Expertise
and insight
for the future

Trung Ngo

A Method for Live SQL Query Subscription in React Native

Metropolia University of Applied Sciences
Bachelor of Engineering
Smart Systems
Bachelor's Thesis
5 May 2021

Author	Trung Ngo
Title	A Method for Live SQL Query Subscription in React Native
Number of Pages	36 pages
Date	5 May 2021
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Smart Systems
Instructors	Sami Sainio, Senior Lecturer

This thesis proposes integrating the well-established embedded SQLite database into React Native in a reactive and unintrusive manner, allowing developers to take advantage of their existing SQL data-wrangling skillset. The current best practices in client-side global data management in React Native, namely Redux and MobX, enables developers to leverage the unidirectional data flow programming paradigm to simplify data organization without having to maintain a complex web of callbacks and observers to reflect the latest data in the application UI. However, they leave important aspects of data work, such as filtering, joining and aggregating to the developers, leading to ad-hoc data pipelines that vary from project to project in style and efficiency. Using a central registry for query rerunning combined with run-time schema dependency introspection, SQLite query results could be retrieved and processed as though they were reactive arrays that are automatically kept updated. The minimal API of the library encourages developers to do more work in the declarative logic programming paradigm of SQL thus simplifying application code. The performance in the initial version was acceptable with round-trip latencies low enough to not incur noticeable UI jitters. Moreover, the library was structured so that the core query rerunner is framework-agnostic and could be ported to different UI frameworks and even to backend frameworks in future works.

Keywords	

Contents

List of Abbreviations

API	Application Programming Interface. The set of functions, data types, and behavioral constraints defining the interaction between the library user's code and the library provider's code.
REST	REpresentational State Transfer. A pattern of HTTP API design in which the system's state is structured into uniquely identifiable resources that could be manipulated with common HTTP operations like POST, PUT, GET, DELETE.
SQL	Structured Query Language. A language for declaratively manipulating data. The word is pronounced either as $/\varepsilon s\ kju\text{:}\ \varepsilon l/$ or $/'si\text{:}\ kw\text{ə}l/$. This thesis adopts the second pronunciation.
UI	User Interface
DBMS	Database management system. Software for maintaining, querying, and updating databases.
DOM	Document Object Model. A tree of objects representing the content to be rendered onto the screen.
MVC	Model View Controller
MVVM	Model View View-Model
DDL	A subset of SQL that deals with defining relations.
DML	A subset of SQL that allows modifying relations' content.
DQL	A subset of SQL that allows querying relations.

1 Introduction

In the last decade, Internet-connected smart mobile devices have become not only ubiquitous but also crucially important to a modern person's everyday life. Statista reported that in 2021, on average, 54.18% of all global Internet web traffic was accessed through a mobile device (excluding tablets) with Africa and Asia leading at 63.9% and 69.9%, respectively. [1] Another study also by Statista revealed a steady growth in worldwide mobile app revenues from 97.7 billion USD in 2014 to 693 billion USD in 2021 and the number is projected to reach 935.2 billion USD in 2023. [2] See Figure 1 below.

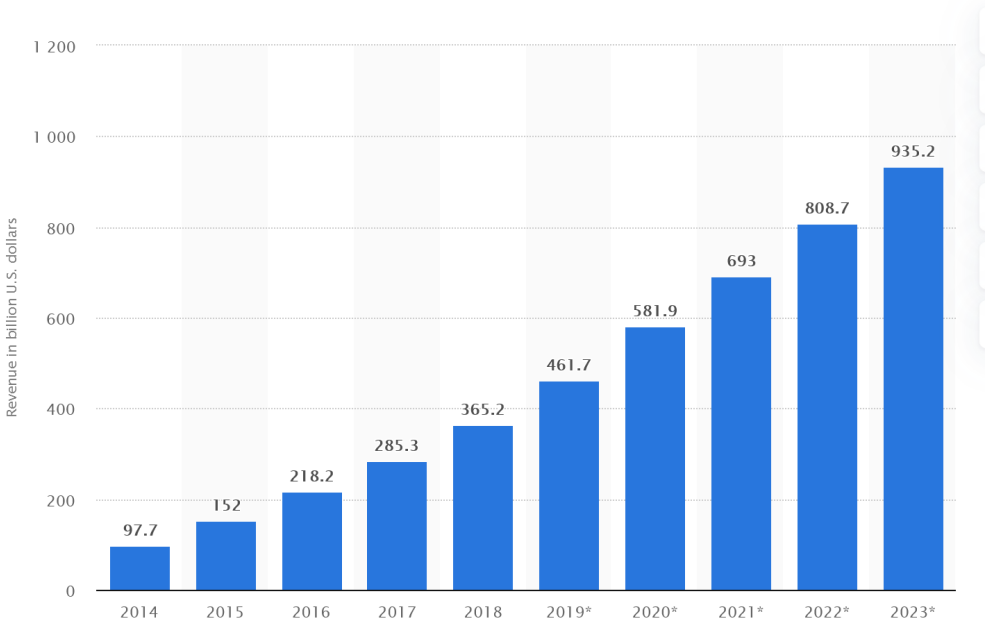

Figure 1. Worldwide mobile app revenues in 2014 to 2023

As such, companies and individual developers are increasingly looking at mobile applications as an attractive platform to develop for. A 2017 report by StackOverflow regarding developer hiring trends during 2017 suggested strong demand for both iOS and Android mobile development jobs while alternative application development platforms such as Windows and MacOS desktop platforms and full-stack web platforms such as Python Django, Ruby on Rails and WordPress all showed saturation in supply. The figures are depicted in Figure 2.

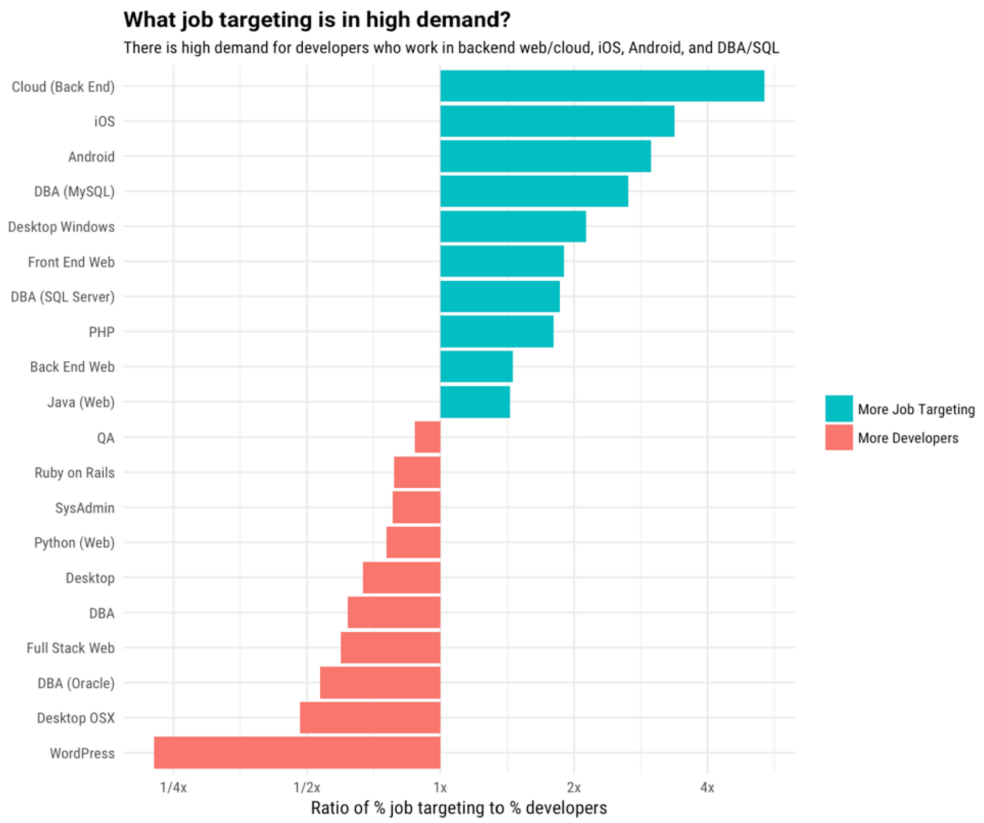

Figure 2. StackOverflow's data on software engineering job demand in 2017 [3]

User-facing mobile applications tend to face a similar set of following operational and design constraints. They are likely to be Internet-powered, fetching and saving data and content from multiple sources and backend services. The data is commonly communicated through an HTTP API design pattern called REST where data is broken down into disparate but connected objects called entities, organized through unique URLs. The mobile application developers have to then combine, filter, and assemble the data that came from various sources into a coherent structure to inform and present to the user. This process is called data enrichment. The enriched data in different UI views should stay consistent when one or more pieces of the constituent data change. Furthermore, for many applications, the data should stay persistent across sessions of interaction, even when network connection is not available. Ideally, the programming environment should make data persistence transparent to the application developers, so the same interface is provided regardless of whether the developers are interacting with online or offline data.

