ORIGINAL SCIENTIFIC PAPER
DOI: 10.63395/STEDConf14022025937S201
Submitted: 22.08.2025.
Accepted: 22.09.2025.

PERFORMANCE COMPARISON OF VARIOUS DATABASE TYPES IN REACT NATIVE APPLICATIONS

Dijana Stojić, Dejan Vujičić, Đorđe Damnjanović, Marina Milošević

University of Kragujevac, Faculty of Technical Sciences Čačak, Svetog Save 65, 32102 Čačak, Serbia, dijana.stojic@ftn.kg.ac.rs, dejan.vujicic@ftn.kg.ac.rs, djordje.damnjanovic@ftn.kg.ac.rs, marina.milosevic@ftn.kg.ac.rs

Coresponding author: Dijana Stojić, University of Kragujevac, Faculty of Technical Sciences Čačak, Svetog Save 65, 32102 Čačak, Serbia, dijana.stojic@ftn.kg.ac.rs

ABSTRACT

In the modern digital age, mobile devices have become an indispensable part of everyday life. With the growing volume of data we handle daily, the speed of data loading and access becomes a crucial factor for the efficient operation of mobile applications. Large datasets are becoming increasingly common, such as, for example, the German collocation dictionary. Given that mobile devices use different operating systems, such as Android and iOS, there is an increasing need for tools that enable the development of cross-platform applications from a single codebase. React Native, as a popular open-source framework, provides exactly this solution. This paper tested the loading of data from the dictionary using React Native applications. This data loading was performed in four ways: using AsyncStorage, SQLite, TinyBase, and Realm databases. A different number of rows was loaded, and the loading speed was compared.

Keywords: Android, database, iOS, performance testing, React Native.

INTRODUCTION

In today's digital age, mobile devices have become an essential part of everyday life. With the growing volume of data handled by mobile applications, performance – especially data loading speed and access time – has become a crucial factor. Traditionally, developing mobile applications required writing separate codebases for Android (using Java or Kotlin) and iOS (using Swift or Objective-C), which significantly increased the time, cost, and complexity of development and maintenance (Alsaid et al., 2021). This is overcome by using React Native, a widely used open-source framework that enables the development of cross-platform applications from a single codebase, thereby increasing development efficiency.

This study investigates how different types of databases affect performance within the React Native environment, with a particular focus on working with large datasets such as a German collocation dictionary. The experimental part of the work utilizes four different data management technologies: AsyncStorage, SQLite, TinyBase, and Realm. As part of the comparison, the performance of these solutions was analyzed depending on the size of the dataset, specifically the number of rows, with particular focus on measuring the time required to load and display data within the application.

The aim of the study is to determine which technology offers the best performance in the context of React Native applications when working with local data and to provide guidelines for choosing a database depending on the specific requirements of the project.

REACT NATIVE

React Native is an open-source framework developed by Meta (formerly Facebook) that enables developers to build mobile applications for both iOS and Android platforms using a single codebase. It is based on JavaScript and leverages the popular React library for building user interfaces. This approach significantly reduces development time and cost, as there is no need to maintain separate codebases for different platforms. One of the key features of React Native is its ability to render components using native APIs, providing a look and feel that is consistent with

platform-specific user experiences, as illustrated in Figure 1. Unlike hybrid frameworks that rely on WebView, React Native applications run natively, which typically results in better performance and smoother animations. React Native also supports a wide range of third-party libraries and native modules, allowing developers to integrate device specific features like camera access, GPS, and push notifications with ease. Its hot reloading functionality speeds up the development process by allowing developers to see changes in real time without rebuilding the entire app. Overall, React Native is a powerful and flexible solution for cross-platform mobile development, making it a popular choice among startups and large companies alike that aim to reach a broader audience with a unified development effort (Eisenman, 2015).

