Linköpings Universitet | Institutionen för Datavetenskap
Examenarbete, 16 hp | Innovativ Programmering
Vårterminen 2018 | LIU-IDA/LITH-EX-G--18/049--SE

Using IndexedDB with a spatial database

Ludvig Eriksson

Handledare: Erik Berglund
Examinator: Anders Fröberg

Linköping University
SE-581 83 Linköping
013-28 10 00, www.liu.se

Using IndexedDB with a spatial database

Ludvig Eriksson
Linköping University
Norrköping, Sweden
lueri638@student.liu.se

ABSTRACT
Web technologies are becoming increasingly useful with new features and the gap to native apps is narrowing. Recently, IndexedDB was added to the web standard to provide large scale storage solutions directly in the browser. Is it performant enough to be used with a spatial database? In this study, such a system is developed for Foran Sverige AB and we learn that IndexedDB indeed can be used for this purpose. Even storage demanding geospatial applications can be developed as a multi-platform system with a single codebase, all while broadening the possible audience reach by avoiding an app installation process.

Author Keywords
Progressive Web Apps; IndexedDB; GeoJSON; spatial database; GIS

INTRODUCTION
The mobile web has always been a useful tool to reach many users, since you aren't limited to a certain platform. Statistics show that the web dominates user reach; during 2017 the top mobile web properties had 2.2 times the number of unique visitors as the top mobile apps[1]. However, the same statistics show that the apps get a whopping 16 times more usage time. One reason for this is that the mobile web always has been relatively limited, whereas apps have had a deeper integration with the operating system, for example via push notifications and complex storage and caching solutions. But what if the high-friction action of installing an app could be bypassed, and a native-like experience instead could be delivered through the web? This is the goal with Progressive Web Apps (PWAs) [1].

One technology commonly used to create PWAs is Service Workers. They are processes running in the background in the browser itself, but they can act as a proxy between the web app and the network and can respond with cached content for much faster load times. For complex storage solutions Service Workers can use IndexedDB, an interface for a JSON-based database that can hold large amounts of data[2].

In addition to having the advantage for the user of requiring a lower effort to visit, PWAs also have other advantages over mobile apps for the developers. With them you don't have to maintain separate code bases for different device types, screen sizes or operating systems. This can be both economically more desirable and require fewer resources for companies that are building systems for mobile platforms.

Motivation
Foran Sverige AB develops systems for gathering, analyzing and presentation of forestry and nature data[3]. When clients are out in the field, often the mobile reception isn't optimal and load times can be unreasonably slow, if present at all. Much of the data delivered to the clients is geospatial vector data which shows various representations of the terrain around them.

Geospatial data can be represented in many ways, but one widely used format is GeoJSON which, like IndexedDB, also is based on JSON. It was released as an open standard in august 2016[4].

Purpose
The goal with this paper is to examine whether IndexedDB is a feasible solution to use with a spatial database by building a layer on top of it to handle the storage and retrieval of GeoJSON files. General spatial queries are also required for the solution to be considered adequate.

Specific Research Question (SRQ)
Can IndexedDB be performant enough to store a spatial database on a mobile device?

What counts as performant enough will differ since each system have different requirements, but in this paper it will be based on the requirements from Foran Sverige AB, which are described in the background chapter.

BACKGROUND
The requirements for the system developed in this study are specified by Foran Sverige AB. Some aspects of it will be tailored to fit their datasets and existing systems. One such aspect is that the datasets that Foran will use this system mostly are divided into groups that only contain features of the same type. Therefore, a possible optimization for this scenario is examined.

[1] "The 2017 U.S. Mobile App Report," comScore, 2017.

[2] Mozilla Foundation, "IndexedDB API" https://developer.mozilla.org/docs/Web/API/IndexedDB_API

[3] Foran Sverige AB, "Om företaget" http://foran.se/sv/om-foeretaget. [Accessed 24 January 2018].

[4] RFC 7946 – "The GeoJSON Format"

Also specified are which spatial operations the system needs to have in order to give the functionality needed to integrate with Foran's other systems. These are listed and explained in the implementation chapter.