One of the most popular approaches for developing mobile applications in recent years has been using React Native, a library introduced by Facebook in 2015, based on their own ReactJS library, which also enjoyed considerable success and wide adoption on the web platform. A 2020 survey by Statista shows that 42% of the respondents have used React Native for cross-platform mobile app development during 2020. [4] ReactJS and React Native simplifies application development through their use of the unidirectional data flow programming pattern and through the fact that they enable the straightforward decomposition of UI code recursively into manageable and reusable components.

In the React Native ecosystem, the most common method of handling global data is through Redux. However, Redux lacks rich querying support so programmers often write ad-hoc and inefficient data joining and/or filtering code for each application. Also, Redux was designed for in-memory data storage so programmers must write explicit data saving and loading code at application start and stop.

On the other hand, there exists a reliable data manipulation engine that supports rich querying, joining, filtering operations, which is SQLite. The library is persistent by default so reading and writing to memory and disk is the same interface. A bonus is that SQL as a language has been continuously gaining adoption on an industry-wide scale since its inception in 1974. According to the Stack Overflow 2020 Developer Survey, 56.9% of the surveyed professional developers reported that they have used SQL in the last year comparable to 69.7% for JavaScript. Furthermore, 30.6% reported that they have used SQLite specifically in the last year. [5] It can be concluded that most developers have at least some exposures to writing queries in SQL and that if employing SQL inside React Native would be made simple, a significant number of developers would be able to take advantage of their existing SQL skills.

However, it is not yet straightforward to use SQL with React Native in an intuitive way. All existing integrations of SQLite into React Native use the fetch model, meaning the programmer must explicitly fetch and re-fetch the data from SQLite when they anticipate that the data has been changed somehow. On top of that, every interaction with the database involves a non-trivial number of steps, which discourages wide-scale and deep integration of SQL into the codebase.

As part of the work as a mobile application developer for Gofore Oyj, the author developed an integration layer for SQLite into React Native that adapts the query engine to the reactive programming model of React Native: programmers subscribe to queries and they are automatically refreshed to reflect the latest data in storage.

This thesis describes the theoretical backing and implementation as well as issues faced during the work.

2 Theoretical Background

2.1 The Reactive Programming Model and React Native

ReactJS is a library for writing graphical user interface created by Jordan Walke to solve the various code maintenance problems Facebook was facing as their JavaScript applications grow in code size and number of maintainers. Tom Occhino first publicly introduced it during JSConfUS 2013 in a presentation and Jordan Walke called “JS Apps at Facebook”. [6] It has since grown steadily in adoption to become the most popular frontend JavaScript web framework as of 2021. See Figure 3 below.

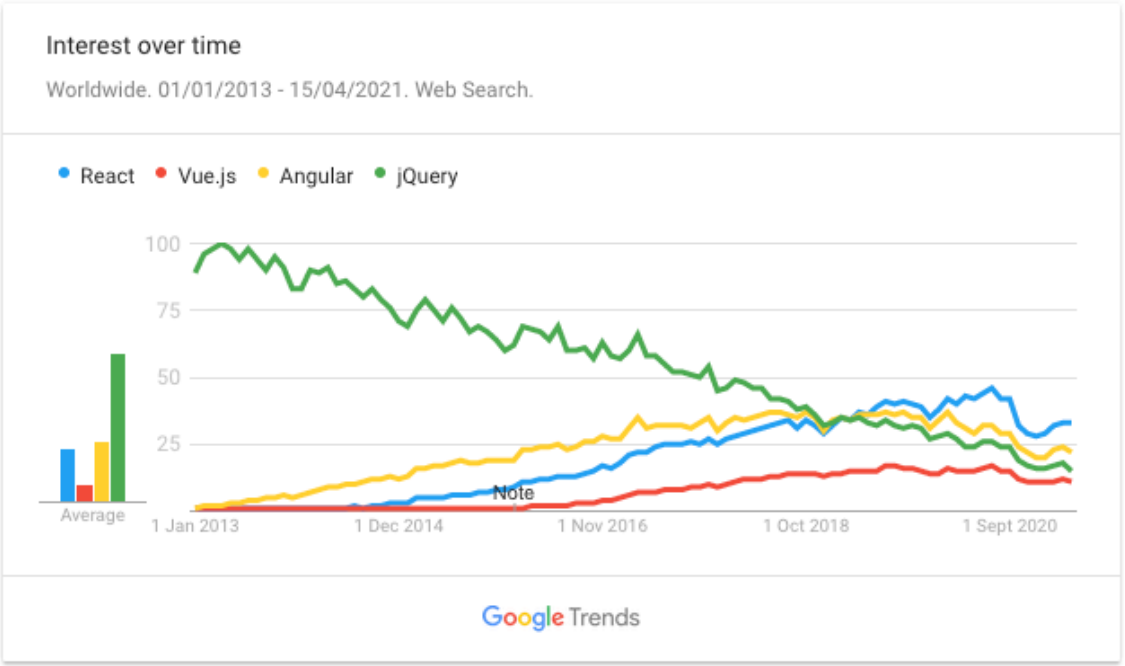

Figure 3. Google Search interest for popular frontend web frameworks from Jan 2013 to April 2021

At a high level, ReactJS allows developers to write a declarative description of a UI component tree that they want to be rendered onto the screen. The component tree is a function of a global state object, which is called as part of the render loop. [6]

```
state = get_initial_state()
object_tree = render(state)

while True:
      event = wait_for_external_events()
      state = process(event)
      new_object_tree = render(state)
      reconcile_object_tree(object_tree, new_object_tree)
```

Listing 1. ReactJS rendering model

The $render()$ function is composable; it can call other render functions to create custom components that in turn call other render functions to create even lower-level components. The top-level state object is passed down, possibly broken down into smaller pieces more relevant to the child components.

```
ReactDOM.render(<App/>, document.getElementById('root'));

function App() {
  return (
      <div>
        <h3>Hello world!</h3>

        <ChildComponent customProperty1={"custom value"}/>
      </div>
  )
}

function ChildComponent({customProperty1}) {
  return (
      <div>
        {customProperty1}
      </div>
  )
}
```

Listing 2. A typical “Hello world” application in ReactJS

ReactJS component code is usually written in an extension of JavaScript called JSX that adds a custom syntax that looks like XHTML to make it easy to write trees of document element tags inside JavaScript. Custom tags can be created from a function that itself returns a tree of elements, such as the $ChildComponent()$ function in the example.

Whenever the state object changes, the $render()$ function is called again, resulting in a new virtual and lightweight object tree. The new object tree is then recursively compared against the old tree to efficiently calculate which properties or members of the old tree

should be changed or removed, or which new members should be added so that the old tree could be transformed into a state that is the same as the new tree. This is effectively equivalent to destroying and replacing the object tree with a new one for every rendering cycle, a relatively simple model to reason about, but without the heavy cost of actually replacing the whole tree. [7]

This UI programming model is arguably simpler than the previous models like MVC or MVVM where developers have to maintain the data model objects and the view tree themselves and write binding code that updates the view objects when the data model changes. As the number of features grows, the number of binding connections in the object graph would grow unmanageably and so does the complexity of understanding what changes because of what. [8]

In ReactJS, data flows in one way from the root component into child components and finally into the native UI components of the platform. Every interaction from the user such as pressing a button or typing into an input field is interpreted as an event and would be passed to an event handler that would modify the global state. The new state would then trickle down into the component tree and any modification would show up. This pattern is called the unidirectional data flow.

As a side note, the individual components are allowed to have private states that affect the final rendered component tree. Nevertheless, these private states are usually reserved to UI-specific states like which tab of a multi-tab section is currently selected. We only care about the global state that is passed down the tree from the root component.

ReactJS was initially written for the client-side Web browser environment but in 2015, Facebook introduced React Native, adapting ReactJS to other platforms like native iOS, Android, and, recently, Microsoft Windows and macOS. The kinds of native components available to construct the UI component tree vary from platform to platform but the programming model remains the same. That is why React Native adopted the slogan “Learn once, write anywhere”. [9]

This thesis focuses on React Native instead of ReactJS because the work was inspired during the development of a commercial mobile application written using React Native.

