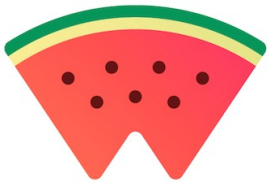


Watermelon DB

A reactive database framework

Build powerful React and React Native apps that scale from hundreds to tens of thousands of records and remain *fast* ⚡

	WatermelonDB
⚡	**Launch your app instantly** no matter how much data you have
📈	**Highly scalable** from hundreds to tens of thousands of records
😎	**Lazy loaded**. Only load data when you need it
🔄	**Offline-first.** Sync with your own backend
📱	**Multiplatform**. iOS, Android, Windows, web, and Node.js
⚛️	**Optimized for React.** Easily plug data into components
🧰	**Framework-agnostic.** Use JS API to plug into other UI frameworks
⏱	**Fast.** And getting faster with every release!
✅	**Proven.** Powers Nozbe Teams since 2017 (and many others)
✨	**Reactive.** (Optional) RxJS API
🔗	**Relational.** Built on rock-solid SQLite foundation
⚠️	**Static typing** with Flow or TypeScript

Why Watermelon?

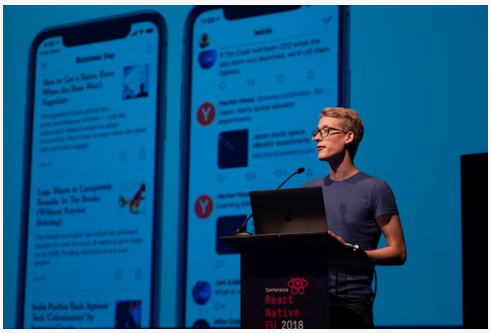
WatermelonDB is a new way of dealing with user data in React Native and React web apps.

It's optimized for building **complex applications** in React Native, and the number one goal is **real-world performance**. In simple words, *your app must launch fast*.

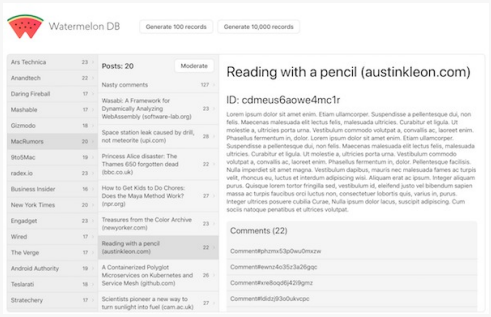
For simple apps, using Redux or MobX with a persistence adapter is the easiest way to go. But when you start scaling to thousands or tens of thousands of database records, your app will now be slow to launch (especially on slower Android devices). Loading a full database into JavaScript is expensive!

Watermelon fixes it **by being lazy**. Nothing is loaded until it's requested. And since all querying is performed directly on the rock-solid SQLite database on a separate native thread, most queries resolve in an instant.

But unlike using SQLite directly, Watermelon is **fully observable**. So whenever you change a record, all UI that depends on it will automatically re-render. For example, completing a task in a to-do app will re-render the task component, the list (to reorder), and all relevant task counters. **Learn more**.



Next-generation React databases
(a talk about WatermelonDB)



Check out web Demo

Usage

Quick (over-simplified) example: an app with posts and comments.

First, you define Models:

```
class Post extends Model {
  @field('name') name
  @field('body') body
  @children('comments') comments
}

class Comment extends Model {
  @field('body') body
  @field('author') author
}
```

Then, you connect components to the data:

```
const Comment = ({ comment }) => (
  <View style={styles.commentBox}>
    <Text>{comment.body} — by {comment.author}</Text>
  </View>
)

// This is how you make your app reactive! ✨
const enhance = withObservables(['comment'], ({ comment }) => ({
  comment,
}))
const EnhancedComment = enhance(Comment)
```

And now you can render the whole Post:

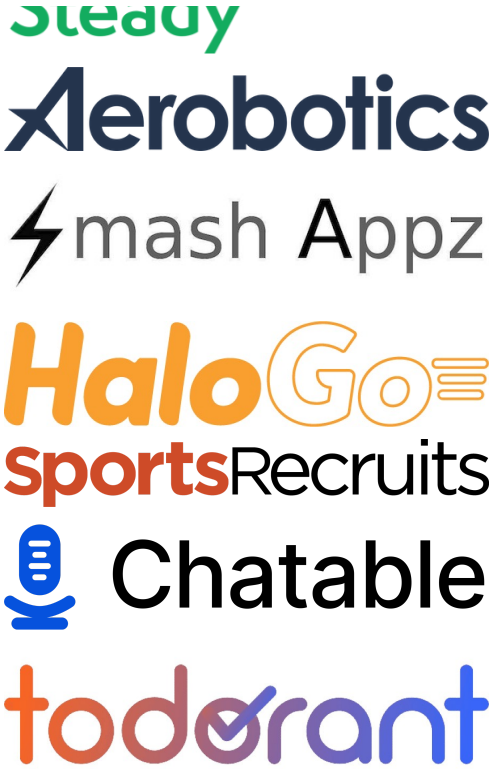
```
const Post = ({ post, comments }) => (
  <View>
    <Text>{post.name}</Text>
    <Text>Comments:</Text>
    {comments.map(comment =>
      <EnhancedComment key={comment.id} comment={comment} />
    )}
  </View>
)

const enhance = withObservables(['post'], ({ post }) => ({
  post,
  comments: post.comments
}))
```

The result is fully reactive! Whenever a post or comment is added, changed, or removed, the right components **will automatically re-render** on screen. Doesn't matter if a change occurred in a totally different part of the app, it all just works out of the box!

➡️ Learn more: see full documentation

Who uses WatermelonDB



Does your company or app use 🍉? Open a pull request and add your logo/icon with link here!

Contributing

WatermelonDB is an open-source project and it needs your help to thrive!

If there's a missing feature, a bug, or other improvement you'd like, we encourage you to contribute! Feel free to open an issue to get some guidance and see Contributing guide for details about project setup, testing, etc.

If you're just getting started, see good first issues that are easy to contribute to. If you make a non-trivial contribution, email me, and I'll send you a nice 🍉 sticker!

If you make or are considering making an app using WatermelonDB, please let us know!

Author and license

WatermelonDB was created by @Nozbe.

WatermelonDB's main author and maintainer is Radek Pietruszewski (website · twitter · engineering posters)

See all contributors.

Edit this page

Next
Installation »

Installation

First, add Watermelon to your project:

```
yarn add @nozbe/watermelondb

# (or with npm:)
npm install @nozbe/watermelondb
```

React Native setup

1. Install the Babel plugin for decorators if you haven't already:

   ```
   yarn add --dev @babel/plugin-proposal-decorators

   # (or with npm:)
   npm install -D @babel/plugin-proposal-decorators
   ```

2. Add ES6 decorators support to your `.babelrc` file:

   ```
   {
     "presets": ["module:metro-react-native-babel-preset"],
     "plugins": [["@babel/plugin-proposal-decorators", { "legacy": true }]]
   }
   ```

3. Set up your iOS or Android project — see instructions below

iOS (React Native)

At least Xcode 13.x and iOS 15 are recommended (earlier versions are not tested for compatibility).

1. **Set up Babel config in your project**

   See instructions above ⬆

2. **Link WatermelonDB's native library (using CocoaPods)**

   Open your `Podfile` and add this:

```
# Uncomment this line if you're not using auto-linking or if auto-linking causes trouble
# pod 'WatermelonDB', path: '../node_modules/@nozbe/watermelondb'

# WatermelonDB dependency, should not be needed on modern React Native
# (please file an issue if this causes issues for you)
# pod 'React-jsi', path: '../node_modules/react-native/ReactCommon/jsi', modular_headers: tr

# WatermelonDB dependency
pod 'simdjson', path: '../node_modules/@nozbe/simdjson', modular_headers: true
```

Make sure you run `pod install` (or `bundle exec pod install`) after updating `Podfile`.

We highly recommend that you *do not* use frameworks. If WatermelonDB fails to build in the frameworks mode for you, use this workaround to force building it in static library mode.

Manual (non-CocoaPods) linking is not supported.

Android (React Native)

Set up Babel config in your project

See instructions above ⬆️

<details>
<summary>Linking Manually</summary>
</details>

<details>
<summary>Using with react-native-screens or react-native-gesture-handler</summary>
</details>

<details>
<summary>Troubleshooting</summary>
</details>

<details>
<summary>JSI Installation (Optional, recommended)</summary>
</details>

Web setup

This guide assumes you use Webpack as your bundler.

1. If you haven't already, install Babel plugins for decorators, static class properties, and async/await to get the most out of Watermelon. This assumes you use Babel 7 and already support ES6 syntax.

```
yarn add --dev @babel/plugin-proposal-decorators
yarn add --dev @babel/plugin-proposal-class-properties
yarn add --dev @babel/plugin-transform-runtime

# (or with npm:)
npm install -D @babel/plugin-proposal-decorators
npm install -D @babel/plugin-proposal-class-properties
```

```
npm install -D @babel/plugin-transform-runtime
```

2. Add ES7 support to your `.babelrc` file:

```
{
  "plugins": [
    ["@babel/plugin-proposal-decorators", { "legacy": true }],
    ["@babel/plugin-proposal-class-properties", { "loose": true }],
    [
      "@babel/plugin-transform-runtime",
      {
        "helpers": true,
        "regenerator": true
      }
    ]
  ]
}
```

Windows (React Native)

WatermelonDB has **experimental** support for React Native Windows.

To set up:

1. Set up Babel config in your project - See instructions above for all React Native platforms
2. Run `npx react-native autolink-windows` to perform autolinking. See section below if you don't use autolinking.

Caveats to keep in mind about React Native Windows support:

- Windows support is new and experimental
- Only JSI port is available, so you must initialize `SQLiteAdapter` with `{ jsi: true }`
- JSI means that Remote Debugging (WebDebugger) is not available. Use direct debugging.
- Enable Hermes when using WatermelonDB on RNW. Chakra has not been tested and may not work.
- Turbo Sync has not been implemented
- onDestroy event has not been implemented. This only causes issues if you need to reload JS bundle at runtime (other than in development).

▸ Linking Manually

NodeJS (SQLite) setup

You only need this if you want to use WatermelonDB in NodeJS with SQLite (e.g. for scripts that share code with your web/React Native app)

1. Install better-sqlite3 peer dependency

```
yarn add --dev better-sqlite3

# (or with npm:)
npm install -D better-sqlite3
```

Next steps

➡️ After Watermelon is installed, **set it up**

Edit this page

Previous
« Check out the README

Next
Setup »

Schema

When using WatermelonDB, you're dealing with **Models** and **Collections**. However, underneath Watermelon sits an **underlying database** (SQLite or LokiJS) which speaks a different language: **tables and columns**. Together, those are called a **database schema** and we must define it first.

Defining a Schema

Say you want Models `Post`, `Comment` in your app. For each of those Models, you define a table. And for every field of a Model (e.g. name of the blog post, author of the comment) you define a column. For example:

```
// model/schema.js
import { appSchema, tableSchema } from '@nozbe/watermelondb'

export const mySchema = appSchema({
  version: 1,
  tables: [
    tableSchema({
      name: 'posts',
      columns: [
        { name: 'title', type: 'string' },
        { name: 'subtitle', type: 'string', isOptional: true },
        { name: 'body', type: 'string' },
        { name: 'is_pinned', type: 'boolean' },
      ]
    }),
    tableSchema({
      name: 'comments',
      columns: [
        { name: 'body', type: 'string' },
        { name: 'post_id', type: 'string', isIndexed: true },
      ]
    }),
  ]
})
```

Note: It is database convention to use plural and snake_case names for table names. Column names are also snake_case. So `Post` become `posts` and `createdAt` becomes `created_at`.

Column types

Columns have one of three types: `string`, `number`, or `boolean`.

Fields of those types will default to `''`, `0`, or `false` respectively, if you create a record with a missing field.

To allow fields to be `null`, mark the column as `isOptional: true`.

Naming conventions

To add a relation to a table (e.g. `Post` where a `Comment` was published, or author of a comment), add a string column ending with `_id`:

```
{ name: 'post_id', type: 'string' },
{ name: 'author_id', type: 'string' },
```

Boolean columns should have names starting with `is_`:

```
{ name: 'is_pinned', type: 'boolean' }
```

Date fields should be `number` (dates are stored as Unix timestamps) and have names ending with `_at`:

```
{ name: 'last_seen_at', type: 'number', isOptional: true }
```

Special columns

All tables *automatically* have a string column `id` (of `string` type) to uniquely identify records -- therefore you cannot declare a column named `id` yourself. (There are also special `_status` and `_changed` columns used for synchronization - you shouldn't touch them yourself).

You can add special `created_at` / `updated_at` columns to enable automatic create/update tracking.

Modifying Schema

Watermelon cannot automatically detect Schema changes. Therefore, whenever you change the Schema, you must increment its version number (`version:` field).

During early development, this is all you need to do - on app reload, this will cause the database to be cleared completely.

To seamlessly update the schema (without deleting user data), use Migrations.

⚠️ Always use Migrations if you already shipped your app.

Indexing

To enable database indexing, add `isIndexed: true` to a column.

Indexing makes querying by a column faster, at the expense of create/update speed and database size.

For example, if you often query all comments belonging to a post (that is, query comments by its `post_id` column), you should mark the `post_id` column as indexed.

However, if you rarely query all comments by its author, indexing `author_id` is probably not worth it.

In general, most `_id` fields are indexed. Occasionally, `boolean` fields are worth indexing (but it's a "low quality index"). However, you should almost never index date (`_at`) columns or `string` columns. You definitely do not want to index long-form user text.

⚠️ Do not mark all columns as indexed to "make Watermelon faster". Indexing has a real performance cost and should be used only when appropriate.

Advanced

Unsafe SQL schema

If you want to modify the SQL used to set up the SQLite database, you can pass `unsafeSql` parameter to `tableSchema` and `appSchema`. This parameter is a function that receives SQL generated by Watermelon, and you can return whatever you want - so you can append, prepend, replace parts of SQL, or return your own SQL altogether. When passed to `tableSchema`, it receives SQL generated for just that table, and when to `appSchema` - the entire schema SQL.

⚠️ Note that SQL generated by WatermelonDB is not considered to be a stable API, so be careful about your transforms as they can break at any time.

```
appSchema({
  ...
  tables: [
    tableSchema({
      name: 'tasks',
      columns: [...],
      unsafeSql: sql => sql.replace(/create table [^)]+\)/, '$& without rowid'),
    }),
  ],
  unsafeSql: (sql, kind) => {
    // Note that this function is called not just when first setting up the database
    // Additionally, when running very large batches, all database indices may be dropped and la
    // recreated as an optimization. More kinds may be added in the future.
    switch (kind) {
      case 'setup':
        return `create blabla;${sql}`
      case 'create_indices':
      case 'drop_indices':
        return sql
      default:
        throw new Error('unexpected unsafeSql kind')
    }
  },
})
```

Next steps

➡️ After you define your schema, go ahead and **define your Models**

Edit this page

Previous
« Setup

Next
Model »

Model

A **Model** class represents a type of thing in your app. For example, `Post`, `Comment`, `User`.

Before defining a Model, make sure you defined its schema.

Create a Model

Let's define the `Post` model:

```
// model/Post.js
import { Model } from '@nozbe/watermelondb'

export default class Post extends Model {
  static table = 'posts'
}
```

Specify the table name for this Model — the same you defined in the schema.

Now add the new Model to `Database`:

```
// index.js
import Post from 'model/Post'

const database = new Database({
  // ...
  modelClasses: [Post],
})
```

Associations

Many models relate to one another. A `Post` has many `Comment`s. And every `Comment` belongs to a `Post`. (Every relation is double-sided). Define those associations like so:

```
class Post extends Model {
  static table = 'posts'
  static associations = {
    comments: { type: 'has_many', foreignKey: 'post_id' },
  }
}
```

```
class Comment extends Model {
  static table = 'comments'
  static associations = {
    posts: { type: 'belongs_to', key: 'post_id' },
  }
}
```

On the "child" side (`comments`) you define a `belongs_to` association, and pass a column name (key) that points to the parent (`post_id` is the ID of the post the comment belongs to).

On the "parent" side (`posts`) you define an equivalent `has_many` association and pass the same column name (⚠ note that the name here is `foreignKey`).

Add fields

Next, define the Model's *fields* (properties). Those correspond to table columns defined earlier in the schema.

```
import { field, text } from '@nozbe/watermelondb/decorators'

class Post extends Model {
  static table = 'posts'
  static associations = {
    comments: { type: 'has_many', foreignKey: 'post_id' },
  }

  @text('title') title
  @text('body') body
  @field('is_pinned') isPinned
}
```

Fields are defined using ES6 decorators. Pass **column name** you defined in Schema as the argument to `@field`.

Field types. Fields are guaranteed to be the same type (string/number/boolean) as the column type defined in Schema. If column is marked `isOptional: true`, fields may also be null.

User text fields. For fields that contain arbitrary text specified by the user (e.g. names, titles, comment bodies), use `@text` - a simple extension of `@field` that also trims whitespace.

Note: Why do I have to type the field/column name twice? The database convention is to use `snake_case` for names, and the JavaScript convention is to use camelCase. So for any multi-word name, the two differ. Also, for resiliency, we believe it's better to be explicit, because over time, you might want to refactor how you name your JavaScript field names, but column names must stay the same for backward compatibility.