The end user of this system will mostly begin with a quite large dataset and then make many small modifications to the data one by one. Initial insertion may take quite a long time if the dataset is large, but since this mostly will happen only once it is deemed acceptable. The timeframe for what is considered acceptable is within one minute for datasets of a few thousands or tens of thousands of features. When launching the system, most often a bulk loading of features within a region will occur. It would be preferable if this happened within a few seconds. After the initial insertion and loading, the one by one modifications need to happen quickly enough so that the system doesn't feel choppy and the user doesn't have to look at a frozen user interface. Studies have shown that human delay perception is around 100 milliseconds [2] [3], so that is what I will use as the acceptable timeframe.

THEORY

Progressive Web Apps

A Progressive Web App, or PWA for short, is technically not an app in the common sense of the word. It is a regular website which leverages certain technologies that often are associated with native apps, such as push notifications or databases. IndexedDB is the database solution for websites, so if a website uses IndexedDB it can be seen as a Progressive Web App.

Spatial databases and IndexedDB

Databases are usually structured in one of two ways; relational or non-relational. IndexedDB is a non-relational database where each entry is accessed via an index. Since spatial data is two-dimensional (latitude and longitude) it is not obvious how to store it with a single index. To solve this problem, a spatial indexing method can be used. They provide ways to structure spatial entries in a non-relational database.

The difference between a regular database and a spatial database is that a spatial database is optimized for performing spatial queries. These might be operations such as intersects, buffer and union, which are calculated on a single or two features.

Spatial databases aren't new, but the combination with PWAs and IndexedDB, which are fairly new technologies (the latest specification, Indexed Database API 2.0, was released by W3C on January 30, 2018[5]), has not been researched much. Also, the datasets handled by companies like Foran can be extremely large (it might for example be detailed vector data covering entire countries) which makes the situation unique.

All interactions with IndexedDB are asynchronous and are made by creating request objects that have callback methods when the operations are completed. Databases are stored per origin[6] and are identified by a unique name. Each database can have multiple *object stores*, which can be seen as different named buckets of data.

A valid concern when it comes to new web technologies is that it might only work in some browsers, and therefore not for all users. IndexedDB, however, is well supported in all major browsers[7], with the exception being that Microsoft Edge only has partial support; it does not support IndexedDB inside blob web workers.

R-trees

One commonly used spatial indexing method is called R-trees and was suggested by Antonin Guttman in 1984 [4]. It works by keeping a data structure which is similar to a binary search tree, which means it can be traversed quickly.

Each feature in the database is indexed by its bounding box, which means that during a search you only need to traverse nodes in the tree whose bounding box intersects with the area you are searching.

OMT algorithm

The Overlap Minimizing Top-down bulk loading algorithm for R-tree (OMT algorithm for short) was presented in 2003 by T. Lee and S. Lee [5] in a paper with the same name. It is an optimization for insertions into an R-tree when you have all data from the beginning, for example by grouping features that are close to each other together to minimize overlaps.

Quadtrees

The quadtree data structure was presented by R. A. Finkel and J. L. Bentley in 1974 [6], and with its optimizations for multi-dimensional indexing it is perfect for indexing points in a mapping system but less so for features that cover areas.

It works by having a top-level node which covers the entire area to be mapped. You then insert points in that node until it has reached the capacity of the tree (which you specify yourself). After that, you divide the tree in four quadrants and insert the next point into the sub-quadrant that contains the point.

Since you can only store a certain number of features in a given node and you want the nodes to contain its features, this method does not work well for features other than points. A feature with an area might not be able to be placed in the tree if its target node already is full, since

[5] W3C, "Indexed Database API 2.0" https://www.w3.org/TR/2018/REC-IndexedDB-2-20180130/

[6] The origin is defined by the scheme, host and port of the URL for the website.

[7] https://caniuse.com/#feat=indexeddb

traversing down the tree may cause the nodes to become smaller than the feature itself.

GeoJSON

GeoJSON is a standardized[8] format that can be utilized to represent geospatial features with JSON files. It supports seven geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon and GeometryCollection. Coordinates are defined by an array of the longitude and latitude, in that order.