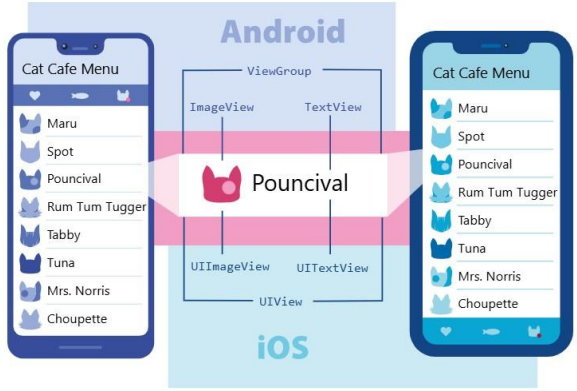


Figure 1. Examples of views used in Android and iOS applications (React Native – Native Components, 2025).

Expo is used for developing React Native applications. Expo has been actively involved in the development of React Native from the very beginning. Expo released the first community libraries just a few months after the release of React Native in 2015. In the same year, Expo developed the React Native Directory – a directory where React Native libraries can be searched and explored.

In 2016, Expo participated in organizing the first React Conf event and was the first independent developer community to support the release of a React Native version. Expo helped add async functions to React Native as well as update the new architecture in the open-source project (Saarikoski, 2025).

In 2024, Expo became the first official React Native framework for application development (React Conf, 2024).

Expo is a toolchain built around React Native to help developers start their projects quickly. Expo provides a set of tools and services to develop, build, deploy, test or even run simulators on the specific platform from the same codebase. Specifically, it offers a collection of ready solutions such as device accelerometer, camera, notifications, geolocation, etc. (Van, 2020).

STORAGE SYSTEMS AND DATABASES

AsyncStorage is a local data storage system used in React Native applications. It allows developers to store data on the user's device in the form of key-value pairs. The storage mechanism is asynchronous, meaning that data operations such as saving or retrieving run in the background without interrupting the main execution thread of the application. This contributes to better app performance and a smoother user experience (Huynh, 2023).

Stored data in AsyncStorage remains persistent across app restarts, making it suitable for maintaining state, user preferences, session tokens, or other small pieces of data that need to be retained between uses. However, it's important to note that AsyncStorage does not encrypt data, which makes it easily accessible but also raises security concerns for sensitive information.

Due to its simplicity and ease of use, AsyncStorage is often chosen for basic offline data persistence in React Native apps, particularly when working with smaller datasets that do not require complex querying or relational structures.

SQLite is a lightweight, relational database engine commonly used in mobile applications for local data storage. In React Native, SQLite allows apps to store structured data directly on the device in a persistent and efficient way. It uses SQL queries to create, read, update, and delete data within local database files (Gaffney, 2022).

This approach is especially useful for applications that require managing moderate amounts of data offline, such as note-taking apps, inventories, or personal information managers. SQLite supports complex queries and relationships between data, providing more flexibility than simple key-value storage systems (Kreibich, 2010).

Because the database is stored locally, React Native apps using SQLite can operate without an internet connection, ensuring data availability at all times. Although it may have some limitations regarding concurrent writes and schema migrations, SQLite remains a reliable solution for many mobile apps needing fast and persistent local storage (Bhosale, 2015).

TinyBase is a small and fast database library for managing state and data in JavaScript apps, including React Native. It provides a flexible in-memory data store focused on ease of use and performance, rather than persistent storage (Danielsson, 2016).

In React Native, TinyBase helps efficiently handle complex state and relational data during app runtime. It supports tables, records, and relationships, similar to relational databases but without slow disk operations.

Its lightweight design and simple API make it easy to integrate with React Native's component lifecycle, enabling automatic updates and smooth rendering when data changes.

While TinyBase doesn't offer built-in persistent storage, it can be used alongside tools like AsyncStorage or SQLite to save and load data as needed. Overall, TinyBase is a useful tool for managing app state and complex data with high performance (TinyBase, 2025).

Realm is a mobile database built for modern apps, offering an easy-to-use, high-performance, and offline-first solution. Unlike traditional databases, it uses an object-oriented model, letting developers work with native objects instead of SQL queries (Danielsson, 2016).

In React Native, Realm efficiently manages complex data locally and supports reactive updates, ensuring UI components sync automatically when data changes. It stores data in a zero-copy format, improving read/write speeds, and offers features like encryption, synchronization with backend servers, and fine-grained permissions.