Date fields

For date fields, use `@date` instead of `@field`. This will return a JavaScript `Date` object (instead of Unix timestamp integer).

```
import { date } from '@nozbe/watermelondb/decorators'

class Post extends Model {
  // ...
  @date('last_event_at') lastEventAt
}
```

Derived fields

Use ES6 getters to define model properties that can be calculated based on database fields:

```
import { field, text } from '@nozbe/watermelondb/decorators'

class Post extends Model {
  static table = 'posts'

  @date('archived_at') archivedAt

  get isRecentlyArchived() {
    // in the last 7 days
    return this.archivedAt &&
      this.archivedAt.getTime() > Date.now() - 7 * 24 * 3600 * 1000
  }
}
```

To-one relation fields

To point to a related record, e.g. `Post` a `Comment` belongs to, or author (`User`) of a `Comment`, use `@relation` or `@immutableRelation`:

```
import { relation, immutableRelation } from '@nozbe/watermelondb/decorators'

class Comment extends Model {
  // ...
  @relation('posts', 'post_id') post
  @immutableRelation('users', 'author_id') author
}
```

➡️ **Learn more:** Relation API

Children (to-many relation fields)

To point to a list of records that belong to this Model, e.g. all `Comment`s that belong to a `Post`, you can define a simple `Query` using `@children`:

```
import { children } from '@nozbe/watermelondb/decorators'

class Post extends Model {
  static table = 'posts'
  static associations = {
    comments: { type: 'has_many', foreignKey: 'post_id' },
  }
```

```
  @children('comments') comments
}
```

Pass the *table name* of the related records as an argument to `@children`. The resulting property will be a `Query` you can fetch, observe, or count.

Note: You must define a `has_many` association in `static associations` for this to work

➡️ **Learn more:** Queries

Custom Queries

In addition to `@children`, you can define custom Queries or extend existing ones, for example:

```
import { children } from '@nozbe/watermelondb/decorators'
import { Q } from '@nozbe/watermelondb'

class Post extends Model {
  static table = 'posts'
  static associations = {
    comments: { type: 'has_many', foreignKey: 'post_id' },
  }

  @children('comments') comments
  @lazy verifiedComments = this.comments.extend(
    Q.where('is_verified', true)
  )
}
```

➡️ **Learn more:** Queries

Writer methods

Define **writers** to simplify creating and updating records, for example:

```
import { writer } from '@nozbe/watermelondb/decorators'

class Comment extends Model {
  static table = 'comments'

  @field('is_spam') isSpam

  @writer async markAsSpam() {
    await this.update(comment => {
      comment.isSpam = true
    })
  }
}
```

Methods must be marked as `@writer` to be able to modify the database.

➡️ **Learn more:** Writers

Advanced fields

You can also use these decorators:

- `@json` for complex serialized data
- `@readonly` to make the field read-only
- `@nochange` to disallow changes to the field *after the first creation*

And you can make observable compound properties using RxJS...

➡️ **Learn more:** Advanced fields

Next steps

➡️ After you define some Models, learn the **Create / Read / Update / Delete API**

Edit this page

Previous
« Schema

Next
Migrations »

Migrations

Schema migrations is the mechanism by which you can add new tables and columns to the database in a backward-compatible way.

Without migrations, if a user of your app upgrades from one version to another, their local database will be cleared at launch, and they will lose all their data.

⚠️ Always use migrations!

Migrations setup

1. Add a new file for migrations:

```
// app/model/migrations.js

import { schemaMigrations } from '@nozbe/watermelondb/Schema/migrations'

export default schemaMigrations({
  migrations: [
    // We'll add migration definitions here later
  ],
})
```

2. Hook up migrations to the Database adapter setup:

```
// index.js
import migrations from 'model/migrations'

const adapter = new SQLiteAdapter({
  schema: mySchema,
  migrations,
})
```

Migrations workflow

When you make schema changes when you use migrations, be sure to do this in this specific order, to minimize the likelihood of making an error.

Step 1: Add a new migration

First, define the migration - that is, define the **change** that occurs between two versions of schema (such as adding a new table, or a new table column).

Don't change the schema file yet!

```
// app/model/migrations.js

import { schemaMigrations, createTable } from '@nozbe/watermelondb/Schema/migrations'

export default schemaMigrations({
  migrations: [
    {
      // ⚠️ Set this to a number one larger than the current schema version
      toVersion: 2,
      steps: [
        // See "Migrations API" for more details
        createTable({
          name: 'comments',
          columns: [
            { name: 'post_id', type: 'string', isIndexed: true },
            { name: 'body', type: 'string' },
          ],
        }),
      ],
    },
  ],
})
```

Refresh your simulator/browser. You should see this error:

> Migrations can't be newer than schema. Schema is version 1 and migrations cover range from 1 to 2

If so, good, move to the next step!

But you might also see an error like "Missing table name in schema", which means you made an error in defining migrations. See "Migrations API" below for details.

Step 2: Make matching changes in schema

Now it's time to make the actual changes to the schema file — add the same tables or columns as in your migration definition

⚠️ Please double and triple check that your changes to schema match exactly the change you defined in the migration. Otherwise you risk that the app will work when the user migrates, but will fail if it's a fresh install — or vice versa.

⚠️ Don't change the schema version yet

```
// model/schema.js

export default appSchema({
```

```
  version: 1,
  tables: [
    // This is our new table!
    tableSchema({
      name: 'comments',
      columns: [
        { name: 'post_id', type: 'string', isIndexed: true },
        { name: 'body', type: 'string' },
      ],
    }),
    // ...
  ]
})
```

Refresh the simulator. You should again see the same "Migrations can't be newer than schema" error. If you see a different error, you made a syntax error.

Step 3: Bump schema version

Now that we made matching changes in the schema (source of truth about tables and columns) and migrations (the change in tables and columns), it's time to commit the change by bumping the version:

```
// model/schema.js

export default appSchema({
  version: 2,
  tables: [
    // ...
  ]
})
```

If you refresh again, your app should show up without issues — but now you can use the new tables/columns

Step 4: Test your migrations

Before shipping a new version of the app, please check that your database changes are all compatible:

1. Migrations test: Install the previous version of your app, then update to the version you're about to ship, and make sure it still works
2. Fresh schema install test: Remove the app, and then install the *new* version of the app, and make sure it works

Why is this order important

It's simply because React Native simulator (and often React web projects) are configured to automatically refresh when you save a file. You don't want to database to accidentally migrate (upgrade) with changes that have a mistake, or changes you haven't yet completed making. By making migrations first, and bumping version last, you can double check you haven't made a mistake.

Migrations API

Each migration must migrate to a version one above the previous migration, and have multiple *steps* (such as adding a new table, or new columns). Larger example:

```
schemaMigrations({
  migrations: [
    {
      toVersion: 3,
      steps: [
        createTable({
          name: 'comments',
          columns: [
            { name: 'post_id', type: 'string', isIndexed: true },
            { name: 'body', type: 'string' },
          ],
        }),
        addColumns({
          table: 'posts',
          columns: [
            { name: 'subtitle', type: 'string', isOptional: true },
            { name: 'is_pinned', type: 'boolean' },
          ],
        }),
      ],
    },
    {
      toVersion: 2,
      steps: [
        // ...
      ],
    },
  ],
})
```

Migration steps:

- `createTable({ name: 'table_name', columns: [ ... ] })` - same API as `tableSchema()`
- `addColumns({ table: 'table_name', columns: [ ... ] })` - you can add one or multiple columns to an existing table. The columns table has the same format as in schema definitions
- Other types of migrations (e.g. deleting or renaming tables and columns) are not yet implemented. See `migrations/index.js`. Please contribute!

Database reseting and other edge cases

1. When you're **not** using migrations, the database will reset (delete all its contents) whenever you change the schema version.
2. If the migration fails, the database will fail to initialize, and will roll back to previous version. This is unlikely, but could happen if you, for example, create a migration that tries to create the same table twice. The reason why the database will fail instead of reset is to avoid losing user data (also it's less confusing in development). You can notice the problem, fix the migration, and ship it again without data loss.

3. When database in the running app has *newer* database version than the schema version defined in code, the database will reset (clear its contents). This is useful in development
4. If there's no available migrations path (e.g. user has app with database version 4, but oldest migration is from version 10 to 11), the database will reset.

Rolling back changes

There's no automatic "rollback" feature in Watermelon. If you make a mistake in migrations during development, roll back in this order:

1. Comment out any changes made to schema.js
2. Comment out any changes made to migrations.js
3. Decrement schema version number (bring back the original number)

After refreshing app, the database should reset to previous state. Now you can correct your mistake and apply changes again (please do it in order described in "Migrations workflow").

Unsafe SQL migrations

Similar to Schema, you can add `unsafeSql` parameter to every migration step to modify or replace SQL generated by WatermelonDB to perform the migration. There is also an `unsafeExecuteSql('some sql;')` step you can use to append extra SQL. Those are ignored with LokiJSAdapter and for the purposes of migration syncs.

Previous
« Model

Next
Relations »

Relations

A `Relation` object represents one record pointing to another — such as the author (`User`) of a `Comment`, or the `Post` the comment belongs to.

Defining Relations

There's two steps to defining a relation:

1. A **table column** for the related record's ID

```
tableSchema({
  name: 'comments',
  columns: [
    // ...
    { name: 'author_id', type: 'string' },
  ]
}),
```

2. A `@relation` field defined on a `Model` class:

```
import { relation } from '@nozbe/watermelondb/decorators'

class Comment extends Model {
  // ...
  @relation('users', 'author_id') author
}
```

The first argument is the *table name* of the related record, and the second is the *column name* with an ID for the related record.

immutableRelation

If you have a relation that cannot change (for example, a comment can't change its author), use `@immutableRelation` for extra protection and performance:

```
import { immutableRelation } from '@nozbe/watermelondb/decorators'

class Comment extends Model {
  // ...
```

```
  @immutableRelation('posts', 'post_id') post
  @immutableRelation('users', 'author_id') author
}
```

Relation API

In the example above, `comment.author` returns a `Relation` object.

> Remember, WatermelonDB is a lazily-loaded database, so you don't get the related `User` record immediately, only when you explicitly fetch it

Observing

Most of the time, you connect Relations to Components by using `observe()` (the same as with Queries):

```
withObservables(['comment'], ({ comment }) => ({
  comment,
  author: comment.author, // shortcut syntax for `author: comment.author.observe()`
}))
```

The component will now have an `author` prop containing a `User`, and will re-render both when the user changes (e.g. comment's author changes its name), but also when a new author is assigned to the comment (if that was possible).

Fetching

To simply get the related record, use `fetch`. You might need it in a Writer

```
const author = await comment.author.fetch()

// Shortcut syntax:
const author = await comment.author
```

Note: If the relation column (in this example, `author_id`) is marked as `isOptional: true`, `fetch()` might return `null`.

ID

If you only need the ID of a related record (e.g. to use in an URL or for the `key=` React prop), use `id`.

```
const authorId = comment.author.id
```

Assigning

Use `set()` to assign a new record to the relation

```
await database.get('comments').create(comment => {
  comment.author.set(someUser)
  // ...
})
```

Note: you can only do this in the `.create()` or `.update()` block.

You can also use `set id` if you only have the ID for the record to assign

```
await comment.update(() => {
  comment.author.id = userId
})
```

Advanced relations

Many-To-Many Relation

If for instance, our app `Post`s can be authored by many `User`s and a user can author many `Post`s. We would create such a relation following these steps:-

1. Create a pivot schema and model that both the `User` model and `Post` model has association to; say `PostAuthor`
2. Create has_many association on both `User` and `Post` pointing to `PostAuthor` Model
3. Create belongs_to association on `PostAuthor` pointing to both `User` and `Post`
4. Retrieve all `Posts` for a user by defining a query that uses the pivot `PostAuthor` to infer the `Post`s that were authored by the User.

```
import { lazy } from '@nozbe/watermelondb/decorators'

class Post extends Model {
  static table = 'posts'
  static associations = {
    post_authors: { type: 'has_many', foreignKey: 'post_id' },
  }

  @lazy
  authors = this.collections
    .get('users')
    .query(Q.on('post_authors', 'post_id', this.id));
}
```

```
import { immutableRelation } from '@nozbe/watermelondb/decorators'

class PostAuthor extends Model {
  static table = 'post_authors'
  static associations = {
    posts: { type: 'belongs_to', key: 'post_id' },
    users: { type: 'belongs_to', key: 'user_id' },
  }
  @immutableRelation('posts', 'post_id') post
```

```
  @immutableRelation('users', 'user_id') user
}
```

```
import { lazy } from '@nozbe/watermelondb/decorators'

class User extends Model {
  static table = 'users'
  static associations = {
    post_authors: { type: 'has_many', foreignKey: 'user_id' },
  }

  @lazy
  posts = this.collections
    .get('posts')
    .query(Q.on('post_authors', 'user_id', this.id));

}
```

```
withObservables(['post'], ({ post }) => ({
  authors: post.authors,
}))
```

Next steps

➡️ Now the last step of this guide: **understand Writers (and Readers)**

Edit this page

Previous
« Migrations

Next
Create, Read, Update, Delete »

More

GitHub

Create, Read, Update, Delete

When you have your Schema and Models defined, learn how to manipulate them!

Reading

Get a collection

The `Collection` object is how you find, query, and create new records of a given type.

```
const postsCollection = database.get('posts')
```

Pass the table name as the argument.

Find a record (by ID)

```
const postId = 'abcdefgh'
const post = await database.get('posts').find(postId)
```

`find()` returns a Promise. If the record cannot be found, the Promise will be rejected.

Query records

Find a list of records matching given conditions by making a Query and then fetching it:

```
const allPosts = await database.get('posts').query().fetch()
const numberOfStarredPosts = await database.get('posts').query(
  Q.where('is_starred', true)
).fetchCount()
```

➡️ **Learn more:** Queries

Modifying the database

All modifications to the database (like creating, updating, deleting records) must be done **in a Writer**, either by wrapping your work in `database.write()`:

```
await database.write(async () => {
  const comment = await database.get('comments').find(commentId)
  await comment.update(() => {
    comment.isSpam = true
  })
})
```

Or by defining a `@writer` method on a Model:

```
import { writer } from '@nozbe/watermelondb/decorators'

class Comment extends Model {
  // (...)
  @writer async markAsSpam() {
    await this.update(comment => {
      comment.isSpam = true
    })
  }
}
```

➡️ **Learn more:** Writers

Create a new record

```
const newPost = await database.get('posts').create(post => {
  post.title = 'New post'
  post.body = 'Lorem ipsum...'
})
```

`.create()` takes a "builder function". In the example above, the builder will get a `Post` object as an argument. Use this object to set values for fields you defined.

Note: Always `await` the Promise returned by `create` before you access the created record.

Note: You can only set fields inside `create()` or `update()` builder functions.

Update a record

```
await somePost.update(post => {
  post.title = 'Updated title'
})
```

Like creating, updating takes a builder function, where you can use field setters.

Note: Always `await` the Promise returned by `update` before you access the modified record.

Delete a record

There are two ways of deleting records: syncable (mark as deleted), and permanent.

If you only use Watermelon as a local database, destroy records permanently, if you synchronize, mark as deleted instead.

```
await somePost.markAsDeleted() // syncable
await somePost.destroyPermanently() // permanent
```

Note: Do not access, update, or observe records after they're deleted.

Advanced

- `Model.observe()` - usually you only use this when connecting records to components, but you can manually observe a record outside of React components. The returned RxJS `Observable` will emit the record immediately upon subscription, and then every time the record is updated. If the record is deleted, the Observable will complete.
- `Query.observe()`, `Relation.observe()` — analagous to the above, but for Queries and Relations
- `Query.observeWithColumns()` - used for sorted lists
- `Collection.findAndObserve(id)` — same as using `.find(id)` and then calling `record.observe()`
- `Model.prepareUpdate()`, `Collection.prepareCreate`, `Database.batch` — used for batch updates
- `Database.unsafeResetDatabase()` destroys the whole database - be sure to see this comment before using it
- To override the `record.id` during the creation, e.g. to sync with a remote database, you can do it by `record._raw` property. Be aware that the `id` must be of type `string`.

  ```
  await database.get('posts').create(post => {
    post._raw.id = serverId
  })
  ```

Advanced: Unsafe raw execute

⚠️ Do not use this if you don't know what you're doing...

There is an escape hatch to drop down from WatermelonDB to underlying database level to execute arbitrary commands. Use as a last resort tool:

```
await database.write(() => {
  // sqlite:
  await database.adapter.unsafeExecute({
    sqls: [
      // [sql_query, [placeholder arguments, ...]]
      ['create table temporary_test (id, foo, bar)', []],
      ['insert into temporary_test (id, foo, bar) values (?, ?, ?)', ['t1', true, 3.14]],
    ]
  })

  // lokijs:
  await database.adapter.unsafeExecute({
    loki: loki => {
      loki.addCollection('temporary_test', { unique: ['id'], indices: [], disableMeta: true })
      loki.getCollection('temporary_test').insert({ id: 't1', foo: true, bar: 3.14 })
```

```
    }
  })
})
```

Next steps

➡️ Now that you can create and update records, **connect them to React components**

Edit this page

Previous
« Relations

Next
Connecting Components »

Query API

Querying is how you find records that match certain conditions, for example:

- Find all comments that belong to a certain post
- Find all *verified* comments made by John
- Count all verified comments made by John or Lucy published under posts made in the last two weeks

Because queries are executed on the database, and not in JavaScript, they're really fast. It's also how Watermelon can be fast even at large scales, because even with tens of thousands of records *total*, you rarely need to load more than a few dozen records at app launch.