All GeoJSON objects must have a property called “type”, which can either be “FeatureCollection” in which case the object only will have one other property called “Features” – an array of GeoJSON features, or it can be “Feature”, which means the object is a single feature. Features must have a property called “geometry”, containing a geometry object, and one called “properties” which contains attributes about the feature – you can store anything you want to associate with the feature here. Geometries in turn also have a property called “type”, which must be one of the seven mentioned above, as well as one called “coordinates”, which depending on the geometry type contains different levels of nested coordinate arrays.

```
{
  "type": "Feature",
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [[0, 0], [8, 0], [8, 8], [0, 8], [0, 0]],
      [[2, 2], [2, 3], [3, 3], [3, 2], [2, 2]]
    ]
  },
  "properties": {}
}
```

Snippet 1. An example of a GeoJSON feature of type Polygon that contains a hole. The feature follows the right-hand rule.

In addition to the format of the JSON file, the standard specifies a few rules for when features are valid. An example is the right-hand rule, which states that for Polygon and MultiPolygon features, exterior rings must be specified in the counter-clockwise direction and interior rings must be specified in the clockwise direction. Snippet 1 shows an example of a valid Polygon feature.

One thing to note is that the latest GeoJSON specification removed alternative coordinate systems, which were a part of the previous specification[9]. All coordinates are now expected to be using the World Geodetic System 1984 (WGS 84). However, the current specification also states that alternative coordinate reference systems may be used if all involved parties have a prior arrangement. The system developed in this paper makes no assumptions about the coordinate reference system used, so it can be used with any. Foran, as well as most applications in Sweden, uses the Swedish Reference Frame 1999 (SWEREF 99).

JSTS

A popular choice for performing various spatial operations in Java is the Java Topology Suite, or JTS. It is an open source library with support for many fundamental geospatial operations. This is the project upon which many others are based[10], one of which is the JavaScript Topology Suite, or JSTS, which is a port of the library to JavaScript. It is a powerful library for many things related to spatial data.

ECMAScript

ECMAScript is a scripting language specification which JavaScript conforms to. New standards are released by the Ecma International organization, the latest finalized version (as of this paper) being ECMAScript 2017[11].

Promises and async/await

With ECMAScript 2015, Promises were introduced[12]. They are objects that can be used for asynchronous tasks and work similar to having callback methods. They may also be used together with the async/await syntax introduced in ECMAScript 2017[13]. Snippet 2 shows an example of a function returning a Promise and Snippets 3 and 4 shows how to call this method using either Promise syntax or async/await. This is an example where the function may fail and return an error, but they can of course be used for tasks that just take some time to execute; then the error handling parts can be removed, and the expression becomes significantly more compact.

While the standards for these features were only recently released, all major browsers already have full support for both Promises[14] and async/await[15].

Modules

JavaScript has grown into something a lot larger than it was in its beginning, and as code bases become ever larger it becomes harder to structure JavaScript code. The lack of

[8] RFC 7946 – “The GeoJSON Format"

[9] https://tools.ietf.org/html/rfc7946#section-4

[10] JTS GitHub repo, https://github.com/locationtech/jts#downstream-projects

[11] http://www.ecma-international.org/ecma-262/8.0/index.html

[12] https://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects

[13] https://www.ecma-international.org/ecma-262/8.0/#sec-async-function-definitions

[14] https://caniuse.com/#feat=promises

[15] https://caniuse.com/#feat=async-functions

modularity in JavaScript has been a well-known issue; several studies have been made of how to use a module-like pattern with JavaScript [7] [8] [9]. But, also released with ECMAScript 2015 was the import syntax for modular development[16]. With this, JavaScript finally received native support for modularity. You can now in your HTML file import a single script file, see Snippet 5, and from that script file import other modules, see Snippet 6.

```
function asynchronousTask() {
    return new Promise(function(resolve, reject) {
        // Perform asynchronous task...
        if (/* succeeded */) {
            resolve(/* result */);
        } else {
            reject(/* error */);
        }
    });
}
```

Snippet 2. An asynchronous function that can either succeed or fail.

```
asynchronousTask().then(function(result) {
    // Task complete
}).catch(function(error) {
    // An error occurred
});
```

Snippet 3. Calling the function from Snippet 2 using Promise syntax.

```
try {
    await asynchronousTask();
    // Task complete
}
catch(error) {
    // An error occurred
}
```

Snippet 4. Calling the function from Snippet 2 using async/await syntax.

```
<script type="module" src="main.js"></script>
```

Snippet 5. Importing a script module in an HTML file.

```
import { module } from 'module.js';
```

Snippet 6. Importing another module in JavaScript.