Realm handles complex relationships and queries without losing performance, integrates well with React Native's lifecycle, and supports transactions for data integrity. Although it requires setup and can increase app size, its benefits make it ideal for data-heavy mobile apps (Andersson, 2018).

In summary, Realm provides a powerful, scalable, and secure local database with real-time updates and synchronization, perfect for modern mobile development (Cobley, 2022).

IMPLEMENTATION

The motivation for testing various mobile database solutions stemmed from the development of a German collocation dictionary within the scope of the DeSKoll project (DeSKoll, 2025). To streamline development across platforms and avoid maintaining separate codebases for iOS and Android, the React Native framework was selected.

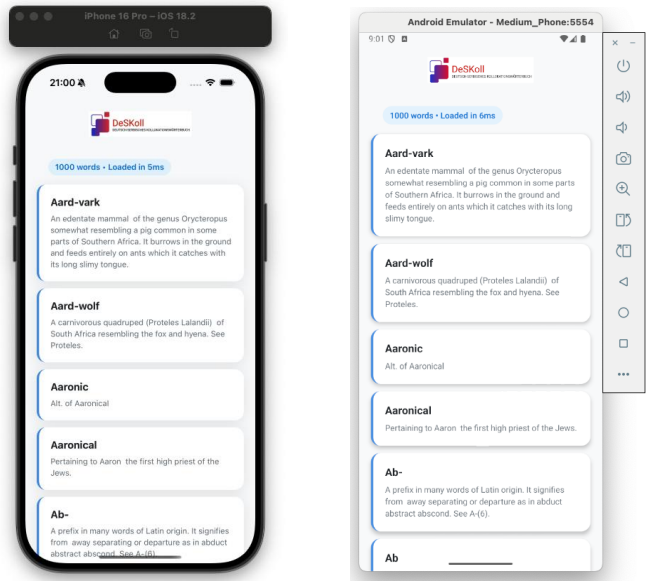
For experimental purposes, a simplified English word dictionary was utilized (OPTED-Dictionary, 2025), consisting of four columns: Word, Count, POS (part of speech), and Definition. In order to reduce complexity during testing, only the Word and Definition columns were considered, while the specific semantic content of the data was not of significance.

Datasets containing 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, and 100000 rows were loaded using four distinct methods. The upper limit of 100,000 rows was chosen based on the projected maximum size of the final collocation dictionary. The dictionary will have significantly fewer rows but more columns, so these measurements will be relevant.

Stojić, D., et al. (2025). Performance comparison of various datebase types in react native applications. *STED Conference 14*(2), 201-208.

Identical user interface layouts were developed for each of the four data loading approaches, ensuring consistency across test scenarios.

All tests were performed in the Expo Go environment, utilizing an iPhone simulator and an Android emulator to ensure cross-platform compatibility. The layouts used for testing are presented in Figure 2.



(a) (b)

Figure 2. Application layout in React Native for testing on simulator for (a) iOS, (b) Android.

The first loading method uses AsyncStorage for local storage and display of the dictionary. When the application starts, the *useEffect* hook is triggered, performance measurement begins, and the app attempts to load the dictionary from memory. If data exists, it is parsed and set to state, then displayed using a *FlatList*. The data flow diagram from the AsyncStorage local database to the screen is shown in Figure 3.

Figure 3. Data flow: AsyncStorage → Screen (React Native).

In the second approach, an SQLite database is used for storing and displaying the dictionary. Data from the SQLite database is loaded using the expo-sqlite API through the useSQLiteContext hook. When the screen is focused, the *loadData* function is triggered, executing the SQL query *SELECT * FROM Dictionary* to retrieve all entries from the dictionary table. The query results are stored in the component's state (data), which automatically triggers the rendering of the list. The time taken to load the data is measured using Date.now() and displayed to the user. The dictionary is displayed using the *FlatList* component, which efficiently renders each word and its definition as a list on the screen. The data flow diagram from the SQLite local database to the screen is shown in Figure 4.