Also, for the time being, React Native is a more feasible target because there is already a reliable WebSQL implementation in React Native.

2.2 Current Data Storage Solutions in React Native

On both React Native and ReactJS, the two most popular options to store and retrieve application-wide global data is Redux and MobX. They take the place of the “state” object in the simplified rendering loop in Listing 1. Data is organized into trees of properties that could be a primitive value, sub-object, or array of primitive values or sub-objects. [10] They are essentially hierarchical databases similar to Microsoft Windows’ Registry.

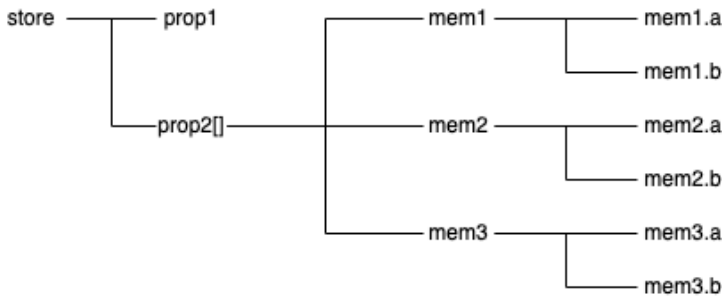

Figure 4. The hierarchical structure of data stores in Redux and MobX

The libraries also provide an application-wide event bus that individual view components can send “actions” to. Actions are event-like objects that specify a command and some arguments to that command; a “reducer”, a function that takes the current state and the action and executes the specified command to get a new state object, replacing the old one, then executes the action. [10]

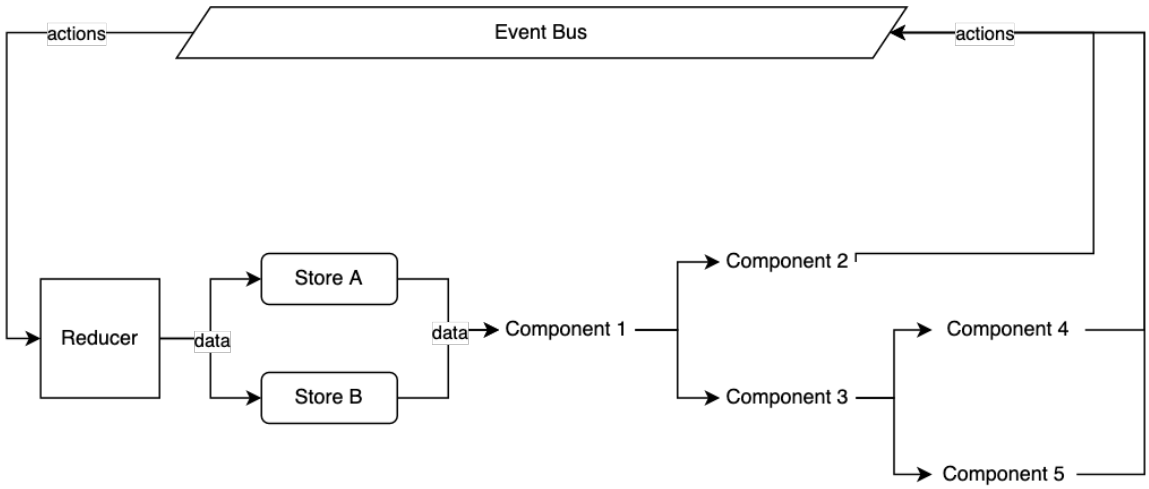

Figure 5. The unidirectional data flow with actions and reducers and a global event bus

As illustrated in Figure 5, this architecture moves the event handlers out of the React components into a new abstraction layer so that multiple components handling the same kind of event do not have to duplicate the handling code and also makes business logic handling code more organized.

Redux and MobX are both in-memory data structures. To provide offline persistence, a common pattern is to integrate them with React Native's own offline storage mechanism called AsyncStorage, a simple persistent key-value storage engine.

The application would register a subscription function that is called every time the datastore is updated. Since the keys and values in AsyncStorage must be string, the function serializes the store to a JSON string and saves that into AsyncStorage. At application start, the stores are "rehydrated" from the JSON snapshot in AsyncStorage. This approach works on small amounts of data but breaks down quickly as the store's size and the number of write operations per second scale up.

2.3 The Relational Model and SQL query language

Edgar F. Codd first introduced the relational database and the relational model in his 1970 paper "A relational *model of data for large shared data banks",* culminated from his research at IBM in the late 1960s. [11] The relational model aims to address the data

dependency issues in existing hierarchical and network data storage and retrieval models at the time that Codd identified:

- *ordering dependence*: data consumption ordering should not depend on data storage ordering,

- *indexing dependence*: indexing should only exist transparently as a performance improvement, not part of the data access interface; and finally

- *access path dependence*: the database should not assume that related pieces of data be found through a particular access path.

The first two points are mostly no longer relevant in modern data systems, but the last point is still applicable to hierarchical data stores, including Redux and MobX.

To understand the relational model, it is necessary to understand the mathematical concept of relation. Given:

- a set of attributes $A = \{A_1, A_2, A_3, \dots, A_n\}$
- $Dom(A_i)$ is the set of all possible values (domain) of the attribute A_i
- The cartesian product $C = Dom(A_1) \times Dom(A_2) \times Dom(A_3) \times \dots \times Dom(A_n)$
- A constraint $p(A_1, A_2, A_3, \dots, A_n)$ is a Boolean-valued function, also called a predicate

Then, a relation $R(A_1, A_2, A_3, \dots, A_n)$ is a subset of the cartesian product C over which the constraint p returns true. A relation is thus also a set and as can be seen below, many set operations apply to relations as well. [12]

Consider an example: Attribute A has 3 possible values x, y, z; attribute B also has 3 possible values $1, 2,$ and 3. The cartesian product $A \times B$ gives 9 possible combinations as shown in Figure 6 below.

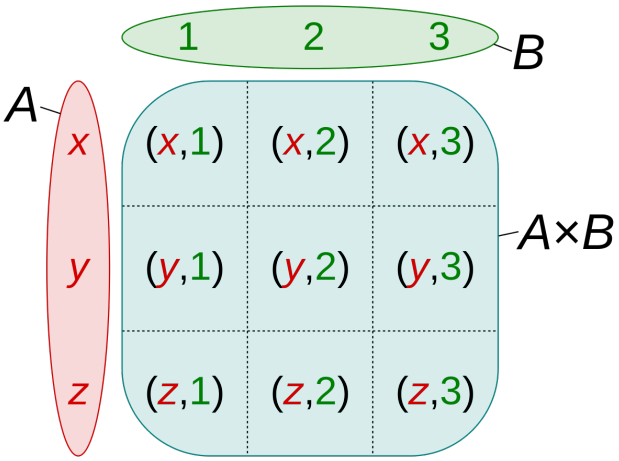

Figure 6. The Cartesian product of two sets

Let's introduce a predicate $p(A, B)$ that returns true for the three pairs $(x, 1), (x, 2)$, and $(y, 1)$. Then the subset $\{(x, 1), (x, 2), (y, 1)\}$ is a relation R that obeys the constraint p.

A relation is a generalization of the concept of function where there can be multiple overlapping mapping pairs. In this example, the value x in the attribute A maps to the set $\{1, 2\}$ in the attribute B and the value 1 in the attribute B maps to the set $\{x, y\}$ in attribute A. This is different from a function, where there can only exist one mapping for every input set into the output set (multiple items from the input set may map to the same output member, however).

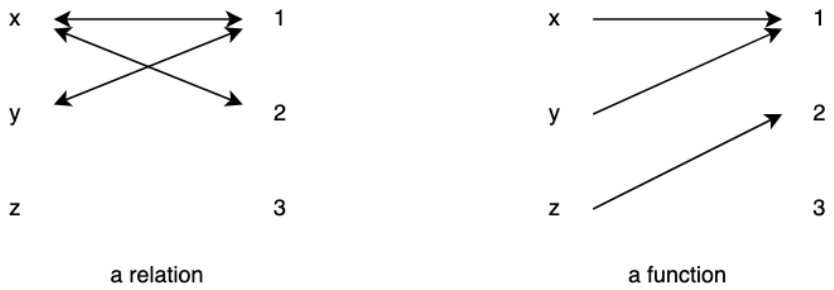

Figure 7. Example of a relation and a function

As an implementation detail, a relation is usually implemented as a list of fix-sized tuples stored in an array, similar to a table. Therefore, in many relational databases, a relation is also known as a table, an attribute as a column, and a tuple as a row.