[16] https://www.ecma-international.org/ecma-262/6.0/#sec-imports

Related work

One initial study showed that Service Workers, the running processes that can provide cached resources by interacting with IndexedDB, don't have a significant impact on energy consumption, especially for newer devices [10]. Since energy consumption is a concern for mobile users, especially if their work is depending on it, this means that the choice of PWAs for this study might very well be suitable for this aspect.

Research has also been made to show that IndexedDB can handle large amounts of data with performance similar to native alternatives [11]. This is also a promising result for this study since I will need to handle large amounts of data, but here I will focus on how well suited IndexedDB is for spatial data in particular.

The fact that so little research has been made about these technologies hopefully means that this study can contribute with some new insights within the field.

METHOD

The work consisted of multiple parts, the first being to choose a spatial indexing method that works well with IndexedDB. Then the chosen indexing method was used to implement the spatial database structure. Finally, the implementation was evaluated in a few different real-world scenarios to determine whether the solution was feasible for the specification.

All parts of the system are implemented as ECMAScript 2015 modules and are therefore simple to use. Even though the implementation is split up into multiple files, the end user only needs to import one. Snippet 7 shows how the end user might import the `SpatialDatabase` class, described later in this chapter.

```
import { SpatialDatabase } from 'database.js';
```

Snippet 7. How the end user might import the `SpatialDatabase` class for use in another system.

Choosing the indexing method

By looking at how common JavaScript libraries handles spatial indexing I found that R-tree was a very common method. It is used internally in for example the mapping library OpenLayers[17] and the spatial analysis library Turf.js[18]. I also found an existing library for building an R-tree in memory, called Rbush[19].

Another popular method, according to Foran, is the Quadtree [6]. This might not seem optimal since it only works for point features, but as mentioned the datasets that

[17] OpenLayers API Documentation (http://openlayers.org/en/latest/apidoc/ol.source.Vector)

[18] Turf.js Source Code (https://github.com/Turfjs/turf)

[19] https://github.com/mourner/rbush

Foran will use with these systems will mostly be divided into groups of the same type. So, it might be interesting to see if Quadtrees could perform better for this type of data, and if so, have a separate implementation that handles point data. Therefore, I decided to compare the performance of R-trees and Quadtrees for storing points in IndexedDB.

I based my R-tree implementation on the Rbush library by rewriting it to store all data in IndexedDB instead of an in-memory object structure. The quadtree implementation I wrote from scratch, based on the pseudo code on the Wikipedia page on Quadtrees[20]. To compare them I used a dataset of over 11,000 points spread around the globe. The two benchmark I used was to insert the points one by one, as well as the time to retrieve all points within a bounding box roughly the size of Europe, which contained about 3,000 of the points.

The results from this benchmark, which can be found in the results chapter in Table 1, concluded that it was only marginally faster to use Quadtrees and not worth splitting the implementation. So, it was decided that the implementation would use R-trees.

IndexedDB layer

I chose to divide my implementation in two layers, the first being the one to interact with IndexedDB directly. This first layer, handled by a class called `LocalDatabase`, can be used regardless of which indexing method the second layer uses. The first layer wraps up the request objects and callback methods of IndexedDB in Promises, which simplifies the interaction with the database as well as making it possible to use its API with async/await. Figure 1 shows a graphical representation of how the layers are structured.

To create a `LocalDatabase`, you need to specify a name for the database you want to access, as well as the names of the object stores you want it to contain (these will be created on the first launch). Then you can start inserting and retrieving objects from the database as seen in Snippet 8.