Defining Queries

@children

The simplest query is made using `@children`. This defines a `Query` for all comments that belong to a `Post`:

```
class Post extends Model {
  // ...
  @children('comments') comments
}
```

➡️ **Learn more:** Defining Models

Extended Query

To **narrow down** a `Query` (add extra conditions to an existing Query), use `.extend()`:

```
import { Q } from '@nozbe/watermelondb'
import { children, lazy } from '@nozbe/watermelondb/decorators'

class Post extends Model {
  // ...
  @children('comments') comments

  @lazy verifiedComments = this.comments.extend(
    Q.where('is_verified', true)
  )
```

```
  @lazy verifiedAwesomeComments = this.verifiedComments.extend(
    Q.where('is_awesome', true)
  )
}
```

Note: Use `@lazy` when extending or defining new Queries for performance

Custom Queries

You can query any table like so:

```
import { Q } from '@nozbe/watermelondb'

const users = await database.get('users').query(
  // conditions that a user must match:
  Q.on('comments', 'post_id', somePostId)
).fetch()
```

This fetches all users that made a comment under a post with `id = somePostId`.

You can define custom queries on a Model like so:

```
class Post extends Model {
  // ...
  @lazy commenters = this.collections.get('users').query(
    Q.on('comments', 'post_id', this.id)
  )
}
```

Executing Queries

Most of the time, you execute Queries by connecting them to React Components like so:

```
withObservables(['post'], ({ post }) => ({
  post,
  comments: post.comments,
  verifiedCommentCount: post.verifiedComments.observeCount(),
}))
```

➡️ **Learn more:** Connecting to Components

Fetch

To simply get the current list or current count (without observing future changes), use `fetch` / `fetchCount`.

```
const comments = await post.comments.fetch()
const verifiedCommentCount = await post.verifiedComments.fetchCount()

// Shortcut syntax:
```

```
const comments = await post.comments
const verifiedCommentCount = await post.verifiedComments.count
```

Query conditions

```
import { Q } from '@nozbe/watermelondb'
// ...
database.get('comments').query(
  Q.where('is_verified', true)
)
```

This will query **all** comments that are verified (all comments with one condition: the `is_verified` column of a comment must be `true`).

When making conditions, you refer to **column names** of a table (i.e. `is_verified`, not `isVerified`). This is because queries are executed directly on the underlying database.

The second argument is the value we want to query for. Note that the passed argument must be the same type as the column (`string`, `number`, or `boolean`; `null` is allowed only if the column is marked as `isOptional: true` in the schema).

Empty query

```
const allComments = await database.get('comments').query().fetch()
```

A Query with no conditions will find **all** records in the collection.

Note: Don't do this unless necessary. It's generally more efficient to only query the exact records you need.

Multiple conditions

```
database.get('comments').query(
  Q.where('is_verified', true),
  Q.where('is_awesome', true)
)
```

This queries all comments that are **both** verified **and** awesome.

Conditions with other operators

Query	JavaScript equivalent
`Q.where('is_verified', true)`	`is_verified === true` (shortcut syntax)
`Q.where('is_verified', Q.eq(true))`	`is_verified === true`
`Q.where('archived_at', Q.notEq(null))`	`archived_at !== null`

Query	JavaScript equivalent
`Q.where('likes', Q.gt(0))`	`likes > 0`
`Q.where('likes', Q.weakGt(0))`	`likes > 0` (slightly different semantics — see "null behavior" for details)
`Q.where('likes', Q.gte(100))`	`likes >= 100`
`Q.where('dislikes', Q.lt(100))`	`dislikes < 100`
`Q.where('dislikes', Q.lte(100))`	`dislikes <= 100`
`Q.where('likes', Q.between(10, 100))`	`likes >= 10 && likes <= 100`
`Q.where('status', Q.oneOf(['published', 'draft']))`	`['published', 'draft'].includes(status)`
`Q.where('status', Q.notIn(['archived', 'deleted']))`	`status !== 'archived' && status !== 'deleted'`
`Q.where('status', Q.like('%bl_sh%'))`	`/.*bl.sh.*/i` (See note below!)
`Q.where('status', Q.notLike('%bl_sh%'))`	`/^((!?.*bl.sh.*).)*$/i` (Inverse regex match) (See note below!)
`Q.where('status', Q.includes('promoted'))`	`status.includes('promoted')`

LIKE / NOT LIKE

You can use `Q.like` for search-related tasks. For example, to find all users whose username start with "jas" (case-insensitive) you can write

```
usersCollection.query(
  Q.where("username", Q.like(`${Q.sanitizeLikeString("jas")}%`)
)
```

where `"jas"` can be changed dynamically with user input.

Note that the behavior of `Q.like` is not exact and can differ somewhat between implementations (SQLite vs LokiJS). For instance, while the comparison is case-insensitive, SQLite cannot by default compare non-ASCII characters case-insensitively (unless you install ICU extension). Use `Q.like` for user input search, but not for tasks that require a precise matching behavior.

Note: It's NOT SAFE to use `Q.like` and `Q.notLike` with user input directly, because special characters like `%` or `_` are not escaped. Always sanitize user input like so:

```
Q.like(`%${Q.sanitizeLikeString(userInput)}%`)
Q.notLike(`%${Q.sanitizeLikeString(userInput)}%`)
```

AND/OR nesting

You can nest multiple conditions using `Q.and` and `Q.or`:

```
database.get('comments').query(
  Q.where('archived_at', Q.notEq(null)),
  Q.or(
    Q.where('is_verified', true),
    Q.and(
      Q.where('likes', Q.gt(10)),
      Q.where('dislikes', Q.lt(5))
    )
  )
)
```

This is equivalent to `archivedAt !== null && (isVerified || (likes > 10 && dislikes < 5))`.

Conditions on related tables ("JOIN queries")

For example: query all comments under posts published by John:

```
// Shortcut syntax:
database.get('comments').query(
  Q.on('posts', 'author_id', john.id),
)

// Full syntax:
database.get('comments').query(
  Q.on('posts', Q.where('author_id', Q.eq(john.id))),
)
```

Normally you set conditions on the table you're querying. Here we're querying **comments**, but we have a condition on the **post** the comment belongs to.

The first argument for `Q.on` is the table name you're making a condition on. The other two arguments are same as for `Q.where`.

Note: The two tables must be associated before you can use `Q.on`.

Multiple conditions on a related table

For example: query all comments under posts that are written by John *and* are either published or belong to `draftBlog`

```
database.get('comments').query(
  Q.on('posts', [
    Q.where('author_id', john.id)
```

```
    Q.or(
      Q.where('published', true),
      Q.where('blog_id', draftBlog.id),
    )
  ]),
)
```

Instead of an array of conditions, you can also pass `Q.and`, `Q.or`, `Q.where`, or `Q.on` as the second argument to `Q.on`.

Nesting `Q.on` within AND/OR

If you want to place `Q.on` nested within `Q.and` and `Q.or`, you must explicitly define all tables you're joining on. (NOTE: The `Q.experimentalJoinTables` API is subject to change)

```
tasksCollection.query(
  Q.experimentalJoinTables(['projects']),
  Q.or(
    Q.where('is_followed', true),
    Q.on('projects', 'is_followed', true),
  ),
)
```

Deep `Q.on`s

You can also nest `Q.on` within `Q.on`, e.g. to make a condition on a grandparent. You must explicitly define the tables you're joining on. (NOTE: The `Q.experimentalNestedJoin` API is subject to change). Multiple levels of nesting are allowed.

```
// this queries tasks that are inside projects that are inside teams where team.foo == 'bar'
tasksCollection.query(
  Q.experimentalNestedJoin('projects', 'teams'),
  Q.on('projects', Q.on('teams', 'foo', 'bar')),
)
```

Advanced Queries

Advanced observing

Call `query.observeWithColumns(['foo', 'bar'])` to create an Observable that emits a value not only when the list of matching records changes (new records/deleted records), but also when any of the matched records changes its `foo` or `bar` column. Use this for observing sorted lists

Count throttling

By default, calling `query.observeCount()` returns an Observable that is throttled to emit at most once every 250ms. You can disable throttling using `query.observeCount(false)`.

Column comparisons

This queries comments that have more likes than dislikes. Note that we're comparing `likes` column to another column instead of a value.

```
database.get('comments').query(
  Q.where('likes', Q.gt(Q.column('dislikes')))
)
```

sortBy, take, skip

You can use these clauses to sort the query by one or more columns. Note that only simple ascending/descending criteria for columns are supported.

```
database.get('comments').query(
  // sorts by number of likes from the most likes to the fewest
  Q.sortBy('likes', Q.desc),
  // if two comments have the same number of likes, the one with fewest dislikes will be at the
  Q.sortBy('dislikes', Q.asc),
  // limit number of comments to 100, skipping the first 50
  Q.skip(50),
  Q.take(100),
)
```

It isn't *necessarily* better or more efficient to sort on query level instead of in JavaScript, **however** the most important use case for `Q.sortBy` is when used alongside `Q.skip` and `Q.take` to implement paging - to limit the number of records loaded from database to memory on very long lists

Fetch IDs

If you only need IDs of records matching a query, you can optimize the query by calling `await query.fetchIds()` instead of `await query.fetch()`

Security

Remember that Queries are a sensitive subject, security-wise. Never trust user input and pass it directly into queries. In particular:

- Never pass into queries values you don't know for sure are the right type (e.g. value passed to `Q.eq()` should be a string, number, boolean, or null -- but not an Object. If the value comes from JSON, you must validate it before passing it!)
- Never pass column names (without whitelisting) from user input
- Values passed to `oneOf`, `notIn` should be arrays of simple types - be careful they don't contain objects
- Do not use `Q.like` / `Q.notLike` without `Q.sanitizeLikeString`
- Do not use `unsafe raw queries` without knowing what you're doing and sanitizing all user input

Unsafe SQL queries

```
const records = await database.get('comments').query(
  Q.unsafeSqlQuery(`select * from comments where foo is not ? and _status is not 'deleted'`, ['b
).fetch()

const recordCount = await database.get('comments').query(
  Q.unsafeSqlQuery(`select count(*) as count from comments where foo is not ? and _status is not
).fetchCount()
```

You can also observe unsafe raw SQL queries, however, if it contains `JOIN` statements, you must explicitly specify all other tables using `Q.experimentalJoinTables` and/or `Q.experimentalNestedJoin`, like so:

```
const records = await database.get('comments').query(
  Q.experimentalJoinTables(['posts']),
  Q.experimentalNestedJoin('posts', 'blogs'),
  Q.unsafeSqlQuery(
    'select comments.* from comments ' +
      'left join posts on comments.post_id is posts.id ' +
      'left join blogs on posts.blog_id is blogs.id' +
      'where ...',
  ),
).observe()
```

⚠️ Please note:

- Do not use this if you don't know what you're doing
- Do not pass user input directly to avoid SQL Injection - use `?` placeholders and pass array of placeholder values
- You must filter out deleted record using `where _status is not 'deleted'` clause
- If you're going to fetch count of the query, use `count(*) as count` as the select result

Unsafe fetch raw

In addition to `.fetch()` and `.fetchIds()`, there is also `.unsafeFetchRaw()`. Instead of returning an array of `Model` class instances, it returns an array of raw objects.

You can use it as an unsafe optimization, or alongside `Q.unsafeSqlQuery`/`Q.unsafeLokiTransform` to create an advanced query that either skips fetching unnecessary columns or includes extra computed columns. For example:

```
const rawData = await database.get('posts').query(
  Q.unsafeSqlQuery(
    'select posts.text1, count(tag_assignments.id) as tag_count, sum(tag_assignments.rank) as ta
      ' left join tag_assignments on posts.id = tag_assignments.post_id' +
      ' group by posts.id' +
      ' order by posts.position desc',
  )
).unsafeFetchRaw()
```

⚠️ You MUST NOT mutate returned objects. Doing so will corrupt the database.

Unsafe SQL/Loki expressions

You can also include smaller bits of SQL and Loki expressions so that you can still use as much of Watermelon query builder as possible:

```
// SQL example:
postsCollection.query(
  Q.where('is_published', true),
  Q.unsafeSqlExpr('tasks.num1 not between 1 and 5'),
)

// LokiJS example:
postsCollection.query(
  Q.where('is_published', true),
  Q.unsafeLokiExpr({ text1: { $contains: 'hey' } })
)
```

For SQL, be sure to prefix column names with table name when joining with other tables.

⚠️ Please do not use this if you don't know what you're doing. Do not pass user input directly to avoid SQL injection.

Multi-table column comparisons and `Q.unsafeLokiTransform`

Example: we want to query comments posted more than 14 days after the post it belongs to was published.

There's sadly no built-in syntax for this, but can be worked around using unsafe expressions like so:

```
// SQL example:
commentsCollection.query(
  Q.on('posts', 'published_at', Q.notEq(null)),
  Q.unsafeSqlExpr(`comments.createad_at > posts.published_at + ${14 * 24 * 3600 * 1000}`)
)

// LokiJS example:
commentsCollection.query(
  Q.on('posts', 'published_at', Q.notEq(null)),
  Q.unsafeLokiTransform((rawRecords, loki) => {
    return rawRecords.filter(rawRecord => {
      const post = loki.getCollection('posts').by('id', rawRecord.post_id)
      return post && rawRecord.created_at > post.published_at + 14 * 24 * 3600 * 1000
    })
  }),
)
```

For LokiJS, remember that `rawRecord` is an unsanitized, unsafe object and must not be mutated. `Q.unsafeLokiTransform` only works when using `LokiJSAdapter` with `useWebWorkers: false`. There can only be one `Q.unsafeLokiTransform` clause per query.

`null` behavior

There are some gotchas you should be aware of. The `Q.gt`, `gte`, `lt`, `lte`, `oneOf`, `notIn`, `like` operators

match the semantics of SQLite in terms of how they treat `null`. Those are different from JavaScript.

Rule of thumb: No null comparisons are allowed.

For example, if you query `comments` for `Q.where('likes', Q.lt(10))`, a comment with 8 likes and 0 likes will be included, but a comment with `null` likes will not! In Watermelon queries, `null` is not less than any number. That's why you should avoid making table columns optional unless you actually need it.

Similarly, if you query with a column comparison, like `Q.where('likes', Q.gt(Q.column('dislikes')))`, only comments where both `likes` and `dislikes` are not null will be compared. A comment with 5 likes and `null` dislikes will NOT be included. 5 is not greater than `null` here.

`Q.oneOf` **operator**: It is not allowed to pass `null` as an argument to `Q.oneOf`. Instead of `Q.oneOf([null, 'published', 'draft'])` you need to explicitly allow `null` as a value like so:

```
postsCollection.query(
  Q.or(
    Q.where('status', Q.oneOf(['published', 'draft'])),
    Q.where('status', null)
  )
)
```

`Q.notIn` **operator**: If you query, say, posts with `Q.where('status', Q.notIn(['published', 'draft']))`, it will match posts with a status different than `published` or `draft`, however, it will NOT match posts with `status == null`. If you want to include such posts, query for that explicitly like with the example above.

`Q.weakGt` **operator**: This is weakly typed version of `Q.gt` — one that allows null comparisons. So if you query `comments` with `Q.where('likes', Q.weakGt(Q.column('dislikes')))`, it WILL match comments with 5 likes and `null` dislikes. (For `weakGt`, unlike standard operators, any number is greater than `null`).

Contributing improvements to Watermelon query language

Here are files that are relevant. This list may look daunting, but adding new matchers is actually quite simple and multiple first-time contributors made these improvements (including like, sort, take, skip). The implementation is just split into multiple files (and their test files), but when you look at them, it'll be easy to add matchers by analogy.

We recommend starting from writing tests first to check expected behavior, then implement the actual behavior.

- `src/QueryDescription/test.js` - Test clause builder (`Q.myThing`) output and test that it rejects bad/unsafe parameters
- `src/QueryDescription/index.js` - Add clause builder and type definition
- `src/__tests__/databaseTests.js` - Add test ("join" if it requires conditions on related tables; "match" otherwise) that checks that the new clause matches expected records. From this, tests running against SQLite, LokiJS, and Matcher are generated. (If one of those is not supported, add `skip{Loki,Sql,Count,Matcher}: true` to your test)

- `src/adapters/sqlite/encodeQuery/test.js` - Test that your query generates SQL you expect. (If your clause is Loki-only, test that error is thrown)
- `src/adapters/sqlite/encodeQuery/index.js` - Generate SQL
- `src/adapters/lokijs/worker/encodeQuery/test.js` - Test that your query generates the Loki query you expect (If your clause is SQLite-only, test that an error is thrown)
- `src/adapters/lokijs/worker/encodeQuery/index.js` - Generate Loki query
- `src/adapters/lokijs/worker/{performJoins/*.js,executeQuery.js}` - May be relevant for some Loki queries, but most likely you don't need to look here.
- `src/observation/encodeMatcher/` - If your query can be checked against a record in JavaScript (e.g. you're adding new "by regex" matcher), implement this behavior here (`index.js`, `operators.js`). This is used for efficient "simple observation". You don't need to write tests - `databaseTests` are used automatically. If you can't or won't implement encodeMatcher for your query, add a check to `canEncode.js` so that it returns `false` for your query (Less efficient "reloading observation" will be used then). Add your query to `test.js`'s "unencodable queries" then.

