas zapisu danych dominuje Domyślna Baza Użytkownika lecz należy pamiętać o jej ograniczeniach takich jak brak relacji czy brak możliwości zapisu zdjęć. Baza Realm w testach zapisu zajmuje drugie miejsce pod względem uzyskanych czasów, czas zapisu nie rośnie znacząco podczas zwiększania ilości zapisywanych danych. Core Data znacząco Zwiększa czas wraz ze wzrostem ilości danych. Najwolniejsze podczas zapisu danych okazały się bazy SQLite i FMDB co pozwala stwierdzić, że zapis danych nie jest ich mocną stroną.

Testy odczytu wszystkich danych pokazują, że zależnie od ilości danych testowane bazy osiągają różne czasy wykonania operacji. Realm pomimo przedostatniego rezultatu przy małej ilości rekordów zwiększa czas odczytu nieznacznie wraz z ilością danych. Domyślna Baza Użytkownika radzi sobie dobrze jedynie z małą ilością danych. Core Data osiąga najmniejszy czas podczas pracy ze średnim zestawem danych. Bazy SQLite i FMDB osiągają najgorsze rezultaty ze wszystkich testowanych baz danych.

Testy odczytu wybranych danych, które wymagały przeszukania zbioru jednoznacznie pokazują, że w takich operacjach najlepiej spisuję się baza SQLite. FMDB wykorzystująca

SQLite posiada większe czasy wykonywania operacji. Rezultaty Domyślnej Bazy Użytkownika zależne są od stopnia skomplikowania operacji, im bardziej złożona operacja tym czas wykonania zadania jest większy. Core Data i Realm w tego typu testach wypada różnie, wpływ na wyniki ma ilość danych i stopień trudności operacji.

W testach usuwania wszystkich danych dominuje Domyślna Baza Użytkownika i Realm. Core Data posiada trzeci co do wielkości czas usunięcia małej i średniej ilości danych. Całkowite czyszczenie bazy najwięcej trwa w przypadku SQLite i FMDB. Testy wymagające przeszukania bazy i usunięciu jedynie wybranych elementów pokazują, że im bardziej skomplikowana jest operacja tym lepsze są bazy SQLite i FMDB. W przeciwnym wypadku Realm i Domyślna Baza Użytkownika uzyskują lepsze rezultaty. Najwolniej tego typu operacje przeprowadzane są przez Core Data.

Edycje danych najwolniej wykonuje Domyślna Baza Użytkownika i Core Data. Doskonale zaś radzą sobie bazy SQL a także Realm, który najlepiej spisuję się podczas edycji całych tabel.

Przytoczone fragmenty kodu obrazują sposoby i różnice implementacji użytych w pracy baz danych. Najmniejszy nakład pracy programisty wymagany jest podczas używania bazy Realm. Dostarcza ona doskonały interfejs dzięki, któremu nawet skomplikowane operacje programista jest w stanie zaimplementować za pomocą jedynie kilku linii kodu. Core Data wymaga doskonałej znajomości tego narzędzia, nakład kodu jest większy niż w przypadku Realm. SQLite posiada nieczytelny interfejs a także wymaga bardzo dobrej znajomości baz SQL. Operację są trudne w zaimplementowaniu. Opisane problemy bazy SQLite eliminuje biblioteka opakowująca FMDB niestety wnosi ona opóźnienia przez, które wypada ona gorzej pod względem wydajności. Domyślna Baza Użytkownika wymaga jedynie znajomości języka programowania lecz implementacja skomplikowanych operacji jest bardzo skomplikowana i wymaga dużej ilości kodu.

Wskazanie najbardziej wydajnej bazy danych jest bardzo trudne. Uzależnione jest to od wielu czynników między innymi: ilości przechowywanych danych, rodzaju najczęściej wykonywanych operacji przez bazę czy też typu przechowywanych danych. Wydajność każdej z bazy może też ulec zmianie poprzez wykorzystanie różnego typu dostępnych bibliotek do operacji na zmiennych. Tego typu biblioteki mogą znacząco poprawić wydajność baz SQL poprzez szybszą konwersje typów zmiennych przechowywanych w obiektach programu pomiędzy typami wymaganymi do zapisu w bazie. Jednoznacznie można zaś stwier-

dzić iż najprostszy interfejs i sposób implementacji posiada biblioteka Realm.

Kody źródłowe

Spis rysunków

Spis tabel