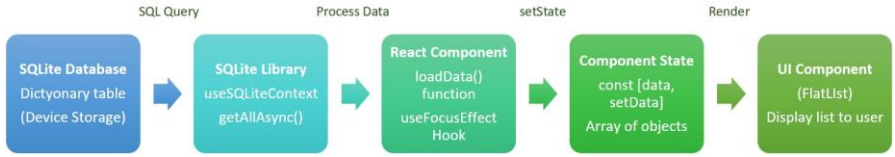


Figure 4. Data flow: SQLite → Screen (React Native).

The third approach combines an SQLite database with the TinyBase library for efficient data management and display. At the start, the Dictionary.db database is copied into the app's working directory. After that, the SQLiteProvider enables access to the database using expo-sqlite. The DataLoader component executes the SQL query *SELECT Word, Definition FROM Dictionary* to retrieve all words and their definitions. The resulting data is then written into the TinyBase store, where each word is used as a row ID and the definition as a cell. Once the data is loaded, the TinyBase context (TinyBaseProvider) wraps the entire app to allow access to the data via TinyBase hooks. The *Dictionary* component uses *useRowIds* to get all word keys, while *DictionaryItem* uses *useCell* to display the definition for each word. The data is rendered using *FlatList*, allowing for performant and dynamic display of dictionary items within the user interface. The data flow diagram from the SQLite local database throw TinyBase library to the screen is shown in Figure 5.

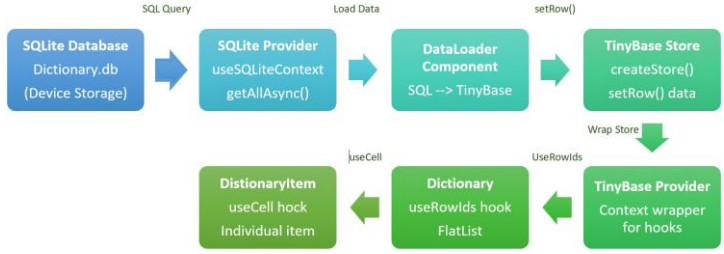


Figure 5. Data flow: SQLite → TinyBase → Screen (React Native).

The fourth method uses Realm, a high-performance local database, for storing and displaying the dictionary within the React Native application. The loading process starts by checking whether the Realm file (Dictionary.realm) exists in the local *documentDirectory* folder. If it does not exist, the database is automatically copied from the assets directory using the expo-file-system and expo-asset libraries. This process is automated in the *copyRealmFileIfNeeded* method, enabling the use of a pre-populated database. After copying, the application uses the RealmService class to open the database and initialize the schema (DictionarySchema). The data is then retrieved directly from the database and displayed using the FlatList component. The data flow diagram from the Realm local database to the screen is shown in Figure 6.

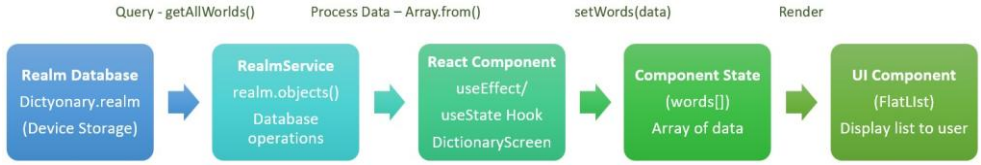


Figure 6. Data flow: Realm → Screen (React Native).

RESULTS AND DISCUSSION

To determine the most suitable solution for local data storage in mobile applications built with the React Native framework, a comprehensive performance evaluation was carried out involving four widely used database systems: SQLite, AsyncStorage, TinyBase, and Realm. The goal was to assess efficiency in handling different data set sizes, with a focus on load speed.

The performance tests were conducted on both an iOS simulator and an Android emulator, providing cross-platform insights into how each database behaves under similar conditions. The

evaluation involved loading predefined datasets of increasing sizes, specifically: 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, and 100000 records. These datasets were designed to simulate real-world application usage, ranging from simple to complex data structures.

For each test scenario, the time required to load the dataset into memory was measured and recorded in milliseconds (ms). To ensure reliability and minimize the impact of potential background activity or fluctuations in the simulator/emulator environments, each test was repeated multiple times, and average values were calculated.