Next steps

➡️ Now that you've mastered Queries, **make more Relations**

Edit this page

Previous
« Connecting Components

Next
Writers, Readers, Batching »

Docs

Installation

Contributing

Community

Stack Overflow

Twitter

More

GitHub

Writers, Readers, and batching

Think of this guide as a part two of Create, Read, Update, Delete.

As mentioned previously, you can't just modify WatermelonDB's database anywhere. All changes must be done within a **Writer**.

There are two ways of defining a writer: inline and by defining a **writer method**.

Inline writers

Here is an inline writer, you can invoke it anywhere you have access to the `database` object:

```
// Note: function passed to `database.write()` MUST be asynchronous
const newPost = await database.write(async => {
  const post = await database.get('posts').create(post => {
    post.title = 'New post'
    post.body = 'Lorem ipsum...'
  })
  const comment = await database.get('comments').create(comment => {
    comment.post.set(post)
    comment.author.id = someUserId
    comment.body = 'Great post!'
  })

  // Note: Value returned from the wrapped function will be returned to `database.write` caller
  return post
})
```

Writer methods

Writer methods can be defined on `Model` subclasses by using the `@writer` decorator:

```
import { writer } from '@nozbe/watermelondb/decorators'

class Post extends Model {
  // ...

  @writer async addComment(body, author) {
    const newComment = await this.collections.get('comments').create(comment => {
      comment.post.set(this)
      comment.author.set(author)
```

```
      comment.body = body
    })
    return newComment
  }
}
```

We highly recommend defining writer methods on `Models` to organize all code that changes the database in one place, and only use inline writers sporadically.

Note that this is the same as defining a simple method that wraps all work in `database.write()` - using `@writer` is simply more convenient.

Note:

- Always mark actions as `async` and remember to `await` on `.create()` and `.update()`
- You can use `this.collections` to access `Database.collections`

Another example: updater action on `Comment`:

```
class Comment extends Model {
  // ...
  @field('is_spam') isSpam

  @writer async markAsSpam() {
    await this.update(comment => {
      comment.isSpam = true
    })
  }
}
```

Now we can create a comment and immediately mark it as spam:

```
const comment = await post.addComment('Lorem ipsum', someUser)
await comment.markAsSpam()
```

Batch updates

When you make multiple changes in a writer, it's best to **batch them**.

Batching means that the app doesn't have to go back and forth with the database (sending one command, waiting for the response, then sending another), but instead sends multiple commands in one big batch. This is faster, safer, and can avoid subtle bugs in your app

Take an action that changes a `Post` into spam:

```
class Post extends Model {
  // ...
  @writer async createSpam() {
    await this.update(post => {
      post.title = `7 ways to lose weight`
```

```
    })
    await this.collections.get('comments').create(comment => {
      comment.post.set(this)
      comment.body = "Don't forget to comment, like, and subscribe!"
    })
  }
}
```

Let's modify it to use batching:

```
class Post extends Model {
  // ...
  @writer async createSpam() {
    await this.batch(
      this.prepareUpdate(post => {
        post.title = `7 ways to lose weight`
      }),
      this.collections.get('comments').prepareCreate(comment => {
        comment.post.set(this)
        comment.body = "Don't forget to comment, like, and subscribe!"
      })
    )
  }
}
```

Note:

- You can call `await this.batch` within `@writer` methods only. You can also call `database.batch()` within a `database.write()` block.
- Pass the list of **prepared operations** as arguments:
  - Instead of calling `await record.update()`, pass `record.prepareUpdate()` — note lack of `await`
  - Instead of `await collection.create()`, use `collection.prepareCreate()`
  - Instead of `await record.markAsDeleted()`, use `record.prepareMarkAsDeleted()`
  - Instead of `await record.destroyPermanently()`, use `record.prepareDestroyPermanently()`
  - Advanced: you can pass `collection.prepareCreateFromDirtyRaw({ put your JSON here })`
  - You can pass falsy values (null, undefined, false) to batch — they will simply be ignored.
  - You can also pass a single array argument instead of a list of arguments

Delete action

When you delete, say, a `Post`, you generally want all `Comment`s that belong to it to be deleted as well.

To do this, override `markAsDeleted()` (or `destroyPermanently()` if you don't sync) to explicitly delete all children as well.

```
class Post extends Model {
  static table = 'posts'
  static associations = {
    comments: { type: 'has_many', foreignKey: 'post_id' },
  }
```

```
  @children('comments') comments

  async markAsDeleted() {
    await this.comments.destroyAllPermanently()
    await super.markAsDeleted()
  }
}
```

Then to actually delete the post:

```
database.write(async () => {
  await post.markAsDeleted()
})
```

Note:

- Use `Query.destroyAllPermanently()` on all dependent `@children` you want to delete
- Remember to call `super.markAsDeleted` — at the end of the method!

Advanced: Why are readers and writers necessary?

WatermelonDB is highly asynchronous, which is a BIG challange in terms of achieving consistent data. Read this only if you are curious:

▸ Why are readers and writers necessary?

Advanced: Readers

Readers are an advanced feature you'll rarely need.

Because WatermelonDB is asynchronous, if you make multiple separate queries, normally you have no guarantee that no records were created, updated, or deleted between fetching these queries.

Code within a Reader, however, has a guarantee that for the duration of the Reader, no changes will be made to the database (more precisely, no Writer can execute during Reader's work).

For example, if you were writing a custom XML data export feature for your app, you'd want the information there to be fully consistent. Therefore, you'd wrap all queries within a Reader:

```
database.read(async () => {
  // no changes will happen to the database until this function exits
})

// alternatively:
class Blog extends Model {
  // ...
```

```
  @reader async exportBlog() {
    const posts = await this.posts.fetch()
    const comments = await this.allComments.fetch()
    // ...
  }
}
```

Advanced: nesting writers or readers

If you try to call a Writer from another Writer, you'll notice that it won't work. This is because while a Writer is running, no other Writer can run simultaneously. To override this behavior, wrap the Writer call in `this.callWriter`:

```
class Comment extends Model {
  // ...

  @writer async appendToPost() {
    const post = await this.post.fetch()
    // `appendToBody` is an `@writer` on `Post`, so we call callWriter to allow it
    await this.callWriter(() => post.appendToBody(this.body))
  }
}

// alternatively:
database.write(async writer => {
  const post = await database.get('posts').find('abcdef')
  await writer.callWriter(() => post.appendToBody('Lorem ipsum...')) // appendToBody is a @write
})
```

The same is true with Readers - use `callReader` to nest readers.

Next steps

➡️ Now that you've mastered all basics of Watermelon, go create some powerful apps — or keep reading **advanced guides**

Edit this page

Previous
« Querying

Next
Intro »

Docs

Version: 0.27.1

Synchronization

WatermelonDB has been designed from scratch to be able to seamlessly synchronize with a remote database (and, therefore, keep multiple copies of data synced with each other).

Note that Watermelon is only a local database — you need to **bring your own backend**. What Watermelon provides are:

- **Synchronization primitives** — information about which records were created, updated, or deleted locally since the last sync — and which columns exactly were modified. You can build your own custom sync engine using those primitives
- **Built-in sync adapter** — You can use the sync engine Watermelon provides out of the box, and you only need to provide two API endpoints on your backend that conform to Watermelon sync protocol

To implement synchronization between your client side database (WatermelonDB) and your server, you need to implement synchronization in the frontend & the backend.

Edit this page

Previous
« Writers, Readers, Batching

Next
Frontend »

Docs

Installation

Contributing

Community

Stack Overflow

Twitter

More

GitHub

Implementing sync in frontend

Using `synchronize()` in your app

To synchronize, you need to pass `pullChanges` and `pushChanges` *(optional)* that talk to your backend and are compatible with Watermelon Sync Protocol. The frontend code will look something like this:

```
import { synchronize } from '@nozbe/watermelondb/sync'

async function mySync() {
  await synchronize({
    database,
    pullChanges: async ({ lastPulledAt, schemaVersion, migration }) => {
      const urlParams = `last_pulled_at=${lastPulledAt}&schema_version=${schemaVersion}&migratic
        JSON.stringify(migration),
      )}`
      const response = await fetch(`https://my.backend/sync?${urlParams}`)
      if (!response.ok) {
        throw new Error(await response.text())
      }

      const { changes, timestamp } = await response.json()
      return { changes, timestamp }
    },
    pushChanges: async ({ changes, lastPulledAt }) => {
      const response = await fetch(`https://my.backend/sync?last_pulled_at=${lastPulledAt}`, {
        method: 'POST',
        body: JSON.stringify(changes),
      })
      if (!response.ok) {
        throw new Error(await response.text())
      }
    },
    migrationsEnabledAtVersion: 1,
  })
}
```

Who calls `synchronize()`?

Upon looking at the example above, one question that may arise is who will call `synchronize()` -- or, in the example above `mySync()`. WatermelonDB does not manage the moment of invocation of the `synchronize()`

function in any way. The database assumes every call of `pullChanges` will return *all* the changes that haven't yet been replicated (up to `last_pulled_at`). The application code is responsible for calling `synchronize()` in the frequency it deems necessary.

Implementing `pullChanges()`

Watermelon will call this function to ask for changes that happened on the server since the last pull.

Arguments:

- `lastPulledAt` is a timestamp for the last time client pulled changes from server (or `null` if first sync)
- `schemaVersion` is the current schema version of the local database
- `migration` is an object representing schema changes since last sync (or `null` if up to date or not supported)

This function should fetch from the server the list of ALL changes in all collections since `lastPulledAt`.

1. You MUST pass an async function or return a Promise that eventually resolves or rejects
2. You MUST pass `lastPulledAt`, `schemaVersion`, and `migration` to an endpoint that conforms to Watermelon Sync Protocol
3. You MUST return a promise resolving to an object of this shape (your backend SHOULD return this shape already):

```
{
  changes: { ... }, // valid changes object
  timestamp: 100000, // integer with *server's* current time
}
```

4. You MUST NOT store the object returned in `pullChanges()`. If you need to do any processing on it, do it before returning the object. Watermelon treats this object as "consumable" and can mutate it (for performance reasons)

Implementing `pushChanges()`

Watermelon will call this function with a list of changes that happened locally since the last push so you can post it to your backend.

Arguments passed:

```
{
  changes: { ... }, // valid changes object
  lastPulledAt: 10000, // the timestamp of the last successful pull (timestamp returned in pullC
}
```

1. You MUST pass `changes` and `lastPulledAt` to a push sync endpoint conforming to Watermelon Sync Protocol
2. You MUST pass an async function or return a Promise from `pushChanges()`
3. `pushChanges()` MUST resolve after and only after the backend confirms it successfully received local

changes
4. `pushChanges()` MUST reject if backend failed to apply local changes
5. You MUST NOT resolve sync prematurely or in case of backend failure
6. You MUST NOT mutate or store arguments passed to `pushChanges()`. If you need to do any processing on it, do it before returning the object. Watermelon treats this object as "consumable" and can mutate it (for performance reasons)

Checking unsynced changes

WatermelonDB has a built in function to check whether there are any unsynced changes. The frontend code will look something like this

```
import { hasUnsyncedChanges } from '@nozbe/watermelondb/sync'

async function checkUnsyncedChanges() {
  const database = useDatabase()
  await hasUnsyncedChanges({ database })
}
```

General information and tips

1. You MUST NOT connect to backend endpoints you don't control using `synchronize()`. WatermelonDB assumes pullChanges/pushChanges are friendly and correct and does not guarantee secure behavior if data returned is malformed.
2. You SHOULD NOT call `synchronize()` while synchronization is already in progress (it will safely abort)
3. You MUST NOT reset local database while synchronization is in progress (push to server will be safely aborted, but consistency of the local database may be compromised)
4. You SHOULD wrap `synchronize()` in a "retry once" block - if sync fails, try again once. This will resolve push failures due to server-side conflicts by pulling once again before pushing.
5. You can use `database.withChangesForTables` to detect when local changes occured to call sync. If you do this, you should debounce (or throttle) this signal to avoid calling `synchronize()` too often.

Adopting Migration Syncs

For Watermelon Sync to maintain consistency after migrations, you must support Migration Syncs (introduced in WatermelonDB v0.17). This allows Watermelon to request from backend the tables and columns it needs to have all the data.

1. For new apps, pass `{migrationsEnabledAtVersion: 1}` to `synchronize()` (or the first schema version that shipped / the oldest schema version from which it's possible to migrate to the current version)
2. To enable migration syncs, the database MUST be configured with migrations spec (even if it's empty)
3. For existing apps, set `migrationsEnabledAtVersion` to the current schema version before making any schema changes. In other words, this version should be the last schema version BEFORE the first migration that should support migration syncs.

4. Note that for apps that shipped before WatermelonDB v0.17, it's not possible to determine what was the last schema version at which the sync happened. `migrationsEnabledAtVersion` is used as a placeholder in this case. It's not possible to guarantee that all necessary tables and columns will be requested. (If user logged in when schema version was lower than `migrationsEnabledAtVersion`, tables or columns were later added, and new records in those tables/changes in those columns occured on the server before user updated to an app version that has them, those records won't sync). To work around this, you may specify `migrationsEnabledAtVersion` to be the oldest schema version from which it's possible to migrate to the current version. However, this means that users, after updating to an app version that supports Migration Syncs, will request from the server all the records in new tables. This may be unacceptably inefficient.
5. WatermelonDB >=0.17 will note the schema version at which the user logged in, even if migrations are not enabled, so it's possible for app to request from backend changes from schema version lower than `migrationsEnabledAtVersion`
6. You MUST NOT delete old migrations, otherwise it's possible that the app is permanently unable to sync.

(Advanced) Adopting Turbo Login

WatermelonDB v0.23 introduced an advanced optimization called "Turbo Login". Syncing using Turbo is up to 5.3x faster than the traditional method and uses a lot less memory, so it's suitable for even very large syncs. Keep in mind:

1. This can only be used for the initial (login) sync, not for incremental syncs. It is a serious programmer error to run sync in Turbo mode if the database is not empty.
2. Syncs with `deleted: []` fields not empty will fail.
3. Turbo only works with SQLiteAdapter with JSI enabled and running - it does not work on web, or if e.g. Chrome Remote Debugging is enabled
4. While Turbo Login is stable, it's marked as "unsafe", meaning that the exact API may change in a future version

Here's basic usage:

```
const isFirstSync = ...
const useTurbo = isFirstSync
await synchronize({
  database,
  pullChanges: async ({ lastPulledAt, schemaVersion, migration }) => {
    const response = await fetch(`https://my.backend/sync?${...}`)
    if (!response.ok) {
      throw new Error(await response.text())
    }

    if (useTurbo) {
      // NOTE: DO NOT parse JSON, we want raw text
      const json = await response.text()
      return { syncJson: json }
    } else {
      const { changes, timestamp } = await response.json()
      return { changes, timestamp }
    }
  },
```

```
  unsafeTurbo: useTurbo,
  // ...
})
```

Raw JSON text is required, so it is not expected that you need to do any processing in pullChanges() - doing that defeats much of the point of using Turbo Login!

If you're using pullChanges to send additional data to your app other than Watermelon Sync's `changes` and `timestamp`, you won't be able to process it in pullChanges. However, WatermelonDB can still pass extra keys in sync response back to the app - you can process them using `onDidPullChanges`. This works both with and without turbo mode:

```
await synchronize({
  database,
  pullChanges: async ({ lastPulledAt, schemaVersion, migration }) => {
    // ...
  },
  unsafeTurbo: useTurbo,
  onDidPullChanges: async ({ messages }) => {
    if (messages) {
      messages.forEach((message) => {
        alert(message)
      })
    }
  },
  // ...
})
```

There's a way to make Turbo Login even more *turbo*! However, it requires native development skills. You need to develop your own native networking code, so that raw JSON can go straight from your native code to WatermelonDB's native code - skipping JavaScript processing altogether.

```
await synchronize({
  database,
  pullChanges: async ({ lastPulledAt, schemaVersion, migration }) => {
    // NOTE: You need the standard JS code path for incremental syncs

    // Create a unique id for this sync request
    const syncId = Math.floor(Math.random() * 1000000000)

    await NativeModules.MyNetworkingPlugin.pullSyncChanges(
      // Pass the id
      syncId,
      // Pass whatever information your plugin needs to make the request
      lastPulledAt,
      schemaVersion,
      migration,
    )

    // If successful, return the sync id
    return { syncJsonId: syncId }
  },
  unsafeTurbo: true,
  // ...
})
```

In native code, perform network request and if successful, extract raw response body data - `NSData *` on iOS, `byte[]` on Android. Avoid extracting the response as a string or parsing the JSON. Then pass it to WatermelonDB's native code:

```
// On Android (Java):
import com.nozbe.watermelondb.jsi.WatermelonJSI;

WatermelonJSI.provideSyncJson(/* id */ syncId, /* byte[] */ data);
```

```
// On iOS (Objective-C):
// (If using Swift, add the import to the bridging header)
#import <WatermelonDB/WatermelonDB.h>

watermelondbProvideSyncJson(syncId, data, &error)
```

Adding logging to your sync

You can add basic sync logs to the sync process by passing an empty object to `synchronize()`. Sync will then mutate the object, populating it with diagnostic information (start/finish time, resolved conflicts, number of remote/local changes, any errors that occured, and more):

```
// Using built-in SyncLogger
import SyncLogger from '@nozbe/watermelondb/sync/SyncLogger'
const logger = new SyncLogger(10 /* limit of sync logs to keep in memory */ )
await synchronize({ database, log: logger.newLog(), ... })

// this returns all logs (censored and safe to use in production code)
console.log(logger.logs)
// same, but pretty-formatted to a string (a user can easy copy this for diagnostic purposes)
console.log(logger.formattedLogs)


// You don't have to use SyncLogger, just pass a plain object to synchronize()
const log = {}
await synchronize({ database, log, ... })
console.log(log.startedAt)
console.log(log.finishedAt)
```

⚠️ Remember to act responsibly with logs, since they might contain your user's private information. Don't display, save, or send the log unless you censor the log.