Spatial database layer

Above the lower IndexedDB layer comes the layer which implements the spatial part of the database, and it's handled by the class `SpatialDatabase`. As with `LocalDatabase`, the implementation is based on Promises. This gives the user two different syntax options for using the database. Snippets 9 and 10 show how to initialize a database and insert a feature into it. A separate initialize function is needed since async functions always returns a promise, and constructors always return the created object, and the initialization needs to be asynchronous since IndexedDB must be setup with its internal request objects and callback structure.

[20] https://en.wikipedia.org/wiki/Quadtree#Pseudo_code

```
let db = await new LocalDatabase({
    name: 'db',
    objectStores: ['one', 'two']
}).initialize();
let obj = /* Any JSON-based object */;
await db.addObject(obj, 'one');
// If obj does not have a property 'id',
// IndexedDB will generate one.
let alsoObj = await db.getObject(obj.id, 'one');
```

Snippet 8. An IndexedDB database called "db" will be created, and it will contain two object stores called "one" and "two". Then an object is inserted into the "one" object store. Finally, that same object is read back from the database.

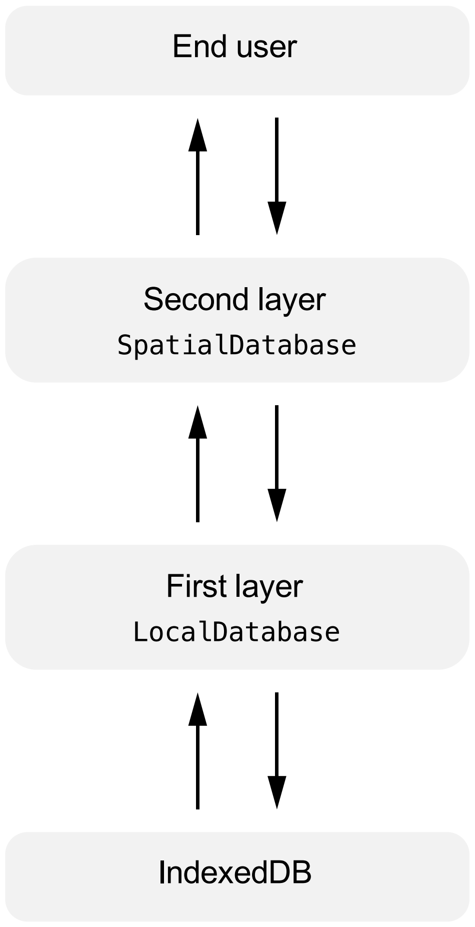

Figure 1. The layered structure of the implementation. The end user only interacts with the `SpatialDatabase` layer, which in turn never interacts with IndexedDB directly. That communication is handled by the `LocalDatabase` layer.

```
let feature = /* a GeoJSON object */;
let db = new SpatialDatabase();
db.initialize().then(function() {
    return db.insert(feature);
}).then(function() {
    // Done
});
```

Snippet 9. Insertion of a GeoJSON feature into the database using Promise syntax.

```
let feature = /* a GeoJSON object */;
let db = await new SpatialDatabase().initialize();
await db.insert(feature);
// Done
```

Snippet 10. Insertion of a GeoJSON feature into the database using async/await syntax.

`SpatialDatabase` uses `LocalDatabase` internally, creating two object stores; one for the features and one for the index (in this case, the R-tree).

In addition to inserting a single feature, a bulk insert method based on the OMT algorithm [5] was added by slightly modifying the implementation from the Rbush library to work with IndexedDB. Not only did this speed up the insertion process, it also sped up searches in the tree. Searching for features within a bounding box was already implemented in the comparison with Quadtrees, but in addition to this a few other ways to retrieve data was added. First, another bounding box search was added where features only need to intersect the bounding box, and not lie completely within it. Another search for features within an arbitrary polygon was also added. Finally, a nearest neighbor search was added, where the user can search for the X nearest neighbor features of a point, X being an integer.