The numerical results of testing on the iOS simulator are shown in Table 1, and the graphic representation is presented in Figure 6.

Table 1. Measurement of the data load time in iOS simulator

Inputs	SQLite (ms)	AsyncStorage (ms)	TinyBase (ms)	Realm (ms)
100	36	3	23	1
200	41	3	37	2
500	47	5	50	3
1000	55	6	102	4
2000	59	6	191	9
5000	61	11	302	20
10000	68	18	352	46
20000	88	86	380	84
50000	141	238	1781	226
100000	224	437	3364	466

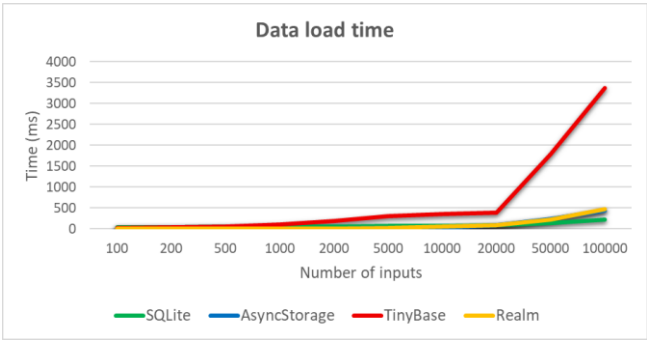


Figure 6. Measurement of the data load time in iOS simulator.

Similar results were obtained during testing on both the Android emulator and the iOS simulator, indicating consistent performance across platforms. The numerical results of testing on the Android emulator are shown in Table 2, and the graphical representation is presented in Figure 7.

Table2. Measurement of the data load time in Android emulator.

Inputs	SQLite (ms)	AsyncStorage (ms)	TinyBase (ms)	Realm (ms)
100	40	5	28	1
200	45	6	41	2
500	51	7	55	3
1000	62	8	113	5
2000	68	8	283	10
5000	82	11	319	21
10000	103	17	384	50
20000	148	143	434	86
50000	273	382	1783	224
100000	490	621	3526	578

SQLite provides stable performance on both platforms. The loading times of small amounts of data are slow compared to other databases, but the growth is relatively uniform with the increase in the amount of data. this indicates stable and predictable performance.

AsyncStorage is very efficient for small to medium datasets, but not suitable for larger amounts of data. The responses are extremely fast for small amounts of data, however, after 20,000 entries there is a sudden spike, which indicates exponential growth and loss of efficiency with larger datasets.

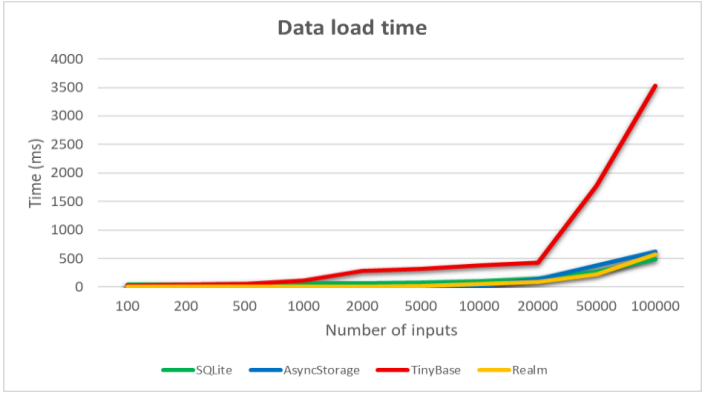


Figure 7. Measurement of the data load time in Android emulator.

TinyBase is not optimized for working with large datasets and shows extremely poor performance on both platforms. The performance is significantly weaker even with small datasets, and at 1,000 entries the growth is large, while at 20,000 entries it is exponential, which makes it unusable for large amounts of data.

Realm has the best performance for small amounts of data, however, performance degrades as the dataset grows. The growth is relatively linear up to 10,000 entries, after which there is a noticeable slowdown.

From the obtained results it can be concluded that Realm and AsyncStorage are the fastest for small datasets. The most stable performance was shown when using SQLite, with a uniform and predictable increase in loading time. The worst performance was obtained with TinyBase, with sudden exponential growth and very long times for large datasets.