Debugging `changes`

If you want to conveniently see incoming and outgoing changes in sync in the console, add these lines to your pullChanges/pushChanges:

⚠️ Leaving such logging committed and running in production is a huge security vulnerability and a

performance hog.

```
// UNDER NO CIRCUMSTANCES SHOULD YOU COMMIT THESE LINES UNCOMMENTED!!!
require('@nozbe/watermelondb/sync/debugPrintChanges').default(changes, isPush)
```

Pass `true` for second parameter if you're checking outgoing changes (pushChanges), `false` otherwise. Make absolutely sure you don't commit this debug tool. For best experience, run this on web (Chrome) -- the React Native experience is not as good.

(Advanced) Replacement Sync

Added in WatermelonDB 0.25, there is an alternative way to synchronize changes with the server called "Replacement Sync". You should only use this as last resort for cases difficult to deal with in an incremental fashion, due to performance implications.

Normally, `pullChanges` is expected to only return changes to data that had occured since `lastPulledAt`. During Replacement Sync, server sends the full dataset - *all* records that user has access to, same as during initial (first/login) sync.

Instead of applying these changes normally, the app will replace its database with the data set received, except that local unpushed changes will be preserved. In other words:

- App will create records that are new locally, and update the rest to the server state as per usual
- Records that have unpushed changes locally will go through conflict resolution as per usual
- HOWEVER, instead of server passing a list of records to delete, app will delete local records not present in the dataset received
- Details on how unpushed changes are preserved:
  - Records marked as `created` are preserved so they have a chance to sync
  - Records marked as `updated` or `deleted` will be preserved if they're contained in dataset received. Otherwise, they're deleted (since they were remotely deleted/server no longer grants you accecss to them, these changes would be ignored anyway if pushed).

If there are no local (unpushed) changes before or during sync, replacement sync should yield the same state as clearing database and performing initial sync. In case replacement sync is performed with an empty dataset (and there are no local changes), the result should be equivalent to clearing database.

When should you use Replacement Sync?

- You can use it as a way to fix a bad sync state (mismatch between local and remote state)
- You can use it in case you have a very large state change and your server doesn't know how to correctly calculate incremental changes since last sync (e.g. accessible records changed in a very complex permissions system)

In such cases, you could alternatively relogin (clear the database, then perform initial sync again), however:

- Replacement Sync preserves local changes to records (and other state such as Local Storage), so there's minimal risk for data loss

- When clearing the database, you need to give up all references to Watermelon objects and stop all observation. Therefore, you need to unmount all UI that touches Watermelon, leading to poor UX. This is not required for Replacement Sync
- On the other hand, Replacement Sync is much, much slower than Turbo Login (it's not possible to combine the two techniques), so this technique might not scale to very large datasets

Using Replacement Sync

In `pullChanges`, return an object with an extra `strategy` field

```
```js
{
  changes: { ... },
  timestamp: ...,
  experimentalStrategy: 'replacement',
}
```
```

Additional `synchronize()` flags

- `_unsafeBatchPerCollection: boolean` - if true, changes will be saved to the database in multiple batches. This is unsafe and breaks transactionality, however may be required for very large syncs due to memory issues
- `sendCreatedAsUpdated: boolean` - if your backend can't differentiate between created and updated records, set this to `true` to supress warnings. Sync will still work well, however error reporting, and some edge cases will not be handled as well.
- `conflictResolver: (TableName, local: DirtyRaw, remote: DirtyRaw, resolved: DirtyRaw) => DirtyRaw` - can be passed to customize how records are updated when they change during sync. See `src/sync/index.js` for details.
- `onWillApplyRemoteChanges` - called after pullChanges is done, but before these changes are applied. Some stats about the pulled changes are passed as arguments. An advanced user can use this for example to show some UI to the user when processing a very large sync (could be useful for replacement syncs). Note that remote change count is NaN in turbo mode.

Edit this page

Previous
« Intro

Next
Backend »

Docs

Installation

Contributing

Community

Stack Overflow

Twitter

More

GitHub

Implementing your Sync backend

Understanding `changes` objects

Synchronized changes (received by the app in `pullChanges` and sent to the backend in `pushChanges`) are represented as an object with *raw records*. Those only use raw table and column names, and raw values (strings/numbers/booleans) — the same as in Schema.

Deleted objects are always only represented by their IDs.

Example:

```
{
  projects: {
    created: [
      { id: 'aaaa', name: 'Foo', is_favorite: true },
      { id: 'bbbb', name: 'Bar', is_favorite: false },
    ],
    updated: [
      { id: 'ccc', name: 'Baz', is_favorite: true },
    ],
    deleted: ['ddd'],
  },
  tasks: {
    created: [],
    updated: [
      { id: 'tttt', name: 'Buy eggs' },
    ],
    deleted: [],
  },
  ...
}
```

Again, notice the properties returned have the format defined in the Schema (e.g. `is_favorite`, not `isFavorite`).

Valid changes objects MUST conform to this shape:

```
Changes = {
  [table_name: string]: {
    created: RawRecord[],
    updated: RawRecord[],
```

```
    deleted: string[],
  }
}
```

Implementing pull endpoint

Expected parameters:

```
{
  lastPulledAt: Timestamp,
  schemaVersion: int,
  migration: null | { from: int, tables: string[], columns: { table: string, columns: string[] }
}
```

Expected response:

```
{ changes: Changes, timestamp: Timestamp }
```

1. The pull endpoint SHOULD take parameters and return a response matching the shape specified above. This shape MAY be different if negotiated with the frontend (however, frontend-side `pullChanges()` MUST conform to this)
2. The pull endpoint MUST return all record changes in all collections since `lastPulledAt`, specifically:
   - all records that were created on the server since `lastPulledAt`
   - all records that were updated on the server since `lastPulledAt`
   - IDs of all records that were deleted on the server since `lastPulledAt`
   - record IDs MUST NOT be duplicated
3. If `lastPulledAt` is null or 0, you MUST return all accessible records (first sync)
4. The timestamp returned by the server MUST be a value that, if passed again to `pullChanges()` as `lastPulledAt`, will return all changes that happened since this moment.
5. The pull endpoint MUST provide a consistent view of changes since `lastPulledAt`
   - You should perform all queries synchronously or in a write lock to ensure that returned changes are consistent
   - You should also mark the current server time synchronously with the queries
   - This is to ensure that no changes are made to the database while you're fetching changes (otherwise some records would never be returned in a pull query)
   - If it's absolutely not possible to do so, and you have to query each collection separately, be sure to return a `lastPulledAt` timestamp marked BEFORE querying starts. You still risk inconsistent responses (that may break app's consistency assumptions), but the next pull will fetch whatever changes occured during previous pull.
   - An alternative solution is to check for the newest change before and after all queries are made, and if there's been a change during the pull, return an error code, or retry.
6. If `migration` is not null, you MUST include records needed to get a consistent view after a local database migration
   - Specifically, you MUST include all records in tables that were added to the local database between the last user sync and `schemaVersion`

    - For all columns that were added to the local app database between the last sync and `schemaVersion`, you MUST include all records for which the added column has a value other than the default value (`0`, `''`, `false`, or `null` depending on column type and nullability)
    - You can determine what schema changes were made to the local app in two ways:
        - You can compare `migration.from` (local schema version at the time of the last sync) and `schemaVersion` (current local schema version). This requires you to negotiate with the frontend what schema changes are made at which schema versions, but gives you more control
        - Or you can ignore `migration.from` and only look at `migration.tables` (which indicates which tables were added to the local database since the last sync) and `migration.columns` (which indicates which columns were added to the local database to which tables since last sync).
        - If you use `migration.tables` and `migration.columns`, you MUST whitelist values a client can request. Take care not to leak any internal fields to the client.
7. Returned raw records MUST match your app's Schema
8. Returned raw records MUST NOT not contain special `_status`, `_changed` fields.
9. Returned raw records MAY contain fields (columns) that are not yet present in the local app (at `schemaVersion` -- but added in a later version). They will be safely ignored.
10. Returned raw records MUST NOT contain arbitrary column names, as they may be unsafe (e.g. `__proto__` or `constructor`). You should whitelist acceptable column names.
11. Returned record IDs MUST only contain safe characters
    - Default WatermelonDB IDs conform to `/^[a-zA-Z0-9]{16}$/`
    - `_-.` are also allowed if you override default ID generator, but `'"\/$` are unsafe
12. Changes SHOULD NOT contain collections that are not yet present in the local app (at `schemaVersion`). They will, however, be safely ignored.
    - NOTE: This is true for WatermelonDB v0.17 and above. If you support clients using earlier versions, you MUST NOT return collections not known by them.
13. Changes MUST NOT contain collections with arbitrary names, as they may be unsafe. You should whitelist acceptable collection names.

Implementing push endpoint

1. The push endpoint MUST apply local changes (passed as a `changes` object) to the database. Specifically:
    - create new records as specified by the changes object
    - update existing records as specified by the changes object
    - delete records by the specified IDs
2. If the `changes` object contains a new record with an ID that already exists, you MUST update it, and MUST NOT return an error code.
    - (This happens if previous push succeeded on the backend, but not on frontend)
3. If the `changes` object contains an update to a record that does not exist, then:
    - If you can determine that this record no longer exists because it was deleted, you SHOULD return an error code (to force frontend to pull the information about this deleted ID)
    - Otherwise, you MUST create it, and MUST NOT return an error code. (This scenario should not happen, but in case of frontend or backend bugs, it would keep sync from ever succeeding.)
4. If the `changes` object contains a record to delete that doesn't exist, you MUST ignore it and MUST NOT return an error code

    - (This may happen if previous push succeeded on the backend, but not on frontend, or if another user deleted this record in between user's pull and push calls)
5. If the `changes` object contains a record that has been modified on the server after `lastPulledAt`, you MUST abort push and return an error code
    - This scenario means that there's a conflict, and record was updated remotely between user's pull and push calls. Returning an error forces frontend to call pull endpoint again to resolve the conflict
6. If application of all local changes succeeds, the endpoint MUST return a success status code.
7. The push endpoint MUST be fully transactional. If there is an error, all local changes MUST be reverted on the server, and en error code MUST be returned.
8. You MUST ignore `_status` and `_changed` fields contained in records in `changes` object
9. You SHOULD validate data passed to the endpoint. In particular, collection and column names ought to be whitelisted, as well as ID format — and of course any application-specific invariants, such as permissions to access and modify records
10. You SHOULD sanitize record fields passed to the endpoint. If there's something slightly wrong with the contents (but not shape) of the data (e.g. `user.role` should be `owner`, `admin`, or `member`, but user sent empty string or `abcdef`), you SHOULD NOT send an error code. Instead, prefer to "fix" errors (sanitize to correct format).
    - Rationale: Synchronization should be reliable, and should not fail other than transiently, or for serious programming errors. Otherwise, the user will have a permanently unsyncable app, and may have to log out/delete it and lose unsynced data. You don't want a bug 5 versions ago to create a persistently failing sync.
11. You SHOULD delete all descendants of deleted records
    - Frontend should ask the push endpoint to do so as well, but if it's buggy, you may end up with permanent orphans

Tips on implementing server-side changes tracking

If you're wondering how to *actually* implement consistent pulling of all changes since the last pull, or how to detect that a record being pushed by the user changed after `lastPulledAt`, here's what we recommend:

- Add a `last_modified` field to all your server database tables, and bump it to `NOW()` every time you create or update a record.
- This way, when you want to get all changes since `lastPulledAt`, you query records whose `last_modified > lastPulledAt`.
- The timestamp should be at least millisecond resolution, and you should add (for extra safety) a MySQL/PostgreSQL procedure that will ensure `last_modified` uniqueness and monotonicity
    - Specificaly, check that there is no record with a `last_modified` equal to or greater than `NOW()`, and if there is, increment the new timestamp by 1 (or however much you need to ensure it's the greatest number)
    - An example of this for PostgreSQL can be found in Kinto
    - This protects against weird edge cases - such as records being lost due to server clock time changes (NTP time sync, leap seconds, etc.)
- Of course, remember to ignore `last_modified` from the user if you do it this way.

- An alternative to using timestamps is to use an auto-incrementing counter sequence, but you must ensure that this sequence is consistent across all collections. You also leak to users the amount of traffic to your sync server (number of changes in the sequence)
- To distinguish between `created` and `updated` records, you can also store server-side `server_created_at` timestamp (if it's greater than `last_pulled_at` supplied to sync, then record is to be `created` on client, if less than — client already has it and it is to be `updated` on client). Note that this timestamp must be consistent with last_modified — and you must not use client-created `created_at` field, since you can never trust local timestamps.
  - Alternatively, you can send all non-deleted records as all `updated` and Watermelon will do the right thing in 99% of cases (you will be slightly less protected against weird edge cases — treatment of locally deleted records is different). If you do this, pass `sendCreatedAsUpdated: true` to `synchronize()` to supress warnings about records to be updated not existing locally.
- You do need to implement a mechanism to track when records were deleted on the server, otherwise you wouldn't know to push them
  - One possible implementation is to not fully delete records, but mark them as DELETED=true
  - Or, you can have a `deleted_xxx` table with just the record ID and timestamp (consistent with last_modified)
  - Or, you can treat it the same way as "revoked permissions"
- If you have a collaborative app with any sort of permissions, you also need to track granting and revoking of permissions the same way as changes to records
  - If permission to access records has been granted, the pull endpoint must add those records to `created`
  - If permission to access records has been revoked, the pull endpoint must add those records to `deleted`
  - Remember to also return all descendants of a record in those cases

Existing Backend Implementations

Note that those are not maintained by WatermelonDB, and we make no endorsements about quality of these projects:

- How to Build WatermelonDB Sync Backend in Elixir
- Firemelon
- Laravel Watermelon

Did you make one? Please contribute a link!

Edit this page

Previous
« Frontend

Next
Limitations »

Version: 0.27.1

Limitations

1. If a record being pushed changes remotely between pull and push, push will just fail. It would be better if it failed with a list of conflicts, so that `synchronize()` can automatically respond. Alternatively, sync could only send changed fields and server could automatically always just apply those changed fields to the server version (since that's what per-column client-wins resolver will do anyway)
2. During next sync pull, changes we've just pushed will be pulled again, which is unnecessary. It would be better if server, during push, also pulled local changes since `lastPulledAt` and responded with NEW timestamp to be treated as `lastPulledAt`.
3. It shouldn't be necessary to push the whole updated record — just changed fields + ID should be enough

   > Note: That might conflict with "If client wants to update a record that doesn't exist, create it"

Don't like these limitations? Good, neither do we! Please contribute - we'll give you guidance.

Edit this page

Previous
« Backend

Next
FAQ »

Frequently Asked Questions

Sync primitives and implementing your own sync entirely from scratch

See: Sync implementation details

Local vs Remote IDs

WatermelonDB has been designed with the assumption that there is no difference between Local IDs (IDs of records and their relations in a WatermelonDB database) and Remote IDs (IDs on the backend server). So a local app can create new records, generating their IDs, and the backend server will use this ID as the true ID. This greatly simplifies synchronization, as you don't have to replace local with remote IDs on the record and all records that point to it.

We highly recommend that you adopt this practice.

Some people are skeptical about this approach due to conflicts, since backend can guarantee unique IDs, and the local app can't. However, in practice, a standard Watermelon ID has 8,000,000,000,000,000,000,000,000 possible combinations. That's enough entropy to make conflicts extremely unlikely. At Nozbe, we've done it this way at scale for more than 15 years, and not once did we encounter a genuine ID conflict or had other issues due to this approach.

> Using the birthday problem, we can calculate that for 36^16 possible IDs, if your system grows to a billion records, the probability of a single conflict is 6e-8. At 100B records, the probability grows to 0.06%. But if you grow to that many records, you're probably a very rich company and can start worrying about things like this *then*.

If you absolutely can't adopt this practice, there's a number of production apps using WatermelonDB that keep local and remote IDs separate — however, more work is required this way. Search Issues to find discussions about this topic — and consider contributing to WatermelonDB to make managing separate local IDs easier for everyone!

Edit this page

Previous
« Limitations

Next
Troubleshoot »

Docs

Installation

Contributing

Community

Stack Overflow

Twitter

More

GitHub

WatermelonDB by Radek Pietruszewski and Nozbe.