Integrity and reliability

An important next step in the implementation was to ensure integrity and reliability of the database. Since the datasets used by the system can be very large, and the devices it can be used on can be fairly limited in terms of resources and computational power, it is not unreasonable to think that a crash may occur. In that case it is important that the data stays intact and isn't being corrupted. There might also be cases where multiple operations are being called on at the same time, and with the asynchronous nature of IndexedDB they may not completed in the same order they were initiated. This may result in errors in the data.

To ensure that the data isn't being corrupted all database operations related to an event needs to happen in a single transaction. To achieve this, I specified a simple data structure in which all changes that need to be made can be gathered, and then sent to a method which executes all changes with a single transaction to the database. An object with this data structure, shown in Snippet 11, can be sent to the lower IndexedDB layer and they will all be performed in a single transaction.

```
{
    "storeName": {
        "add": [obj, ...],
        "update": [obj, ...],
        "delete": [obj, ...]
    },
    ...
}
```

Snippet 11. Data structure for performing multiple updates in a single transaction. There are three possible changes for an object (in the snippet called `obj`): it can be added, updated or deleted. And these operations can be made to any object store (in the snippet defined by `storeName`).

To tackle the problem with multiple asynchronous methods not necessarily finishing in the correct order I added an internal queue structure to the `SpatialDatabase` class. All operations performed by the database will be added to the back of the queue, and if another operation already is being executed it will have to wait until that is finished before being run itself. This ensures that all operations on the database will return in the same order that they were called.

Validation

Since an IndexedDB database can store any JSON-based object, it is technically possible for the user to put anything in there. This would cause obvious problems when trying to perform operations which expect the data to be GeoJSON features. A few measures were taken to ensure both that the objects inserted in fact are GeoJSON features, as well as making sure the features have valid geometries and follow the GeoJSON standard.

- Objects must have a specified type of "Feature".
- Objects must have a geometry, and its type must be one of the seven supported by the GeoJSON standard.
- Polygon and MultiPolygon features must follow the right-hand rule.
- Polygon and MultiPolygon features must not have any self-intersections.
- Polygon and MultiPolygon features must not have holes that are outside of their exterior rings.
- Polygon and MultiPolygon rings must be closed.
- Coordinates must consist of valid numbers.

Spatial operations

In addition to inserting and searching for features in the database, a number of spatial operations for modifying features were added. These methods help the user of this database to perform common operations needed, as specified by Foran.

The implementations of these methods utilize the JSTS library, which has these and many more functions available. JSTS is only used to manipulate the geometries themselves, and this system wraps the library calls in own functions that only exposes an API for working with GeoJSON features. All operations are kept in separate files and are imported to the `SpatialDatabase` class as modules. An example of this can be seen in Snippet 12. By doing this it is easy to extend the system with new operations in the future, and since the `SpatialDatabase` class in no way knows about JSTS it is easy to change or have different underlying libraries performing the actual geometry manipulations.

```
import { buffer } from 'operations/buffer.js';
```

Snippet 12. How the buffer operation is imported into the `SpatialDatabase` class.

All calls to the database that includes a feature as a parameter can either be made with the GeoJSON object itself or via its unique identifier. Snippet 13 shows an example of how one of the spatial operations described later in this subsection can be used in these two ways.

```
// Using an identifier
let buffered = await db.buffer(123, 100);
console.log(buffered.id); // 123

// Using a feature object
let feature = /* Existing feature in database */;
await db.buffer(feature, 100);
```

Snippet 13. Buffering a feature with 100 meters. The call is made twice, first with only an identifier of a feature, second with a feature object itself.

Buffer
Takes an existing feature in the database and adds a buffer of a user specified size around it. An example can be seen in Figures 2 and 3, where the first is the feature before the buffer and the second is the feature after the buffer. Also specifiable is the number of line segments that should be used to represent a quadrant of a circle as well as the end cap style, which for example is used when buffering around the end of a line. Three styles are available.

- Round – a semi-circle.
- Flat – a straight line perpendicular to the end segment.
- Square – a half square.