In practice, a relational database system stores fact on multiple different entities by breaking down the facts into relations/tables, each tuple in the relations stores a fact on the entities. A simple relational database describing an address book can illustrate this.

There are 3 entities that the system manages: person (P), address (A), and the fact that a person lives in an address (PA).

- A person entity has 3 attributes: a unique identifier (id), first name ($first$), last name ($last$).
- An address entity has 4 attributes: a unique identifier (id), street address ($street$), postal code ($post$), city ($city$).
- A "person having an address" fact has 2 attributes: person identifier (id_p) and address identifier (id_a). This is a separate entity because a person may have multiple addresses and many people can live under the same address.

Each attribute of the above entities has an infinite set of possible values and the cartesian product for each set of attributes ($P.id \times P.first \times P.last$, for example) also contains an infinite number of members. Adding a constraint requiring that the tuple $\{P.id, P.first, P.last\}$ describes a person that the database administrator knows personally reduces the cartesian product to a relation with a finite number of members.

Codd defined some operations on relations, collectively known as the relational algebra. [11]

- *Selection*: a unary operation. Restrict the relation to a subset in which the given selection condition holds.

$$Select(A, Cond) = \{a \mid a \in A, Cond(a)\}$$

- *Projection*: a unary operation. Pick certain columns of a relation, removing the others, and then remove from the resulting array any duplication in the rows.
- *Union*: a binary set operation, where the two operands must share the same set of attributes. The resulting relation contains all tuples in both the operands.

$$Union(A, B) = \{a \mid a \in A \vee a \in B\}$$

- *Difference*: a binary set operation, where the two operands must share the same set of attributes. The resulting relation contains the tuples that exist only in the left operand.

$$Diff(A, B) = \{a \mid a \in A \wedge \neg\, a \in B\}$$

- *Intersection*: a binary set operation, where the two operands must share the same set of attributes. The resulting relation contains the tuples that exist in both operands.

$$Intersect(A, B) = \{a \mid a \in A \wedge a \in B\}$$

- *Cartesian product*: a binary set operation, two operands do not have to share the same set of attributes. Combine two relations into a relation of ordered pair (a, b):

$$A \times B \;=\; \{(a, b) \mid a \in A \wedge\; b \in B\}$$

- *Join*: combine two relations based on a given condition. Conceptually, this is the same as taking a selection over the cartesian product of the relations.

$$Join(A, B) = \;\{(a, b) \mid a \in A \wedge b \in B \wedge\; a(attr_1)\; \theta\; b(attr_2)\}$$

  In which, a and b are tuples from the relations, $a(attr_1)$ and $b(attr_2)$ are attributes from the tuples and θ is a binary relational operator from the set $\{<, \leq, =, \neq, >, \geq\}$.

  The general case is called a θ-join. If θ is the equality ($=$) operator, then it is called an equijoin. If an equijoin is done on attributes having the same name from the relations, then it is called a natural join.

  There are more complex forms of join, but they do not significantly change the model and thus are not of interest to this thesis.

Composing the relational algebra's operations in a specific order gives a computationally feasible mechanism for deriving new knowledge from ground facts. Figure 8 presents two possible trees of execution to calculate a relation containing the first and last names of people in Helsinki who are above 18 years old. While equivalent, the first tree is more

efficient in both time and memory compared to the second since the number of candidate tuples for the join operation is significantly reduced by the select operations.

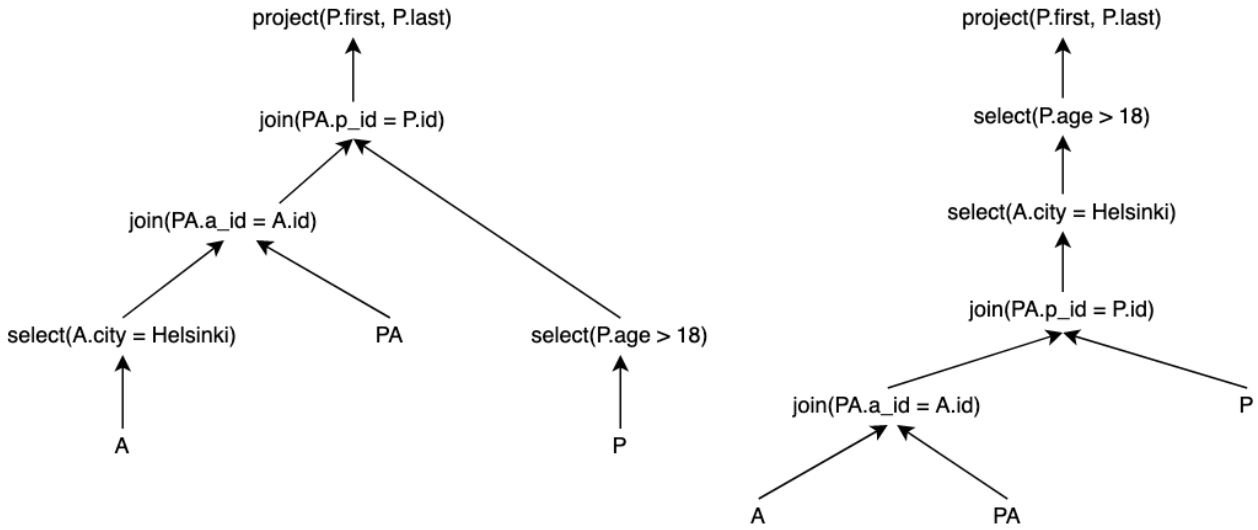

Figure 8. Two possible relational algebra execution trees giving the same result

Together with relational algebra, Codd also defined relational calculus, a query language based on first-order predicate logic over tuple variables with existential quantifiers. [13] The relational algebra queries above are functionally equivalent to this relational calculus query:

$$\exists\, p(first, last) \wedge \exists p \in P \wedge \exists a \in A \wedge \exists pa \in PA \wedge$$
$$p(age) > 18 \wedge a(city) = Helsinki \wedge pa(id_p) = p(id) \wedge pa(id_a) = a(id)$$

Listing 3. A query in relational calculus

This reads as "find all tuples $(first, last)$ that are sub-tuples of p *AND* exists a tuple p as a member of P *AND* exists a tuple a as a member of A *AND* exists a tuple pa as a member of PA AND the age property of p is greater than 18 *AND* the $city$ property of a is $Helsinki$ *AND* the id_p property of pa matches the id property of p *AND* the id_a property of pa matches property id of a".

The relational calculus query is more declarative than the two queries written in relational algebra because the latter specify the order of operations to be taken whilst the former only specifies the logical connection between the attributes and variables.

Codd provided formal proof that queries written in relational calculus and relational algebra are functionally equivalent. [13] In the same paper, he described a scheme where a query language based on relational calculus would be the user interface between an end-user and the database system and the query would be automatically translated into a possible equivalent series of relational algebraic operations that would ultimately be executed. As can be seen above, some relational algebraic queries are more efficient than others while giving the same result. An execution engine would use several optimization techniques to find an optimal strategy. Frank P. Palermo in his 1974 “A Data Base Search Problem” produced some of the earliest attempts at dynamic optimization of the relational calculus to relational algebra retrieval process. [14]

There have been many attempts at realizing the relational model, but SQL is the first and most successful commercially available implementation. It was also designed at the IBM Research Laboratory in San Jose, California, and first introduced in the paper *“SEQUEL: A Structured English Query Language”* by Donald D. Chamberlin and Raymond F. Boyce. [15] It was aimed at giving data accessibility to a class of power users including construction engineers, statisticians, business operation planners, etc. who are willing to spend time to learn a computer language to retrieve the data they need to solve their respective problems but are not necessarily adept at understanding and navigating the various time and memory complexity constraints of a computer system to write an efficient query involving procedural steps. For that reason, SQL is relatively human-readable and possesses a declarative execution model similar to relational calculus largely.

It should be noted that the language diverges from the formal relational model in an important way: a SQL table (relation) is a multi-set, not a set of tuples, and is allowed to have duplicate rows. Also, it incorporates such features as tuple ordering and aggregation (returning a single value for a set of tuples) that are extra-relational but frequently used in data-related tasks.