Version: 0.27.1

Troubleshoot

⚠ **Note about a React Native / UglifyES bug**. When you import Watermelon Sync, your app might fail to compile in release mode. To fix this, configure Metro bundler to use Terser instead of UglifyES. Run:

```
yarn add metro-minify-terser
```

Then, update `metro.config.js`:

```
module.exports = {
    // ...
    transformer: {
        // ...
        minifierPath: 'metro-minify-terser',
    },
}
```

You might also need to switch to Terser in Webpack if you use Watermelon for web.

Edit this page

Various Pro Tips

Database viewer

See discussion

Android - you can use the new App Inspector in modern versions of Android Studio.

Via Flipper You can also use Facebook Flipper with a plugin. See discussion.

iOS - check open database path in iOS System Log (via Console for plugged-in device, or Xcode logs, or by using `find`), then open it via `sqlite3` in the console, or an external tool like sqlitebrowser

Which SQLite version am I using?

This usually only matters if you use raw SQL to use new SQLite versions:

- On iOS, we use whatever SQLite version is bundled with the OS. Here's a table of iOS version - SQLite version matches
- On Android in JSI mode, we use SQLite bundled with WatermelonDB. See `@nozbe/sqlite` NPM dependency version to see which SQLite version is bundled.
- On Android NOT in JSI mode, we use the SQLite bundled with the OS

BTW: We're happy to accept contributions so that you can choose custom version or build of SQLite in all modes and on all platforms, but it needs to be opt-in (this adds to build time and binary size and most people don't need this)

Prepopulating database on native

There's no built-in support for this. One way is to generate a SQLite DB (you can use the the Node SQLite support in 0.19.0-2 pre-release or extract it from an ios/android app), bundle it with the app, and then use a bit of code to check if the DB you're expecting it available, and if not, making a copy of the default DB — before you attempt loading DB from JS side. See discussion

Override entity ID generator

You can optionally overide WatermelonDB's id generator with your own custom id generator in order to create specific random id formats (e.g. if UUIDs are used in the backend). In your database index file, pass a function with your custom ID generator to `setGenerator`:

```
// Define a custom ID generator.
function randomString(): string {
  return 'RANDOM STRING';
}
setGenerator(randomString);

// or as anonymous function:
setGenerator(() => 'RANDOM STRING');
```

To get UUIDs specifically, install uuid and then pass their id generator to `setGenerator`:

```
import { v4 as uuidv4 } from 'uuid';

setGenerator(() => uuidv4());
```

Edit this page

Previous
« LocalStorage

Next
Performance Tips »

Architecture

Base objects

`Database` is the root object of Watermelon. It owns:

- a `DatabaseAdapter`
- a map of `Collection`s

`DatabaseAdapter` connects Watermelon's reactive world to low-level imperative world of databases. See Adapters.

`Collection` manages all records of a given kind:

- it has a cache of records already fetched from the database (`RecordCache`)
- it has the public API to `find`, `query` and `create` existing records
- it implements fetch/update/delete operations on records

`Model` is an instance of a collection record. A model *class* describes a *kind* of a record. `Model` is the base class for your concrete models (e.g. `Post`, `Comment`, `Task`):

- it describes the specific instance - `id` + all custom fields and actions
- it has public API to `update`, `markAsDeleted` and `destroyPermanently`
- implements record-level observation `observe()`
- static fields describe base information about a model (`table`, `associations`) - See Defining models

As a general rule, `Model` manages the state of a specific instance, and `Collection` of the entire collection of records. So for example, `model.markAsDeleted()` changes the local state of called record, but then delegates to its collection to notify collection observers and actually remove from the database

`Query` is a helper object that gives us a nice API to perform queries (`query.observe()`, `query.fetchCount()`):

- created via `collection.query()`
- encapsulates a `QueryDescription` structure which actually describes the query conditions
- fetch/observe methods actually delegate to `Collection` to perform database operations
- caches `Observable`s created by `observe/observeCount` methods so they can be reused and shared

Helper functions

Watermelon's objects and classes are meant to be as minimal as possible — only manage their own state and be an API for your app. Most logic should be stateless, and implemented as pure functions:

`QueryDescription` is a structure (object) describing the query, built using `Q.*` helper functions

`encodeMatcher()`, `simpleObserver()`, `reloadingObserver()`, `fieldObserver()` implement query observation logic.

Model decorators transform simple class properties into Watermelon-aware record fields.

Much of Adapters' logic is implemented as pure functions too. See Adapters.

Edit this page

Previous
« iOS - Sharing database across targets

Next
Database Adapters »

Changelog

All notable changes to this project will be documented in this file.

Contributors: Please add your changes to CHANGELOG-Unreleased.md

0.27.1 - 2023-10-15

Fix missing Changelog for 0.27 release

0.27 - 2023-08-29

Highlights

Removed legacy Swift and Kotlin React Native Modules

Following the addition of new Native Modules in 0.26, we're removing the old implementations. We expect this to simplify installation process and remove a ton of compatibility and configuration issues due to Kotlin version mismatchs and the CocoaPods-Swift issues when using `use_frameworks!` or Expo.

Experimental React Native Windows support

WatermelonDB now has *experimental* support for React Native Windows. See Installation docs for details.

Introducing Watermelon React

All React/React Native helpers for Watermelon are now available from a new `@nozbe/watermelodb/react` folder:

- `DatabaseProvider`, `useDatabase`, `withDatabase`
- NEW: `withObservables` - `@nozbe/with-observables` as a separate package is deprecated, and is now bundled with WatermelonDB
- NEW: HOC helpers: `compose`, `withHooks`
- NEW: `<WithObservables />` component, a component version of `withObservables` HOC. Useful when a value being observed is localized to a small part of a larger component, because you can effortlessly narrow down which parts of the component are re-rendered when the value changes without having to extract a new component.

Imports from previous `@nozbe/watermelondb/DatabaseProvider` and `@nozbe/watermelondb/hooks` folders are deprecated and will be removed in a future version.

Introducing Watermelon Diagnostics

All debug/dev/diagnostics tools for Watermelon are now available from a new `@nozbe/watermelondb/diagnostics` folder:

- NEW: `censorRaw` - takes a `RawRecord/DirtyRaw` and censors its string values, while preserving IDs, _status, _changed, and numeric/boolean values. Helpful when viewing database contents in context that could expose private user information
- NEW: `diagnoseDatabaseStructure` - analyzes database to find inconsistencies, such as orphaned records (`belongs_to` relations on model that point to records that don't exist) or broken LokiJS database. Use this to find bugs in your data model.
- NEW: `diagnoseSyncConsistency` - compares local database with the server version (contents of first/full sync) to find inconsistencies, missing and excess records. Use this to find bugs in your backend sync implementation.

BREAKING CHANGES

- `@nozbe/with-observables` is no longer a WatermelonDB dependency. Change your imports to `import { withObservables } from '@nozbe/watermelondb/react'`

Changes unlikely to cause issues:

- [iOS] If `import WatermelonDB` is used in your Swift app (for Turbo sync), remove it and replace with `#import <WatermelonDB/WatermelonDB.h>` in the bridging header
- [iOS] If you use `_watermelonDBLoggingHook`, remove it. No replacement is provided at this time, feel free to contribute if you need this
- [iOS] If you use `-DENABLE_JSLOCK_PERFORMANCE_HACK`, remove it. JSLockPerfHack has been non-functional for some time already, and has now been removed. Please file an issue if you relied on it.

Deprecations

- Imports from `@nozbe/watermelondb/DatabaseProvider` and `@nozbe/watermelondb/hooks`. Change to `@nozbe/watermelondb/react`

New features

- New `@experimentalFailsafe` decorator you can apply before `@relation/@immutableRelation` so that if relation points to a record that does not exist, `.fetch()/.observe()` yield `undefined` instead of throwing an error

Fixes

- [Flow/TS] Improved typing of DatabaseContext
- Fixed `Cannot read property 'getRandomIds' of null`. This error occured if native modules were not

correctly installed, however the location of the error caused a lot of confusion.

0.26 - 2023-04-28

Highlights

New Native Modules

We're transitioning SQLite adapters for React Native **from Kotlin and Swift to Java and Objective-C**.

This is only a small part of WatermelonDB, yet is responsible for a disproportionate amount of issues raised, such as Kotlin version conflicts, Expo build failures, CocoaPods use_frameworks! issues. It makes library installation and updates more complicated for users. It complicates maintenance. Swift doesn't play nicely with either React Native's legacy Native Module system, nor can it interact cleanly with C++ (JSI/New Architecture) without going through Objective-C++.

In other words, in the context of a React Native library, the benefit of these modern, nicer to use languages is far outweighed by the downsides. That's why we (@radex & @rozpierog) decided to rewrite the iOS and Android implementations to Objective-C and Java respectively.

0.26 is a transition release, and it contains both implementations. If you find a regression caused by the new bridge, pass `{disableNewBridge: true}` to `new SQLiteAdapter()` **and file an issue**. We plan to remove the old implementation in 0.27 or 0.28 release.

New documentation

We have a brand new documentation page, built with Docusaurus (contributed by @ErickLuizA).

We plan to expand guides, add typing to examples, and add a proper API reference, but we need your help to do this! See: https://github.com/Nozbe/WatermelonDB/issues/1481

BREAKING CHANGES

- [iOS] You should remove import of WatermelonDB's `SupportingFiles/Bridging.h` from your app project's `Bridging.h`. If this removal causes build issues, please file an issue.
- [iOS] In your Podfile, replace previous WatermelonDB's pod imports with this:

```
# Uncomment this line if you're not using auto-linking
# pod 'WatermelonDB', path: '../node_modules/@nozbe/watermelondb'
# WatermelonDB dependency
pod 'simdjson', path: '../node_modules/@nozbe/simdjson', modular_headers: true
```

- Removed functions deprecated for 2+ years:
  - `Collection.unsafeFetchRecordsWithSQL()`. Use `.query(Q.unsafeSqlQuery('select * from...')).fetch()` instead.
  - `Database.action()`. Use `Database.write()` instead.

- `.subAction()`. Use `.callWriter()` instead.
- `@action` decorator. Use `@writer` instead.

Deprecations

New features

- [Android] Added `experimentalUnsafeNativeReuse` option to SQLiteAdapter. See `src/adapters/sqlite/type.js` for more details
- You can now pass an array to `Q.and(conditions)`, `Q.or(conditions)`, `collection.query(conditions)`, `query.extend(conditions)` in addition to spreading multiple arguments
- Added JSDoc comments to many APIs

Fixes

- Improved resiliency to "Maximum call stack size exceeded" errors
- [JSI] Improved reliability when reloading RCTBridge
- [iOS] Fix "range of supported deployment targets" Xcode warning
- `randomId` uses better randum number generator
- Fixed "no such index" when using non-standard schemas and >1k bulk updates
- Fixes and changes included in `@nozbe/with-observables@1.5.0`
- [Flow] `query.batch([model, falsy])` no longer raises an error

Performance

- Warning is now given if a large number of arguments is passed to `Q.and, Q.or, Collection.query, Database.batch` instead of a single array
- `randomId()` is now 2x faster on Chrome, 10x faster on Safari, 2x faster on iOS (Hermes)

Changes

- `randomId`: now also generates upper-case letters
- Simplified CocoaPods/iOS integration
- Docs improvements: SQLite versions, Flow declarations, Installation
- Improved diagnostic warnings and errors: JSI, Writer/Reader
- Remove old diagnostic warnings no longer relevant: `multiple Q.on()s`, `Database`, `LokiJSAdapter`, `SQLiteAdapter`
- Updated `flow-bin` to 0.200. This shouldn't have an impact on you, but could fix or break Flow if you don't have WatermelonDB set to `[declarations]` mode
- Updated `@babel/runtime` to 7.20.13
- Updated `rxjs` to 7.8.0
- Updated `sqlite` (SQLite used on Android in JSI mode) to 3.40.1
- Updated `simdjson` to 3.1.0

Internal

- Cleaned up QueryDescription, ios folder structure, JSI implementation by splitting them into smaller parts.
- [Android][jsi] Simplify CMakeLists
- Improve release script

0.25.5 - 2023-02-01

- Fix Android auto-linking

0.25.4 - 2023-01-31

- [Sync] Improve memory consumption (less likely to get "Maximum callstack exceeded" error)
- [TypeScript] Fix type of `DirtyRaw` to `{ [key: string]: any }` (from `Object`)

0.25.3 - 2023-01-30

- Fixed TypeError regression

0.25.2 - 2023-01-30

Fixes

- Fix TypeScript issues (@paulrostorp feat. @enahum)
- Fix compilation on Kotlin 1.7
- Fix regression in Sync that could cause `Record ID xxx#yyy was sent over the bridge, but it's not cached` error

Internal

- Update internal dependencies
- Fix Android CI
- Improve TypeScript CI

0.25.1 - 2023-01-23

- Fix React Native 0.71+ Android broken build

0.25 - 2023-01-20

Highlights

- Fix broken build on React Native 0.71+
- [Expo] Fixes Expo SDK 44+ build errors (@Kudo)
- [JSI] Fix an issue that sometimes led to crashing app upon database close

BREAKING CHANGES

- [Query] `Q.where(xxx, undefined)` will now throw an error. This is a bug fix, since comparing to undefined was never allowed and would either error out or produce a wrong result in some cases. However, it could technically break an app that relied on existing buggy behavior
- [JSI+Swift] If you use `watermelondbProvideSyncJson()` native iOS API, you might need to add `import WatermelonDB`

New features

- [adapters] Adapter objects can now be distinguished by checking their `static adapterType`
- [Query] New `Q.includes('foo')` query for case-sensitive exact string includes comparison
- [adapters] Adapter objects now returns `dbName`
- [Sync] Replacement Sync - a new advanced sync feature. Server can now send a full dataset (same as during initial sync) and indicate with `{ experimentalStrategy: 'replacement' }` that instead of applying a diff, local database should be replaced with the dataset sent. Local records not present in the changeset will be deleted. However, unlike clearing database and logging in again, unpushed local changes (to records that are kept after replacement) are preserved. This is useful for recovering from a corrupted local database, or as a hack to deal with very large state changes such that server doesn't know how to efficiently send incremental changes and wants to send a full dataset instead. See docs for more details.
- [Sync] Added `onWillApplyRemoteChanges` callback

Performance

- [LokiJS] Updated Loki with some performance improvements
- [iOS] JSLockPerfHack now works on iOS 15
- [Sync] Improved performance of processing large pulls
- Improved `@json` decorator, now with optional `{ memo: true }` parameter

Changes

- [Docs] Added additional Android JSI installation step

Fixes

- [TypeScript] Improve typings: add unsafeExecute method, localStorage property to Database
- [android] Fixed compilation on some setups due to a missing `<cassert>` import
- [sync] Fixed marking changes as synced for users that don't keep globally unique (only per-table unique)

IDs

- Fix `Model.experimentalMarkAsDeleted/experimentalDestroyPermanently()` throwing an error in some cases
- Fixes included in updated `withObservables`

0.24 - 2021-10-28

BREAKING CHANGES

- `Q.experimentalSortBy`, `Q.experimentalSkip`, `Q.experimentalTake` have been renamed to `Q.sortBy`, `Q.skip`, `Q.take` respectively
- **RxJS has been updated to 7.3.0**. If you're not importing from `rxjs` in your app, this doesn't apply to you. If you are, read RxJS 7 breaking changes: https://rxjs.dev/deprecations/breaking-changes

New features

- **LocalStorage**. `database.localStorage` is now available
- **sortBy, skip, take** are now available in LokiJSAdapter as well
- **Disposable records**. Read-only records that cannot be saved in the database, updated, or deleted and only exist for as long as you keep a reference to them in memory can now be created using `collection.disposableFromDirtyRaw()`. This is useful when you're adding online-only features to an otherwise offline-first app.
- [Sync] `experimentalRejectedIds` parameter now available in push response to allow partial rejection of an otherwise successful sync

Fixes

- Fixes an issue when using Headless JS on Android with JSI mode enabled - pass `usesExclusiveLocking: true` to SQLiteAdapter to enable
- Fixes Typescript annotations for Collection and adapters/sqlite

0.23 - 2021-07-22

This is a big release to WatermelonDB with new advanced features, great performance improvements, and important fixes to JSI on Android.

Please don't get scared off the long list of breaking changes - they are all either simple Find&Replace renames or changes to internals you probably don't use. It shouldn't take you more than 15 minutes to upgrade to 0.23.

BREAKING CHANGES

- **iOS Installation change**. You need to add this line to your Podfile: `pod 'simdjson', path: '../node_modules/@nozbe/simdjson'`

- Deprecated `new Database({ actionsEnabled: false })` options is now removed. Actions are always enabled.
- Deprecated `new SQLiteAdapter({ synchronous: true })` option is now removed. Use `{ jsi: true }` instead.
- Deprecated `Q.unsafeLokiFilter` is now removed. Use `Q.unsafeLokiTransform((raws, loki) => raws.filter(raw => ...))` instead.
- Deprecated `Query.hasJoins` is now removed
- Changes to `LokiJSAdapter` constructor options:
  - `indexedDBSerializer` -> `extraIncrementalIDBOptions: { serializeChunk, deserializeChunk }`
  - `onIndexedDBFetchStart` -> `extraIncrementalIDBOptions: { onFetchStart }`
  - `onIndexedDBVersionChange` -> `extraIncrementalIDBOptions: { onversionchange }`
  - `autosave: false` -> `extraLokiOptions: { autosave: false }`
- Changes to Internal APIs. These were never meant to be public, and so are unlikely to affect you:
  - `Model._isCommited`, `._hasPendingUpdate`, `._hasPendingDelete` have been removed and changed to `Model._pendingState`
  - `Collection.unsafeClearCache()` is no longer exposed
- Values passed to `adapter.setLocal()` are now validated to be strings. This is technically a bug fix, since local storage was always documented to only accept strings, however applications may have relied on this lack of validation. Adding this validation was necessary to achieve consistent behavior between SQLiteAdapter and LokiJSAdapter
- `unsafeSql` passed to `appSchema` will now also be called when dropping and later recreating all database indices on large batches. A second argument was added so you can distinguish between these cases. See Schema docs for more details.
- **Changes to sync change tracking**. The behavior of `record._raw._changed` and `record._raw._status` (a.k.a. `record.syncStatus`) has changed. This is unlikely to be a breaking change to you, unless you're writing your own sync engine or rely on these low-level details.
  - Previously, _changed was always empty when _status=created. Now, _changed is not populated during initial creation of a record, but a later update will add changed fields to _changed. This change was necessary to fix a long-standing Sync bug.