CONCLUSIONS

The comparison of different types of data storage in React Native applications shows that the optimal choice of storage is highly context-dependent. There is no single solution that works best in every situation.

The choice depends on the specific requirements of the application, the expected data volumes, and the targeted platforms. For applications with predictable data sizes up to 10,000 entries, AsyncStorage represents an optimal choice due to its simplicity and good performance. For applications that require scalability and handling large amounts of data, SQLite is the only solution that provides consistent performance across all scenarios. Realm can be used in specific cases where its advantages for smaller data volumes can be leveraged, but it requires careful testing and monitoring. TinyBase is not recommended for production applications due to consistently poor performance on both platforms.

In summary, developers should carefully evaluate their application's requirements—such as data size and complexity, need for local storage, performance, and implementation simplicity – to choose the most appropriate solution.

Further studies should concentrate on evaluating performance using real-world datasets and diverse operation types, along with examining how factors like memory usage, battery

consumption, and network traffic influence the overall efficiency of storage solutions in React Native applications.

ACKNOWLEDGMENT

This research was supported by the Science Fund of the Republic of Serbia, PROMIS, Grant no. 10916, German-Serbian Collocation Dictionary for German Language Learning and Teaching – DeSKoll. Also, by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia, and these results are parts of the Grant No. 451-03-136/2025-03/200132, with University of Kragujevac - Faculty of Technical Sciences Čačak.

DECLARATIONS OF INTEREST STATEMENT

The authors affirm that there are no conflicts of interest to declare in relation to the research presented in this paper.

LITERATURE

Alsaid, M. A. M. M., Ahmed, T. M., Jan, S., Khan, F. Q., & Khattak, A. U. (2021). A Comparative Analysis of Mobile Application Development Approaches: Mobile Application Development Approaches. *Proceedings of the Pakistan Academy of Sciences: a. Physical and computational sciences*, *58*(1), 35-45.

Andersson, T. (2018). Analysis and quantitative comparison of storage, management, and scalability of data in Core Data system in relation to Realm.

Bhosale, S. T., Patil, T., & Patil, P. (2015). Sqlite: Light database system. *Int. J. Comput. Sci. Mob. Comput*, *44*(4), 882-885.

Cobley, P., & Geneste, G. (2022). Realm. In *Mobile Forensics–The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices* (pp. 181-221). Cham: Springer International Publishing.

Danielsson, W. (2016). React Native application development: A comparison between native Android and React Native.

DeSKoll – German-Serbian Collocation Dictionary for German Language Learning and Teaching. (2025). Retrieved May 25, 2025, from https://deskoll-dictionary.kg.ac.rs/

Eisenman, B. (2015). Learning react native: Building native mobile apps with JavaScript. " O'Reilly Media, Inc.".

Gaffney, K. P., Prammer, M., Brasfield, L., Hipp, D. R., Kennedy, D., & Patel, J. M. (2022). SQLite: past, present, and future. *Proceedings of the VLDB Endowment*, *15*(12), 3535-3547.

Huynh, C. (2023). MealPointer Mobile Application.

Kreibich, J. (2010). *Using SQLite*. " O'Reilly Media, Inc.".

OPTED-Dictionary – The Online Plain Text English Dictionary. (2025). Retrieved May 25, 2025, from https://www.kaggle.com/datasets/dfydata/the-online-plain-text-english-dictionary-opted/data

React Conf (Ohjaaja). (2024). React Conf Keynote (Day 2) [Video recording]. Retrieved May 22, 2025, from https://www.youtube.com/watch?v=Q5SMmKb7qVI

React Native – Native Components. (2025). Retrieved May 23, 2025, from https://reactnative.dev/docs/intro-react-native-components

Saarikoski, A. (2025). Expo-sovelluskehyksen hyödyntäminen React Native-sovelluskehityksessä.

TinyBase – A reactive data store & sync engine. Retrieved May 25, 2025, from https://tinybase.org/

Van, H. (2020). Building a universal application with React and React Native.