The SQL language contains three sublanguages catering to three interdependent aspects of data management [16]:

- Data Definition Language (DDL): a higher-order language to $CREATE$, $DROP$, and $ALTER$ relations.
- Data Manipulation Language (DML): the $INSERT$, $UPDATE$ and $DELETE$ statements to manipulate the tuples of the relations.
- Data Query Language (DQL): the $SELECT$ statement, corresponding to the relational calculus described above. This language transforms data into a specified form but does not change it.

Most DQL queries follow this basic block structure:

```
SELECT <list of attributes>
FROM <list of relations>
[JOIN <list of relations to be joined>]
WHERE <list of join and/or filtering conditions>
[ORDER BY <list of attributes to sort the result by>];
```

Listing 4. The basic structure of a SQL query

For example, Listing 3 can be written as such:

```
SELECT P.first, P.last
FROM PA
JOIN P ON PA.p_id = P.id
JOIN A ON PA.a_id = A.id
WHERE A.city = 'Helsinki' AND P.age > 18;
```

Listing 5. Listing 3 rewritten in SQL

In most SQL systems, this query would be translated to a naïve relational algebra query, which would then be rewritten into a more optimized form based on the dependency and/or non-dependence of operation ordering and also on previously collected statistics on the data (for example, certain operations could be omitted if it is known that a table is already pre-sorted or that a column contains the number 1 exclusively). Such an execution scheme is depicted in Figure 9.

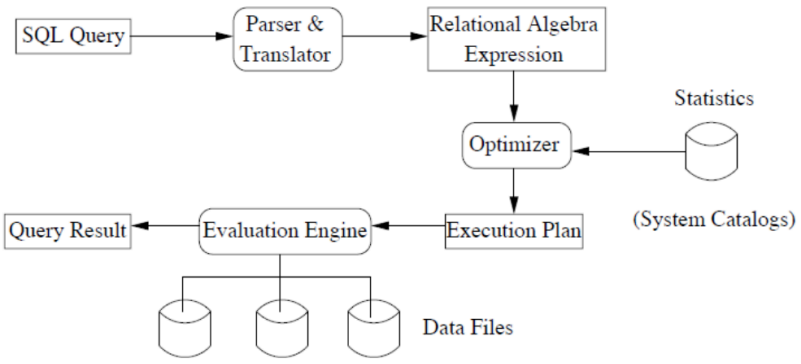

Figure 9. Execution scheme of a SQL query in a modern DBMS

Most SQL engines provide an $EXPLAIN$ command that shows the execution plan in a form similar to the relational algebra graphs in the previous section.

The result of the query is a relation with an arity of 2 containing the first and last names of people older than 18 years old and living in Helsinki. However, this new relation only exists for the duration of the query's consumption inside a temporary object called a cursor. As soon as the cursor is closed or exhausted, the relation is removed from memory. It is, however, possible to define a kind of derived relation based on the result of a query template through a SQL feature called "view". As an example, a $people\_gt\_18yrs\_helsinki$ view is created from the SQL query above:

```
CREATE VIEW people_gt_18yrs_helsinki AS
SELECT P.first, P.last
FROM PA
JOIN P ON PA.p_id = P.id
JOIN A ON PA.a_id = A.id
WHERE A.city = 'Helsinki' AND P.age > 18;
```

Listing 6. Example of a SQL view

The new $people\_gt\_18yrs\_helsinki$ relation now contains the same tuples as the source query and always reflects the latest state of the database when the underlying rows are removed, added, or modified. It should be noted, however, that views are recalculated on the fly every time they are queried. Some DBMS may cache the result of views into persistent memory through a variation called materialized views.

Views allow users to encode institutional knowledge so that multiple clients, be it human operators or application code, demanding the same kind of data do not have to duplicate the same query everywhere it is used. More importantly, views introduce composability into a relational database so that a complex and data-intensive application may push as much knowledge as possible into the database through a directed graph of tables (ground facts) and views (derived facts) that refer to tables and other views. User interfacing UI code would show enriched data by querying the views while data modification happens only to the ground facts and those modifications would be reflected automatically into the derived facts. Figure 10 shows an example of such a graph (the underscored items are ground fact tables).

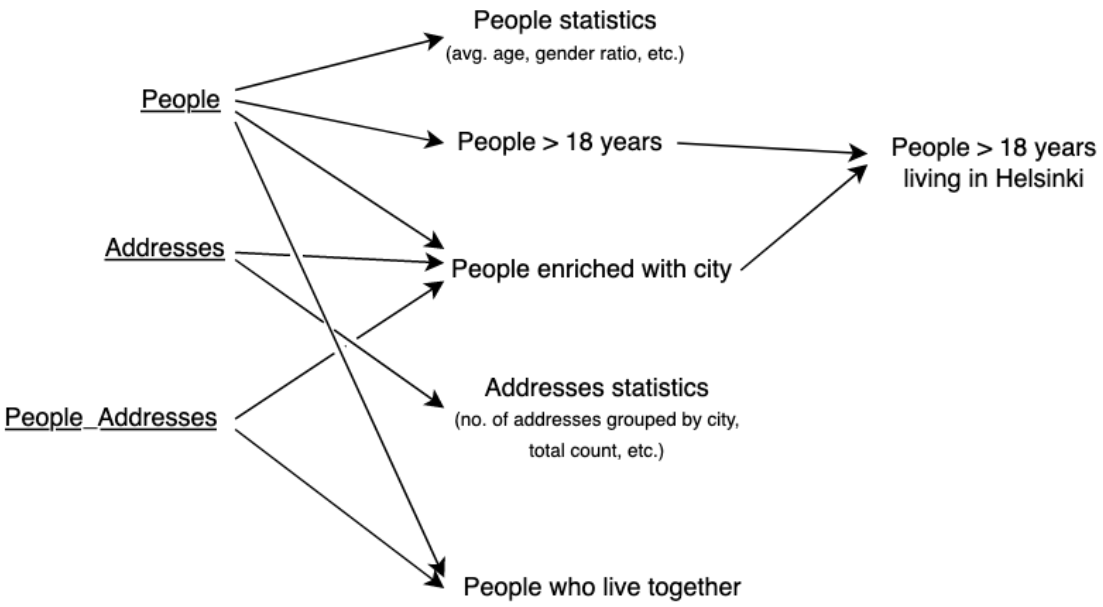

Figure 10. Ground facts and derived facts in a database system

The mobile Messenger app engineering team at Facebook also shares this view of deriving as much fact as possible at the database level in a recent technical blog post. [17]

3 Implementation

The next sections detail an implementation of a wrapper that would add reactivity and subscription to SQL so that a SQL engine would become the heart of the unidirectional data flow model of React Native.

3.1 SQLite in React Native

As of this thesis' publication, there are several competing SQL systems on the market. Notable options include Microsoft SQL Server, Oracle Database, IBM Db2, PostgreSQL, and SQLite. The last one, SQLite, is different from other offerings because it is an embedded database, implemented as a library instead of as a client-server multitenant architecture. The embedding application links against and makes procedural calls to SQLite to read and write to a single file that contains all the data. [18] Because of this architectural design decision, it is attractive as an offline data storage option for client-side applications, especially mobile applications due to the library's small size (a few hundred KB), high performance, and a relatively featureful implementation of the SQL specifications. Thus, SQLite is the natural choice as the backing SQL engine for this project.

To call into SQLite from React Native, the library "react-native-sqlite-2" is used. It is the most actively maintained of such wrappers and used in some commercial projects, including Notes by Mozilla. [19] The library exposes an API that is compatible with W3C's WebSQL specifications. Below is an example of interacting with a SQL database using the API:

```
function prepareDatabase(ready, error) {
  return openDatabase('documents', '1.0', 'Offline document storage',
5*1024*1024, function (db) {
    db.changeVersion('', '1.0', function (t) {
      t.executeSql('CREATE TABLE docids (id, name)');
    }, error);
  });
}

function showDocCount(db, span) {
  db.readTransaction(function (t) {
    t.executeSql('SELECT COUNT(*) AS c FROM docids', [], function (t, r) {
      span.textContent = r.rows[0].c;
    }, function (t, e) {
      // couldn't read database
      span.textContent = '(unknown: ' + e.message + ')';
    });
  });
}

prepareDatabase(function(db) {
  // got database
  var span = document.getElementById('doc-count');
  showDocCount(db, span);
}, function (e) {
  // error getting database
```

```
alert(e.message);
```

Listing 7. Example of writing to and reading from a SQL database through the WebSQL API [20]

As can be seen, the API is imperative, verbose, and does not fit well into the reactive data flow model of React Native.