Deprecations

- `database.action(() => {})` is now deprecated. Use `db.write(() => {})` instead (or `db.read(() => {})` if you only need consistency but are not writing any changes to DB)
- `@action` is now deprecated. Use `@writer` or `@reader` instead
- `.subAction()` is now deprecated. Use `.callReader()` or `.callWriter()` instead
- `Collection.unsafeFetchRecordsWithSQL()` is now deprecated. Use `collection.query(Q.unsafeSqlQuery("select * from...")).fetch()` instead.

New features

- `db.write(writer => { ... writer.batch() })` - you can now call batch on the interface passed to a writer block
- **Fetching record IDs and unsafe raws.** You can now optimize fetching of queries that only require IDs, not full cached records:

    - `await query.fetchIds()` will return an array of record ids
    - `await query.unsafeFetchRaw()` will return an array of unsanitized, unsafe raw objects (use alongside `Q.unsafeSqlQuery` to exclude unnecessary or include extra columns)
    - advanced `adapter.queryIds()`, `adapter.unsafeQueryRaw` are also available
- **Raw SQL queries**. New syntax for running unsafe raw SQL queries:
    - `collection.query(Q.unsafeSqlQuery("select * from tasks where foo = ?", ['bar'])).fetch()`
    - You can now also run `.fetchCount()`, `.fetchIds()` on SQL queries
    - You can now safely pass values for SQL placeholders by passing an array
    - You can also observe an unsafe raw SQL query -- with some caveats! refer to documentation for more details
- **Unsafe raw execute**. You can now execute arbitrary SQL queries (SQLiteAdapter) or access Loki object directly (LokiJSAdapter) using `adapter.unsafeExecute` -- see docs for more details
- **Turbo Login**. You can now speed up the initial (login) sync by up to 5.3x with Turbo Login. See Sync docs for more details.
- New diagnostic tool - **debugPrintChanges**. See Sync documentation for more details

Performance

- The order of Q. clauses in a query is now preserved - previously, the clauses could get rearranged and produce a suboptimal query
- [SQLite] `adapter.batch()` with large numbers of created/updated/deleted records is now between 16-48% faster
- [LokiJS] Querying and finding is now faster - unnecessary data copy is skipped
- [jsi] 15-30% faster querying on JSC (iOS) when the number of returned records is large
- [jsi] up to 52% faster batch creation (yes, that's on top of the improvement listed above!)
- Fixed a performance bug that caused observed items on a list observer with `.observeWithColumns()` to be unnecessarily re-rendered just before they were removed from the list

Changes

- All Watermelon console logs are prepended with a 🍉 tag
- Extra protections against improper use of writers/readers (formerly actions) have been added
- Queries with multiple top-level `Q.on('table', ...)` now produce a warning. Use `Q.on('table', [condition1, condition2, ...])` syntax instead.
- [jsi] WAL mode is now used

Fixes

- [jsi] Fix a race condition where commands sent to the database right after instantiating SQLiteAdapter would fail
- [jsi] Fix incorrect error reporting on some sqlite errors
- [jsi] Fix issue where app would crash on Android/Hermes on reload
- [jsi] Fix IO errors on Android
- [sync] Fixed a long-standing bug that would cause records that are created before a sync and updated

during sync's push to lose their most recent changes on a subsequent sync

Internal

- Internal changes to SQLiteAdapter:
  - .batch is no longer available on iOS implementation
  - .batch/.batchJSON internal format has changed
  - .getDeletedRecords, destroyDeletedRecords, setLocal, removeLocal is no longer available
- encoded SQLiteAdapter schema has changed
- LokiJSAdapter has had many internal changes

0.22 - 2021-05-07

BREAKING CHANGES

- [SQLite] `experimentalUseJSI: true` option has been renamed to `jsi: true`

Deprecations

- [LokiJS] `Q.unsafeLokiFilter` is now deprecated and will be removed in a future version. Use `Q.unsafeLokiTransform((raws, loki) => raws.filter(raw => ...))` instead.

New features

- [SQLite][JSI] `jsi: true` now works on Android - see docs for installation info

Performance

- Removed dependency on rambdax and made the util library smaller
- Faster withObservables

Changes

- Synchronization: `pushChanges` is optional, will not calculate local changes if not specified.
- withObservables is now a dependency of WatermelonDB for simpler installation and consistent updates. You can (and generally should) delete `@nozbe/with-observables` from your app's package.json
- [Docs] Add advanced tutorial to share database across iOS targets - @thiagobrez
- [SQLite] Allowed callbacks (within the migrationEvents object) to be passed so as to track the migration events status (onStart, onSuccess, onError) - @avinashlng1080
- [SQLite] Added a dev-only `Query._sql()` method for quickly extracting SQL from Queries for debugging purposes

Fixes

- Non-react statics hoisting in `withDatabase()`

- Fixed incorrect reference to `process`, which can break apps in some environments (e.g. webpack5)
- [SQLite][JSI] Fixed JSI mode when running on Hermes
- Fixed a race condition when using standard fetch methods alongside `Collection.unsafeFetchRecordsWithSQL` - @jspizziri
- withObservables shouldn't cause any RxJS issues anymore as it no longer imports RxJS
- [Typescript] Added `onSetUpError` and `onIndexedDBFetchStart` fields to `LokiAdapterOptions`; fixes TS error - @3DDario
- [Typescript] Removed duplicated identifiers `useWebWorker` and `useIncrementalIndexedDB` in `LokiAdapterOptions` - @3DDario
- [Typescript] Fix default export in logger util

0.21 - 2021-03-24

BREAKING CHANGES

- [LokiJS] `useWebWorker` and `useIncrementalIndexedDB` options are now required (previously, skipping them would only trigger a warning)

New features

- [Model] `Model.update` method now returns updated record
- [adapters] `onSetUpError: Error => void` option is added to both `SQLiteAdapter` and `LokiJSAdapter`. Supply this option to catch initialization errors and offer the user to reload or log out
- [LokiJS] new `extraLokiOptions` and `extraIncrementalIDBOptions` options
- [Android] Autolinking is now supported.
  - If You upgrade to `<= v0.21.0` **AND** are on a version of React Native which supports Autolinking, you will need to remove the config manually linking WatermelonDB.
  - You can resolve this issue by **REMOVING** the lines of config from your project which are *added* in the `Manual Install ONLY` section of the Android Install docs.

Performance

- [LokiJS] Improved performance of launching the app

Changes

- [LokiJS] `useWebWorker: true` and `useIncrementalIndexedDB: false` options are now deprecated. If you rely on these features, please file an issue!
- [Sync] Optional `log` passed to sync now has more helpful diagnostic information
- [Sync] Open-sourced a simple SyncLogger you can optionally use. See docs for more info.
- [SQLiteAdapter] `synchronous:true` option is now deprecated and will be replaced with `experimentalUseJSI: true` in the future. Please test if your app compiles and works well with `experimentalUseJSI: true`, and if not - file an issue!
- [LokiJS] Changed default autosave interval from 250 to 500ms

- [Typescript] Add `experimentalNestedJoin` definition and `unsafeSqlExpr` clause

Fixes

- [LokiJS] Fixed a case where IndexedDB could get corrupted over time
- [Resilience] Added extra diagnostics for when you encounter the `Record ID aa#bb was sent over the bridge, but it's not cached` error and a recovery path (LokiJSAdapter-only). Please file an issue if you encounter this issue!
- [Typescript] Fixed type on OnFunction to accept `and` in join
- [Typescript] Fixed type `database#batch(records)`'s argument `records` to accept mixed types

Internal

- Added an experimental mode where a broken database state is detected, further mutations prevented, and the user notified

0.20 - 2020-10-05

BREAKING CHANGES

This release has unintentionally broken RxJS for some apps using `with-observables`. If you have this issue, please update `@nozbe/with-observables` to the latest version.

New features

- [Sync] Conflict resolution can now be customized. See docs for more details
- [Android] Autolinking is now supported
- [LokiJS] Adapter autosave option is now configurable

Changes

- Interal RxJS imports have been refactor such that rxjs-compat should never be used now
- [Performance] Tweak Babel config to produce smaller code
- [Performance] LokiJS-based apps will now take up to 30% less time to load the database (id and unique indicies are generated lazily)

Fixes

- [iOS] Fixed crash on database reset in apps linked against iOS 14 SDK
- [LokiJS] Fix `Q.like` being broken for multi-line strings on web
- Fixed warn "import cycle" from DialogProvider (#786) by @gmonte.
- Fixed cache date as instance of Date (#828) by @djorkaeffalexandre.

0.19 - 2020-08-17

New features

- [iOS] Added CocoaPods support - @leninlin
- [NodeJS] Introducing a new SQLite Adapter based integration to NodeJS. This requires a peer dependency on better-sqlite3 and should work with the same configuration as iOS/Android - @sidferreira
- [Android] `exerimentalUseJSI` option has been enabled on Android. However, it requires some app-specific setup which is not yet documented - stay tuned for upcoming releases
- [Schema][Migrations] You can now pass `unsafeSql` parameters to schema builder and migration steps to modify SQL generated to set up the database or perform migrations. There's also new `unsafeExecuteSql` migration step. Please use this only if you know what you're doing — you shouldn't need this in 99% of cases. See Schema and Migrations docs for more details
- [LokiJS][Performance] Added experimental `onIndexedDBFetchStart` and `indexedDBSerializer` options to `LokiJSAdapter`. These can be used to improve app launch time. See `src/adapters/lokijs/index.js` for more details.

Changes

- [Performance] findAndObserve is now able to emit a value synchronously. By extension, this makes Relations put into withObservables able to render the child component in one shot. Avoiding the extra unnecessary render cycles avoids a lot of DOM and React commit-phase work, which can speed up loading some views by 30%
- [Performance] LokiJS is now faster (refactored encodeQuery, skipped unnecessary clone operations)

0.18 - 2020-06-30

Another WatermelonDB release after just a week? Yup! And it's jam-packed full of features!

New features

- [Query] `Q.on` queries are now far more flexible. Previously, they could only be placed at the top level of a query. See Docs for more details. Now, you can:
  - Pass multiple conditions on the related query, like so:

    ```
    collection.query(Q.on('projects', [Q.where('foo', 'bar'), Q.where('bar', 'baz')]))
    ```

  - You can place `Q.on` deeper inside the query (nested inside `Q.and()`, `Q.or()`). However, you must explicitly list all tables you're joining on at the beginning of a query, using: `Q.experimentalJoinTables(['join_table1', 'join_table2'])`.
  - You can nest `Q.on` conditions inside `Q.on`, e.g. to make a condition on a grandchild. To do so, it's required to pass `Q.experimentalNestedJoin('parent_table', 'grandparent_table')` at the

beginning of a query

- [Query] `Q.unsafeSqlExpr()` and `Q.unsafeLokiExpr()` are introduced to allow adding bits of queries that are not supported by the WatermelonDB query language without having to use `unsafeFetchRecordsWithSQL()`. See docs for more details

- [Query] `Q.unsafeLokiFilter((rawRecord, loki) => boolean)` can now be used as an escape hatch to make queries with LokiJSAdapter that are not otherwise possible (e.g. multi-table column comparisons). See docs for more details

Changes

- [Performance][LokiJS] Improved performance of queries containing query comparisons on LokiJSAdapter
- [Docs] Added Contributing guide for Query language improvements
- [Deprecation] `Query.hasJoins` is deprecated
- [DX] Queries with bad associations now show more helpful error message
- [Query] Counting queries that contain `Q.experimentalTake` / `Q.experimentalSkip` is currently broken - previously it would return incorrect results, but now it will throw an error to avoid confusion. Please contribute to fix the root cause!

Fixes

- [Typescript] Fixed types of Relation

Internal

- `QueryDescription` structure has been changed.

0.17.1 - 2020-06-24

- Fixed broken iOS build - @mlecoq

0.17 - 2020-06-22

New features

- [Sync] Introducing Migration Syncs - this allows fully consistent synchronization when migrating between schema versions. Previously, there was no mechanism to incrementally fetch all remote changes in new tables and columns after a migration - so local copy was likely inconsistent, requiring a re-login. After adopting migration syncs, Watermelon Sync will request from backend all missing information. See Sync docs for more details.

- [iOS] Introducing a new native SQLite database integration, rewritten from scratch in C++, based on React Native's JSI (JavaScript Interface). It is to be considered experimental, however we intend to make it the default (and eventually, the only) implementation. In a later release, Android version will be introduced.

```
The new adapter is up to 3x faster than the previously fastest `synchronous: true` option,
however this speedup is only achieved with some unpublished React Native patches.

To try out JSI, add `experimentalUseJSI: true` to `SQLiteAdapter` constructor.
```

- [Query] Added `Q.experimentalSortBy(sortColumn, sortOrder)`, `Q.experimentalTake(count)`, `Q.experimentalSkip(count)` methods (only availble with SQLiteAdapter) - @Kenneth-KT
- `Database.batch()` can now be called with a single array of models
- [DX] `Database.get(tableName)` is now a shortcut for `Database.collections.get(tableName)`
- [DX] Query is now thenable - you can now use `await query` and `await query.count` instead of `await query.fetch()` and `await query.fetchCount()`
- [DX] Relation is now thenable - you can now use `await relation` instead of `await relation.fetch()`
- [DX] Exposed `collection.db` and `model.db` as shortcuts to get to their Database object

Changes

- [Hardening] Column and table names starting with `__`, Object property names (e.g. `constructor`), and some reserved keywords are now forbidden
- [DX][Hardening] QueryDescription builder methods do tighter type checks, catching more bugs, and preventing users from unwisely passing unsanitized user data into Query builder methods
- [DX][Hardening] Adapters check early if table names are valid
- [DX] Collection.find reports an error more quickly if an obviously invalid ID is passed
- [DX] Intializing Database with invalid model classes will now show a helpful error
- [DX] DatabaseProvider shows a more helpful error if used improperly
- [Sync] Sync no longer fails if pullChanges returns collections that don't exist on the frontend - shows a warning instead. This is to make building backwards-compatible backends less error-prone
- [Sync][Docs] Sync documentation has been rewritten, and is now closer in detail to a formal specification
- [Hardening] database.collections.get() better validates passed value
- [Hardening] Prevents unsafe strings from being passed as column name/table name arguments in QueryDescription

Fixes

- [Sync] Fixed `RangeError: Maximum call stack size exceeded` when syncing large amounts of data - @leninlin
- [iOS] Fixed a bug that could cause a database operation to fail with an (6) SQLITE_LOCKED error
- [iOS] Fixed 'jsi/jsi.h' file not found when building at the consumer level. Added path `$(SRCROOT)/../../../../../ios/Pods/Headers/Public/React-jsi` to Header Search Paths (issue #691) - @victorbutler
- [Native] SQLite keywords used as table or column names no longer crash

- Fixed potential issues when subscribing to database, collection, model, queries passing a subscriber function with the same identity more than once

Internal

- Fixed broken adapter tests

0.15.1, 0.16.1-fix, 0.16.2 - 2020-06-03

This is a security patch for a vulnerability that could cause maliciously crafted record IDs to cause all or some of user's data to be deleted. More information available via GitHub security advisory

0.16.1 - 2020-05-18

Changes

- `Database.unsafeResetDatabase()` is now less unsafe — more application bugs are being caught

Fixes

- [iOS] Fix build in apps using Flipper
- [Typescript] Added type definition for `setGenerator`.
- [Typescript] Fixed types of decorators.
- [Typescript] Add Tests to test Types.
- Fixed typo in learn-to-use docs.
- [Typescript] Fixed types of changes.

Internal

- [SQLite] Infrastruture for a future JSI adapter has been added

0.16 - 2020-03-06

⚠ Breaking

- `experimentalUseIncrementalIndexedDB` has been renamed to `useIncrementalIndexedDB`

Low breakage risk

- [adapters] Adapter API has changed from returning Promise to taking callbacks as the last argument. This won't affect you unless you call on adapter methods directly. `database.adapter` returns a new `DatabaseAdapterCompat` which has the same shape as old adapter API. You can use `database.adapter.underlyingAdapter` to get back `SQLiteAdapter` / `LokiJSAdapter`
- [Collection] `Collection.fetchQuery` and `Collection.fetchCount` are removed. Please use

`Query.fetch()` and `Query.fetchCount()`.