3.2 Query Controller

To facilitate the reactive flow of data, the system relies on a core module called query controller. Inside the controller, there are two major components: the query cache that executes and caches read-only query statements (DQL) and the data modification handler that processes modification statements (DML).

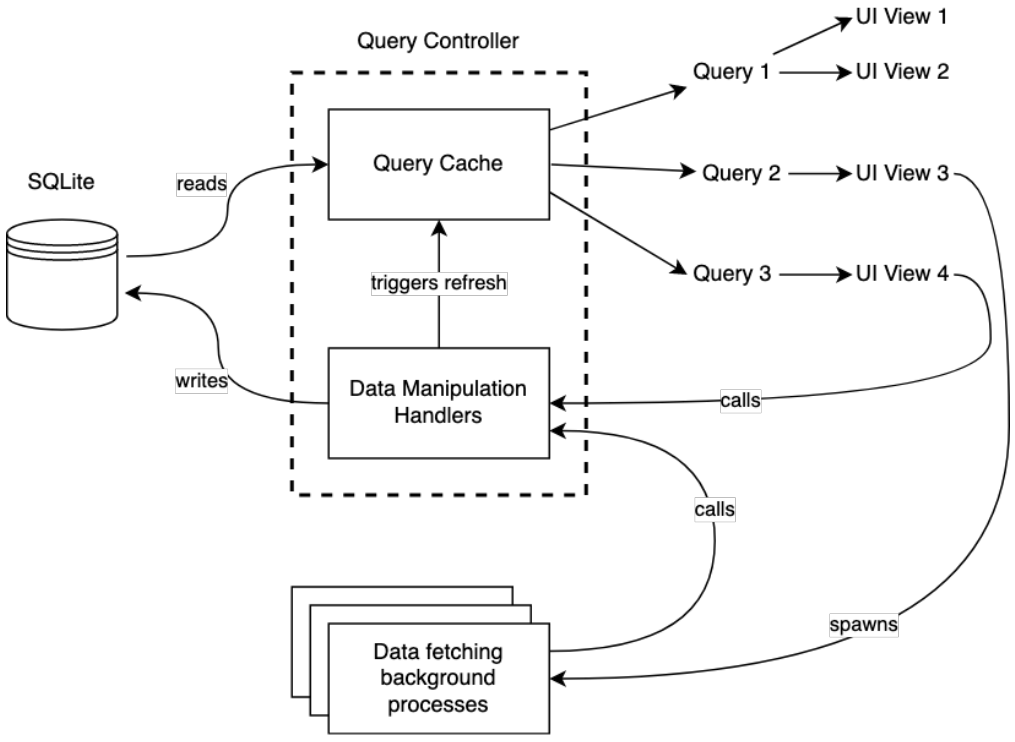

Figure 11. Overview of the query controller system

UI views register queries that are specific to their use case with the controller. They may select only a few relevant columns from the dataset, optionally joined with other tables or SQL views to achieve the desired shape and content. Multiple views could issue the

same query independently (for example UI View 1 and UI View 2 in Figure 11 above), each query is hashed by the controller so duplicated queries are executed only once.

During the execution of the program, the database might be updated, either by direct changes from the user interface through input forms or background data fetching tasks. Nevertheless, all data manipulation must go through the data modification handler. This component forwards the data manipulation statement to the database and after that has been successfully executed, triggers a data refresh at the query cache. The cache itself contains reactive data structures, which, when updated, in turn, trigger a rerender at the view layer, reflecting the latest state in the database, thus completing the unidirectional data flow cycle.

3.3 Relation Dependency Graph

When the data modification handler triggers a refresh, an important task is to decide which queries need to be rerun. A naïve implementation could rerun all active queries, which might work well for smaller databases. However, this approach wastes CPU cycles, draining battery life and potentially introducing unnecessary roundtrip latency to the UI, especially in more complex applications. A more efficient execution technique is to construct a graph of dependency between the queries, views, and tables in the database. This graph flows only in one direction and contains no circular dependency (otherwise, query execution would loop forever). In other words, it is a directed acyclic graph (DAG).

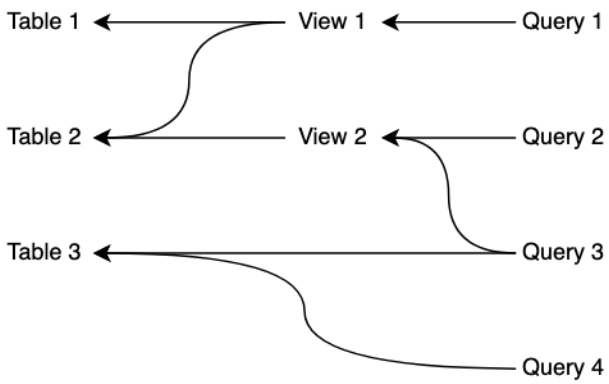

Figure 12. Example of a relation dependency graph

In Figure 12 above, Query 1 depends on View 1, which depends on Table 1; Query 3 depends on View 2 and Table 3, and so on. When Table 1 is updated, only Query 1 should be rerun. When Table 2 is updated, however, Query 1, 2, and 3 should all be rerun.

The dependency graph contains both DDL (tables and views) and DQL objects (queries). Therefore, it has to be constructed in two parts:

- a whole database introspection at the initialization phase of the application to build the links between views and tables; and
- incremental parsing of individual queries as they are registered to the query cache to build the links between the queries to the views and the tables.

The actual parsing is delegated to an existing SQLite language-parsing library called SQLite-parser [21]. This library takes SQLite statements and returns an abstract syntax tree (AST) describing the statements. For example, given this query:

```
SELECT col1, col2 FROM table1 JOIN table2 ON table1.id1 = table2.id2;
```

Listing 8. A simple SQL query with join

The corresponding AST is:

```
{
  "type": "statement",
  "variant": "list",
  "statement": [
      {
        "type": "statement",
        "variant": "select",
        "result": [
            {
              "type": "identifier",
              "variant": "column",
              "name": "col1"
            },
            {
              "type": "identifier",
              "variant": "column",
              "name": "col2"
            }
        ],
        "from": {
            "type": "map",
            "variant": "join",
            "source": {
              "type": "identifier",
              "variant": "table",
              "name": "table1"
            },
            "map": [
```

```
                {
                    "type": "join",
                    "variant": "join",
                    "source": {
                      "type": "identifier",
                      "variant": "table",
                      "name": "table2"
                    },
                    "constraint": {
                      "type": "constraint",
                      "variant": "join",
                      "format": "on",
                      "on": {
                          "type": "expression",
                          "format": "binary",
                          "variant": "operation",
                          "operation": "=",
                          "left": {
                            "type": "identifier",
                            "variant": "column",
                            "name": "table1.id1"
                          },
                          "right": {
                            "type": "identifier",
                            "variant": "column",
                            "name": "table2.id2"
                          }
                      }
                    }
                }
              ]
          }
        }
    ]
}
```

Listing 9. The abstract syntax tree for Listing 8

The relevant parts are the source nodes for $table1$ and $table2$, with identical shapes. A straightforward depth-first pre-order traversal through this tree would extract the 2 nodes in lexical ordering. This works even for subqueries and recursive common table expressions (CTE).

On the DDL side, the schema of the database can be introspected by querying the $sqlite\_master$ table, which has this schema:

```
CREATE TABLE sqlite_schema(
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
);
```

Listing 10. Schema of the $sqlite\_master$ table

The *type* column could have the values *table*, *index*, *view*, or *trigger*. Only the *view* type is of interest because only views can depend on tables or other views. The *sql* column contains the SQL code used to construct the view. This SQL statement can be parsed using the method described above.

Table 1. Example of the *sqlite_master* table from a test database

type	name	tbl_name	rootpage	sql
table	t1	t1	2	`CREATE TABLE t1 ( id integer, col1 text, col2 text )`
table	t2	t2	3	`CREATE TABLE t2 ( id integer, col3 text, col4 text, t1_id integer )`
view	v1	v1	0	`CREATE VIEW v1 as select * from t1 join t2 on t1.id = t2.t1_id`

Once the dependency graph is constructed, any time a DML query is executed, it is parsed to find out which table would be modified as a result, and then the list of affected tables is traced through the dependency graph to determine which query subscriptions should be re-executed, completing the data flow cycle.

3.4 API

With the inner workings of the library described in the sections above, this section gives an overview of its public interface. From the perspective of application developers, the library provides a modular two-layered API.

The core layer is the stateful query controller class. This class has no dependency on ReactJS or React Native so in theory; it could be ported to other UI frameworks or libraries that also use JavaScript/TypeScript like Ionic, NativeScript, etc. without much effort. The class exposes only a few methods for subscribing to queries and to assert data changes. Below is the class' interface with type annotations in TypeScript:

```
type SubscriptionId = string;
type OnDataCallback = (error: Error | null, rows: any[]) => void;

class QueryController {

  // construct the query controller. The only dependency is a
  // WebSQL-compatible database object.
  constructor(db: WebSqlDb);

  // Register a query result subscription. The caller provides a query,
  // a list of parameters to the query, and a callback function that is
  // called with any new data or errors.
  //
  // Returns a subscription ID that can be used to unsubscribe later.
  public subscribe(query: string, args: any[], onData: OnDataCallback):
     SubscriptionId;

  // Unsubscribe a previously registered query result subscription.
  public unsubscribe(subscriptionId: SubscriptionId): void;

  // Run a DML query. One of INSERT, UPDATE, DELETE. The query would
  // potentially trigger a rerun of data subscription queries.
  public runUpdateQuery(query: string, args: any[]): Promise<any>;
}
```

Listing 11. The public interface of the $QueryController$ class

On top of this, there is a second layer that provides a more ergonomic integration with React Native. The second layer uses the context and hook features from ReactJS to adapt the stateful interface of $QueryController$ to the functional programming style used in React components.

ReactJS context is an inversion-of-control dependency-injection container. The developer can inject an object with a name at the root of the component tree and then any child component, no matter how deep down in the tree, can read that object, if they have the correct name. This is similar to a global object but with the benefit, that the child component's dependency on the global object is made explicit as part of that component's public contract so that the child component can be taken apart and tested or used separately with the dependency replaced. [22] The second layer provides a singleton

instance of the query controller that is created by application developers with their own $WebSqlDb$ instance. Any child component could then access this query controller object.

Secondly, ReactJS provides a feature called hook to reconcile stateful behaviors with the functional $render()$ tree. In theory, ReactJS' programming model is that $render()$ is called whenever there is a change in data and the new tree would replace the old tree. But in reality, the component tree is preserved and survives through several invocations to $render()$. Therefore, individual components in the tree exhibit stateful behaviors: they go through lifecycle events like mounting and unmounting and they may have private internal states (e.g., the textual value of an input field). ReactJS provides hooks: special functions which when called would allow to "hook" into the stateful lifecycle of the underlying component. [23]

There are two kinds of hooks that are of interest to this project, first is the state hook. It is a function that returns an array of two values, the first of which is the current value of the state variable, the second of which is a function that takes one argument that changes the state to a new value. When the state is updated, the React component subtree where the current component is the root is re-rendered to show the latest state.

```
const [state, setState] = React.useState(initialValue);
```

Listing 12. Example of the ReactJS state hook

The state hook is commonly used to store the current state of input fields or to store the result of an external HTTP data fetch. In our case, the state hook is used to store the immediate result of the query subscription.

The second kind of hook used is the effect hook. It takes a function and a list of values that the function depends on and calls the function the first time when the component is mounted and whenever any of the dependent values is changed. The callback function returns a clean-up function, which is called when the component is unmounted from the tree. Here is an example of using the effect hook together with the state hook to fetch external data and clean-up on unmount:

```
React.useEffect(function callback() {
  setResultState(await fetchExternalData(dep1, dep2));
```

```
  return function cleanUp() {
    console.log("the clean-up function is called");
  }
}, [dep1, dep2]);
```

Listing 13. Example of the ReactJS effect hook

Hooks are composable. It is possible to define a custom function that calls other hooks and return results from the hooks. The new function itself becomes a hook, usable in any React component. As such, hooks are an excellent way to encapsulate state and behaviors, similar to the concept of object composition in object-oriented programming, highly preferable over inheritance. [24]

The React Native integration layer is thus a simple hook that takes a SQL query string with a list of query arguments. It then uses the effect hook to immediately create and run a stateful query subscription through the query controller, taken from the ReactJS context. The query subscription callback continuously stores the query data in a result state, created from the state hook. On component unmount, the effect hook's clean-up function calls the $unsubscribe()$ method from the query controller. The query hook is wrapped inside the custom $useReactiveSqlite()$ hook that bundles the query hook and the update method of the query controller together:

```
function useReactiveSqlite() {
  const queryController = useContext(ReactiveSqliteContext);

  function useQuery(query: string, args: any[] = []) {
    let [result, setResult] = useState<any[]>([]);
    let [error, setError] = useState<Error|null>(null);

    const updateData = useCallback(function (error, data) {
      if (error) {
        setError(error)
      } else {
        setResult(data);
      }
    }, []);

    useEffect(function () {
      let subId = queryController!.subscribe(query, args, updateData);

      return function cleanup() {
        queryController!.unsubscribe(subId);
      }
    }, [query]);

    return {result, error};
  }
```

```
  return {
    queryController: queryController,
    query: useQuery,
    update: queryController!.runUpdateQuery.bind(queryController)
  }
}
```

Listing 14. Source code for the $useReactiveSqlite()$ custom hook

From the application programmer's perspective, the whole lifecycle management of query subscription is abstracted into just a single declarative function call (the embolden line) returning a value (an array of rows) that automatically updates as data changes:

```
function MainComponent() {
  let {query, update} = useReactiveSqlite();

  const queryResult = query("SELECT * FROM t3");

  function updateData() {
    update(
        "INSERT INTO t3 (id, col5) values (5, 'a sample string value')", []);
  }

  return (
      <SafeAreaView>
        <StatusBar barStyle={"light-content"}/>
        <ScrollView
            contentInsetAdjustmentBehavior="automatic">

          <Text>
            {JSON.stringify(queryResult)}
          </Text>

          <Button onPress={updateData} title={"Update Data"}>

          </Button>
        </ScrollView>
      </SafeAreaView>
  )
}
```

Listing 15. Example of subscribing to a SQL query and running an INSERT statement

This minimal interface arguably achieves the goal of facilitating the ease of use of SQL queries in React Native applications in a reactive and declarative style.

It is outside the project's scope to provide a system for managing SQL queries. But application developers are encouraged to take advantage of several existing libraries and patterns, including Knex.js [25] and Squel.js [26] that allow programmatically constructing the queries, or yesql [27], which allows organizing queries into files and folders.

4 Results

4.1 Benchmarks

To assess the approach's efficiency, a preliminary benchmark was performed. A series of test cases were run that insert random rows into the database with increasing numbers of rows. A performance counter is then consulted to measure the elapsed time between the moment right before the data was inserted and the moment right after the last rendering that occurred because of the data change was finalized. The finding is shown in Table 2.

Table 2. Average rendering round-trip latency of consecutive insert operations

Rows count \ Sample No.	1	2	3	4	5	Average (ms)
100	1746	1748	1730	1743	1744	1742,2
200	3481	3448	3453	3470	3473	3465,0
400	6946	7004	7064	6861	6941	6963,2
800	14529	14915	14942	14942	15111	14887,8
1600	31078	31405	31499	30931	31664	31315,4

When plotted, a strong linear pattern can be identified from the data.

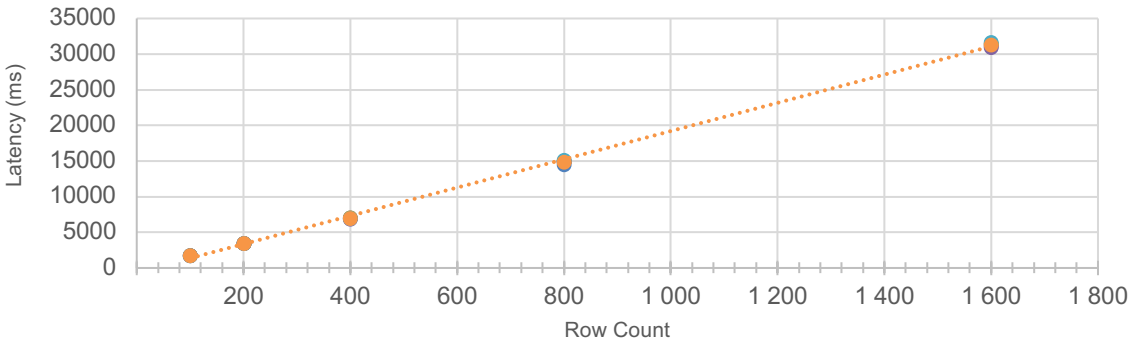

Figure 13. Round-trip latency of write operations by row count

On average, it takes approximately 17ms for a single data update operation to fully go through the cycle. This number is close to the 16,67ms time budget to render a single

graphical frame on a device with a screen operating at 60fps so there would be no noticeable delay to the end-user. See Figure 13 above.