New features

- [SQLiteAdapter][iOS] Add new `synchronous` option to adapter: `new SQLiteAdapter({ ..., synchronous: true })`. When enabled, database operations will block JavaScript thread. Adapter actions will resolve in the next microtask, which simplifies building flicker-free interfaces. Adapter will fall back to async operation when synchronous adapter is not available (e.g. when doing remote debugging)
- [LokiJS] Added new `onQuotaExceededError?: (error: Error) => void` option to `LokiJSAdapter` constructor. This is called when underlying IndexedDB encountered a quota exceeded error (ran out of allotted disk space for app) This means that app can't save more data or that it will fall back to using in-memory database only Note that this only works when `useWebWorker: false`

Changes

- [Performance] Watermelon internals have been rewritten not to rely on Promises and allow some fetch/observe calls to resolve synchronously. Do not rely on this -- external API is still based on Rx and Promises and may resolve either asynchronously or synchronously depending on capabilities. This is meant as a internal performance optimization only for the time being.
- [LokiJS][Performance] Improved worker queue implementation for performance
- [observation] Refactored observer implementations for performance

Fixes

- Fixed a possible cause for "Record ID xxx#yyy was sent over the bridge, but it's not cached" error
- [LokiJS] Fixed an issue preventing database from saving when using `experimentalUseIncrementalIndexedDB`
- Fixed a potential issue when using `database.unsafeResetDatabase()`
- [iOS] Fixed issue with clearing database under experimental synchronous mode

New features (Experimental)

- [Model] Added experimental `model.experimentalSubscribe((isDeleted) => { ... })` method as a vanilla JS alternative to Rx based `model.observe()`. Unlike the latter, it does not notify the subscriber immediately upon subscription.
- [Collection] Added internal `collection.experimentalSubscribe((changeSet) => { ... })` method as a vanilla JS alternative to Rx based `collection.changes` (you probably shouldn't be using this API anyway)
- [Database] Added experimental `database.experimentalSubscribe(['table1', 'table2'], () => { ... })` method as a vanilla JS alternative to Rx-based `database.withChangesForTables()`. Unlike the latter, `experimentalSubscribe` notifies the subscriber only once after a batch that makes a change in multiple collections subscribed to. It also doesn't notify the subscriber immediately upon subscription, and doesn't send details about the changes, only a signal.
- Added `experimentalDisableObserveCountThrottling()` to `@nozbe/watermelondb/observation/observeCount` that globally disables count observation throttling. We think that throttling on WatermelonDB level is not a good feature and will be removed in a future release - and will be better implemented on app level if necessary

- [Query] Added experimental `query.experimentalSubscribe(records => { ... })`, `query.experimentalSubscribeWithColumns(['col1', 'col2'], records => { ... })`, and `query.experimentalSubscribeToCount(count => { ... })` methods

0.15 - 2019-11-08

Highlights

This is a **massive** new update to WatermelonDB! 🍉

- **Up to 23x faster sync**. You heard that right. We've made big improvements to performance. In our tests, with a massive sync (first login, 45MB of data / 65K records) we got a speed up of:
  - 5.7s -> 1.2s on web (5x)
  - 142s -> 6s on iOS (23x)

  Expect more improvements in the coming releases!

- **Improved LokiJS adapter**. Option to disable web workers, important Safari 13 fix, better performance, and now works in Private Modes. We recommend adding `useWebWorker: false, experimentalUseIncrementalIndexedDB: true` options to the `LokiJSAdapter` constructor to take advantage of the improvements, but please read further changelog to understand the implications of this.

- **Raw SQL queries** now available on iOS and Android thanks to the community

- **Improved TypeScript support** — thanks to the community

⚠️ Breaking

- Deprecated `bool` schema column type is removed -- please change to `boolean`
- Experimental `experimentalSetOnlyMarkAsChangedIfDiffers(false)` API is now removed

New featuers

- [Collection] Add `Collection.unsafeFetchRecordsWithSQL()` method. You can use it to fetch record using raw SQL queries on iOS and Android. Please be careful to avoid SQL injection and other pitfalls of raw queries

- [LokiJS] Introduces new `new LokiJSAdapter({ ..., experimentalUseIncrementalIndexedDB: true })` option. When enabled, database will be saved to browser's IndexedDB using a new adapter that only saves the changed records, instead of the entire database.

  This works around a serious bug in Safari 13 (https://bugs.webkit.org/show_bug.cgi?id=202137) that causes large databases to quickly balloon to gigabytes of temporary trash

  This also improves performance of incremental saves, although initial page load or very, very large saves might be slightly slower.

This is intended to become the new default option, but it's not backwards compatible (if enabled, old database will be lost). **You're welcome to contribute an automatic migration code.**

Note that this option is still experimental, and might change in breaking ways at any time.

- [LokiJS] Introduces new `new LokiJSAdapter({ ..., useWebWorker: false })` option. Before, web workers were always used with `LokiJSAdapter`. Although web workers may have some performance benefits, disabling them may lead to lower memory consumption, lower latency, and easier debugging. YMMV.
- [LokiJS] Added `onIndexedDBVersionChange` option to `LokiJSAdapter`. This is a callback that's called when internal IDB version changed (most likely the database was deleted in another browser tab). Pass a callback to force log out in this copy of the app as well. Note that this only works when using incrementalIDB and not using web workers
- [Model] Add `Model._dangerouslySetRawWithoutMarkingColumnChange()` method. You probably shouldn't use it, but if you know what you're doing and want to live-update records from server without marking record as updated, this is useful
- [Collection] Add `Collection.prepareCreateFromDirtyRaw()`
- @json decorator sanitizer functions take an optional second argument, with a reference to the model

Fixes

- Pinned required `rambdax` version to 2.15.0 to avoid console logging bug. In a future release we will switch to our own fork of `rambdax` to avoid future breakages like this.

Improvements

- [Performance] Make large batches a lot faster (1.3s shaved off on a 65K insert sample)
- [Performance][iOS] Make large batch inserts an order of magnitude faster
- [Performance][iOS] Make encoding very large queries (with thousands of parameters) 20x faster
- [Performance][LokiJS] Make batch inserts faster (1.5s shaved off on a 65K insert sample)
- [Performance][LokiJS] Various performance improvements
- [Performance][Sync] Make Sync faster
- [Performance] Make observation faster
- [Performance][Android] Make batches faster
- Fix app glitches and performance issues caused by race conditions in `Query.observeWithColumns()`
- [LokiJS] Persistence adapter will now be automatically selected based on availability. By default, IndexedDB is used. But now, if unavailable (e.g. in private mode), ephemeral memory adapter will be used.
- Disabled console logs regarding new observations (it never actually counted all observations) and time to query/count/batch (the measures were wildly inaccurate because of asynchronicity - actual times are much lower)
- [withObservables] Improved performance and debuggability (update withObservables package separately)

- Improved debuggability of Watermelon -- shortened Rx stacks and added function names to aid in understanding call stacks and profiles
- [adapters] The adapters interface has changed. `query()` and `count()` methods now receive a `SerializedQuery`, and `batch()` now takes `TableName<any>` and `RawRecord` or `RecordId` instead of `Model`.
- [Typescript] Typing improvements
  - Added 3 missing properties `collections`, `database` and `asModel` in Model type definition.
  - Removed optional flag on `actionsEnabled` in the Database constructor options since its mandatory since 0.13.0.
  - fixed several further typing issues in Model, Relation and lazy decorator
- Changed how async functions are transpiled in the library. This could break on really old Android phones but shouldn't matter if you use latest version of React Native. Please report an issue if you see a problem.
- Avoid `database` prop drilling in the web demo

0.14.1 - 2019-08-31

Hotfix for rambdax crash

- [Schema] Handle invalid table schema argument in appSchema
- [withObservables] Added TypeScript support (changelog)
- [Electron] avoid `Uncaught ReferenceError: global is not defined` in electron runtime (#453)
- [rambdax] Replaces `contains` with `includes` due to `contains` deprecation https://github.com/selfrefactor/rambda/commit/1dc1368f81e9f398664c9d95c2efbc48b5cdff9b#diff-04c6e90faac2675aa89e2176d2eec7d8R2209

0.14.0 - 2019-08-02

New features

- [Query] Added support for `notLike` queries 🎉
- [Actions] You can now batch delete record with all descendants using experimental functions `experimentalMarkAsDeleted` or `experimentalDestroyPermanently`

0.13.0 - 2019-07-18

⚠️ Breaking

- [Database] It is now mandatory to pass `actionsEnabled:` option to Database constructor. It is recommended that you enable this option:

```
const database = new Database({
  adapter: ...,
  modelClasses: [...],
```

```
  actionsEnabled: true
})
```

See `docs/Actions.md` for more details about Actions. You can also pass `false` to maintain backward compatibility, but this option **will be removed** in a later version

- [Adapters] `migrationsExperimental` prop of `SQLiteAdapter` and `LokiJSAdapter` has been renamed to `migrations`.

New features

- [Actions] You can now batch deletes by using `prepareMarkAsDeleted` or `prepareDestroyPermanently`
- [Sync] Performance: `synchronize()` no longer calls your `pushChanges()` function if there are no local changes to push. This is meant to save unnecessary network bandwidth. ⚠ Note that this could be a breaking change if you rely on it always being called
- [Sync] When setting new values to fields on a record, the field (and record) will no longer be marked as changed if the field's value is the same. This is meant to improve performance and avoid unnecessary code in the app. ⚠ Note that this could be a breaking change if you rely on the old behavior. For now you can import `experimentalSetOnlyMarkAsChangedIfDiffers` from `@nozbe/watermelondb/Model/index` and call if with `(false)` to bring the old behavior back, but this will be removed in the later version -- create a new issue explaining why you need this
- [Sync] Small perf improvements

Improvements

- [Typescript] Improved types for SQLite and LokiJS adapters, migrations, models, the database and the logger.

0.12.3 - 2019-05-06

Changes

- [Database] You can now update the random id schema by importing `import { setGenerator } from '@nozbe/watermelondb/utils/common/randomId'` and then calling `setGenerator(newGenenerator)`. This allows WatermelonDB to create specific IDs for example if your backend uses UUIDs.
- [Typescript] Type improvements to SQLiteAdapter and Database
- [Tests] remove cleanup for react-hooks-testing-library@0.5.0 compatibility

0.12.2 - 2019-04-19

Fixes

- [TypeScript] 'Cannot use 'in' operator to search for 'initializer'; decorator fix

Changes

- [Database] You can now pass falsy values to `Database.batch(...)` (false, null, undefined). This is useful in keeping code clean when doing operations conditionally. (Also works with `model.batch(...)`)
- [Decorators]. You can now use `@action` on methods of any object that has a `database: Database` property, and `@field @children @date @relation @immutableRelation @json @text @nochange` decorators on any object with a `asModel: Model` property.
- [Sync] Adds a temporary/experimental `_unsafeBatchPerCollection: true` flag to `synchronize()`. This causes server changes to be committed to database in multiple batches, and not one. This is NOT preferred for reliability and performance reasons, but it works around a memory issue that might cause your app to crash on very large syncs (>20,000 records). Use this only if necessary. Note that this option might be removed at any time if a better solution is found.

0.12.1 - 2019-04-01

⚠️ Hotfix

- [iOS] Fix runtime crash when built with Xcode 10.2 (Swift 5 runtime).

  ⚠️ **Note**: You need to upgrade to React Native 0.59.3 for this to work. If you can't upgrade React Native yet, either stick to Xcode 10.1 or manually apply this patch: https://github.com/Nozbe/WatermelonDB/pull/302/commits/aa4e08ad0fa55f434da2a94407c51fc5ff18e506

Changes

- [Sync] Adds basic sync logging capability to Sync. Pass an empty object to `synchronize()` to populate it with diagnostic information:

  ```
  const log = {}
  await synchronize({ database, log, ...})
  console.log(log.startedAt)
  ```

  See Sync documentation for more details.

0.12.0 - 2019-03-18

Added

- [Hooks] new `useDatabase` hook for consuming the Database Context:

  ```
  import { useDatabase } from '@nozbe/watermelondb/hooks'
  const Component = () => {
    const database = useDatabase()
  }
  ```

- [TypeScript] added `.d.ts` files. Please note: TypeScript definitions are currently incomplete and should

be used as a guide only. **PRs for improvements would be greatly appreciated!**

Performance

- Improved UI performance by consolidating multiple observation emissions into a single per-collection batch emission when doing batch changes

0.11.0 - 2019-03-12

Breaking

- ⚠ Potentially BREAKING fix: a `@date` field now returns a Jan 1, 1970 date instead of `null` if the field's raw value is `0`. This is considered a bug fix, since it's unexpected to receive a `null` from a getter of a field whose column schema doesn't say `isOptional: true`. However, if you relied on this behavior, this might be a breaking change.
- ⚠ BREAKING: `Database.unsafeResetDatabase()` now requires that you run it inside an Action

Bug fixes

- [Sync] Fixed an issue where synchronization would continue running despite `unsafeResetDatabase` being called
- [Android] fix compile error for kotlin 1.3+

Other changes

- Actions are now aborted when `unsafeResetDatabase()` is called, making reseting database a little bit safer
- Updated demo dependencies
- LokiJS is now a dependency of WatermelonDB (although it's only required for use on the web)
- [Android] removed unused test class
- [Android] updated ktlint to `0.30.0`

0.10.1 - 2019-02-12

Changes

- [Android] Changed `compile` to `implementation` in Library Gradle file
  - ⚠ might break build if you are using Android Gradle Plugin <3.X
- Updated `peerDependency` `react-native` to `0.57.0`
- [Sync] Added `hasUnsyncedChanges()` helper method
- [Sync] Improved documentation for backends that can't distinguish between `created` and `updated` records
- [Sync] Improved diagnostics / protection against edge cases

- [iOS] Add missing `header search path` to support **ejected** expo project.
- [Android] Fix crash on android < 5.0
- [iOS] `SQLiteAdapter`'s `dbName` path now allows you to pass an absolute path to a file, instead of a name
- [Web] Add adaptive layout for demo example with smooth scrolling for iOS

0.10.0 - 2019-01-18

Breaking

- **BREAKING:** Table column `last_modified` is no longer automatically added to all database tables. If you don't use this column (e.g. in your custom sync code), you don't have to do anything. If you do, manually add this column to all table definitions in your Schema:

  ```
  { name: 'last_modified', type: 'number', isOptional: true }
  ```

  Don't bump schema version or write a migration for this.

New

- **Actions API**.

  This was actually released in 0.8.0 but is now documented. With Actions enabled, all create/update/delete/batch calls must be wrapped in an Action.

  To use Actions, call `await database.action(async () => { /* perform writes here */ }`, and in Model instance methods, you can just decorate the whole method with `@action`.

  This is necessary for Watermelon Sync, and also to enable greater safety and consistency.

  To enable actions, add `actionsEnabled: true` to `new Database({ ... })`. In a future release this will be enabled by default, and later, made mandatory.

  See documentation for more details.

- **Watermelon Sync Adapter** (Experimental)

  Added `synchronize()` function that allows you to easily add full synchronization capabilities to your Watermelon app. You only need to provide two fetch calls to your remote server that conforms to Watermelon synchronization protocol, and all the client-side processing (applying remote changes, resolving conflicts, finding local changes, and marking them as synced) is done by Watermelon.

  See documentation for more details.

- **Support caching for non-global IDs at Native level**

0.9.0 - 2018-11-23

New

- Added `Q.like` - you can now make queries similar to SQL `LIKE`

0.8.0 - 2018-11-16

New

- Added `DatabaseProvider` and `withDatabase` Higher-Order Component to reduce prop drilling
- Added experimental Actions API. This will be documented in a future release.

Fixes

- Fixes crash on older Android React Native targets without `jsc-android` installed

0.7.0 - 2018-10-31

Deprecations

- [Schema] Column type 'bool' is deprecated — change to 'boolean'

New

- Added support for Schema Migrations. See documentation for more details.
- Added fundaments for integration of Danger with Jest

Changes

- Fixed "dependency cycle" warning
- [SQLite] Fixed rare cases where database could be left in an unusable state (added missing transaction)
- [Flow] Fixes `oneOf()` typing and some other variance errors
- [React Native] App should launch a little faster, because schema is only compiled on demand now
- Fixed typos in README.md
- Updated Flow to 0.85

0.6.2 - 2018-10-04

Deprecations

- The `@nozbe/watermelondb/babel/cjs` / `@nozbe/watermelondb/babel/esm` Babel plugin that ships with Watermelon is deprecated and no longer necessary. Delete it from your Babel config as it will be removed in a future update

Refactoring

- Removed dependency on `async` (Web Worker should be ~30KB smaller)
- Refactored `Collection` and `simpleObserver` for getting changes in an array and also adds CollectionChangeTypes for differentiation between different changes
- Updated dependencies
- Simplified build system by using relative imports
- Simplified build package by outputting CJS-only files

0.6.1 - 2018-09-20

Added

- Added iOS and Android integration tests and lint checks to TravisCI

Changed

- Changed Flow setup for apps using Watermelon - see docs/Advanced/Flow.md
- Improved documentation, and demo code
- Updated dependencies

Fixed

- Add quotes to all names in sql queries to allow keywords as table or column names
- Fixed running model tests in apps with Watermelon in the loop
- Fixed Flow when using Watermelon in apps

0.6.0 - 2018-09-05

Initial release of WatermelonDB

Edit this page

Previous
« Contributing

Docs

Installation

Contributing

Community

Stack Overflow

Twitter

More

GitHub