The data is further encouraging because it shows that, asymptotically, the approach is efficient with a time complexity of$O(n)$. It is acknowledged, however, that the project is still in its infancy and a few opportunities for optimization could be identified.

One area to improve is re-render cascading. In the current implementation, if 1000 similar write operations are executed consecutively then the subscription handler would be called 1000 times although not all of them are necessary. Perhaps there could be a subscription rerun queue that throttles to at most once per animation frame and drops the stale reruns.

Another issue is that the update query runner executes every write operation inside its own transaction. This is problematic in the case where an action from the user triggers several writes to different database tables or multiple writes to the same table. It would be more efficient to provide an alternative API that batches multiple related writes together in a single transaction.

Lastly, the dependency analysis is performed for every write operation even though the query template is known statically. A possible remedy is to analyze the write query only once when it is first encountered and use the query template itself as a hash key to retrieve the dependency calculation result for subsequent calls.

With the proper optimizations in place, a sub-10ms or even sub-5ms average round-trip latency should be achievable.

4.2 Comparison with Existing Options in React Native

The major alternatives for managing data in a React Native application mostly fall into two categories: local only data flow containers, including Redux, MobX, and this thesis' reactive SQLite wrapper; the other category includes server-side database services that have React Native-compatible client that synchronizes and caches data locally, including

Google's Firebase, MongoDB's Realm, Apache CouchDB/PouchDB pairing and Facebook's GraphQL (technically only an RPC-level query transfer specification with multiple possible commercial backends, including Apollo Data Graph Platform and Amazon AWS' AppSync). The reactive SQLite wrapper is intended to compete with Redux and MobX and not with offerings in the second category. Nevertheless, Table 3 compares them all against each other on some important qualitative criteria regarding usage in a React Native project to give a better overview.

Table 3. Comparison of major data management alternatives in React Native

	Redux	**MobX**	**Reactive SQLite**	**Firebase**	**Realm**	**PouchDB**	**GraphQL**
Sync with external network-connected server	No	No	No	Yes	Yes	In combination with CouchDB	Possible with some backend implementations
Offline persistence	Add-on available	Add-on available	Built-in	Built-in	Built-in	Built-in	Add-on available
Rich data enrichment support	No	No	Yes	Limited	Limited	Limited	Yes (need extensive server-side support)
Latency	Low	Low	Acceptable	Network dependent	Network dependent	Network dependent	Network dependent

Ease of use with React Native	Verbose and require a heavy initial setup	Low friction	Low friction and familiar with many developers	Heavy initial setup but low friction after that	Low friction	Low friction	Low friction
Popularity / Community support	Very high	High	Very Low	Low	High	Low	High

Although there are some overlaps, this method combines attractive aspects of the two camps: being both local-only and supporting a rich query language. It shines when combining multiple data sources like REST APIs on the client side. In some cases, it might be superior to the database synchronizing solutions since they are optimized for the mass distributed computing use case, they have to make compromises in the query language and many rich querying features are not supported.

5 Conclusion

The query subscription model significantly simplifies the design of complex and data-driven user interfaces. This thesis explored adding query subscription to a mobile application using React Native. Additionally, the library was structured in such a way that there is a non-React specific core that could be applied to other mobile or cross-platform JavaScript UI frameworks. Similarly, front-end web applications written with plain React could also make use of the reactive SQL engine when combined with SQL emulation in the browser, using, for example, NanoSQL [28], a library that aims to provide a simple SQL language execution engine for the browser environment. On the other side of the backend-frontend divide, the technique could also be ported to server-side programming in NodeJS as a base for backend web soft real-time applications where data is streamed over WebSocket or Server-sent Events to the client.

References

[1] J. Clement, "Share of mobile internet traffic in global regions 2021," 12 April 2021. [Online]. Available: https://www.statista.com/statistics/306528/share-of-mobile-internet-traffic-in-global-regions/. [Accessed 5 May 2021].

[2] Statista Research Department, " Worldwide mobile app revenues in 2014 to 2023," 11 March 2021. [Online]. Available: https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/. [Accessed 5 May 2021].

[3] A. Mazzina, "Developer Hiring Trends in 2017," 9 March 2017. [Online]. Available: https://stackoverflow.blog/2017/03/09/developer-hiring-trends-2017/.

[4] S. Liu, "Cross-platform mobile frameworks used by software developers worldwide in 2019 and 2020," June 2020. [Online]. Available: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. [Accessed 15 April 2021].

[5] Stack Overflow, "Stack Overflow 2020 Developer Survey," Stack Overflow, 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020. [Accessed 12 April 2021].

[6] T. Occhino and J. Walke, "JS Apps at Facebook," 5 Aug 2013. [Online]. Available: https://www.youtube.com/watch?v=GW0rj4sNH2w. [Accessed 15 April 2021].

[7] Facebook Inc., "Reconciliation – React," [Online]. Available: https://reactjs.org/docs/reconciliation.html. [Accessed 15 April 2021].

[8] P. Hunt, P. O'Shannessy and T. Coatta, "React: Facebook's Functional Turn on Writing JavaScript," *ACM Queue,* pp. 1-17, July-August 2016.

[9] Facebook Inc., "React Native," Facebook Inc., [Online]. Available: https://reactnative.dev/. [Accessed 15 April 2021].

[10] D. Abramov, "Core Concepts – Redux," [Online]. Available: https://redux.js.org/introduction/core-concepts. [Accessed 15 April 2021].

[11] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM,* no. June, p. 377–387, 1970.

[12] T. Ho and B. V. Le, "Some results about keys of relational schemas," *Acta Cybern,* vol. 7, no. 1, pp. 99-113, 1985.

[13] E. F. Codd, "A data base sublanguage founded on the relational calculus," in *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, Association for Computing Machinery, 1971, p. 35–68.

[14] F. P. Palermo, "A Data Base Search Problem," in *Information Systems*, Springer, 1974.

[15] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A Structured English Query Language," in *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, Association for Computing Machinery, 1974, p. 249–264.

[16] International Organization for Standardization, ISO/IEC 9075-1:2016(en) Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework), International Organization for Standardization, 2016.

[17] M. Agsen, "Project LightSpeed: Rewriting the Messenger codebase for a faster, smaller, and simpler messaging app," 2 March 2020. [Online]. Available: https://engineering.fb.com/2020/03/02/data-infrastructure/messenger/. [Accessed 10 April 2021].

[18] D. R. Hipp, "SQLite," [Online]. Available: https://sqlite.org/index.html. [Accessed 15 April 2021].

[19] T. Matsuyama, "react-native-sqlite-2," 2020. [Online]. Available: https://github.com/craftzdog/react-native-sqlite-2. [Accessed 15 April 2021].

[20] I. Hickson, "Web SQL Database," W3C Working Group, 18 November 2010. [Online]. Available: https://www.w3.org/TR/webdatabase/. [Accessed 15 April 2021].

[21] N. Wronski, "sqlite-parser - JavaScript implentation of SQLite 3 query parser," 2017. [Online]. Available: https://github.com/codeschool/sqlite-parser. [Accessed 14 April 2021].

[22] Facebook Inc., "Context – React," [Online]. Available: https://reactjs.org/docs/context.html. [Accessed 15 April 2021].

[23] Facebook Inc., "Hooks at a Glance – React," [Online]. Available: https://reactjs.org/docs/hooks-overview.html. [Accessed 15 April 2021].

[24] E. Freeman, E. Robson, K. Sierra and B. Bates, "Head First Design Patterns," O'Reilly, 2004, p. 23.

[25] I. Savin, "Knex.js - A SQL Query Builder for Javascript," [Online]. Available: http://knexjs.org/. [Accessed 15 April 2021].

[26] R. Nair, "Squel.js - SQL query string builder for Javascript," [Online]. Available: https://hiddentao.github.io/squel/v4/index.html. [Accessed 15 April 2021].

[27] A. Mattila, "yesql," 2020. [Online]. Available: https://github.com/pihvi/yesql. [Accessed 15 April 2021].

[28] S. Lott, "nanoSQL 2," 19 May 2019. [Online]. Available: https://nanosql.io/. [Accessed 14 April 2021].