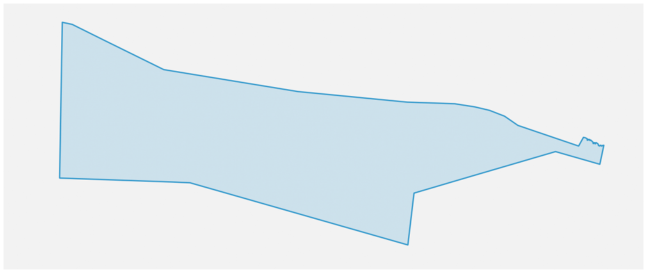

Figure 2. An arbitrary Polygon feature. This is the same feature that is available in Appendix 1.

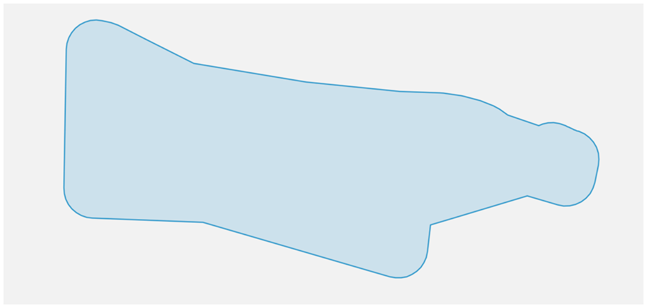

Figure 3. A buffered Polygon feature. For the original feature, see Figure 2.

Line clip
Takes an existing feature and clips it with a line, creating two or more new features as a result. Figure 4 shows a Polygon feature with a crossing LineString feature. Figure 5 shows the result of the line clip operation on that Polygon feature with that line. The user has the option to copy over attributes from the original feature to the resulting ones.

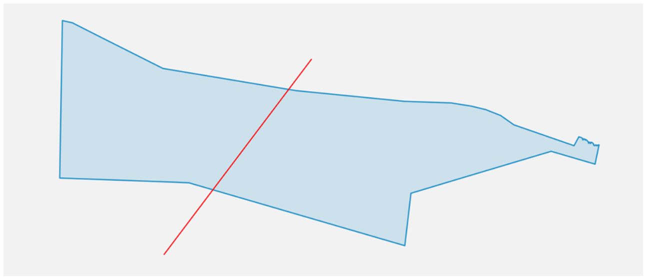

Figure 4. An arbitrary Polygon feature, shown in blue, and a LineString feature, shown in red. The Polygon feature is available in Appendix 1, and the LineString feature is available in Appendix 3.

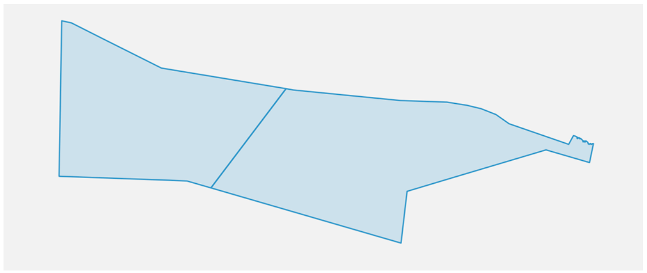

Figure 5. The result of a line clip operation performed on a Polygon feature with a LineString feature, both of which can be seen in Figure 4.

Polygon clip

Takes an existing feature and cuts out a polygon shape from it. Figure 6 shows an example of a Polygon feature, in blue, with another overlapping Polygon feature, in red. Figure 7 shows the resulting feature after the red feature has been clipped from the blue feature.

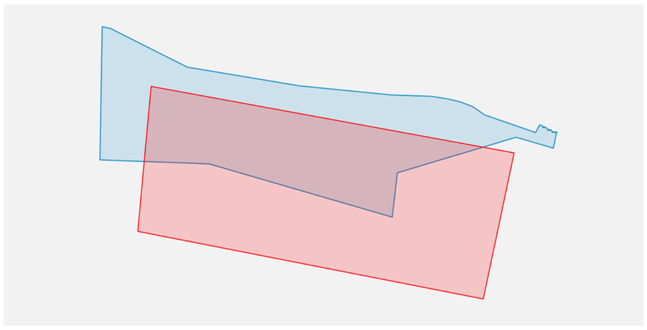

Figure 6. Two Polygon features. The blue feature is available in Appendix 1 and the red feature in Appendix 2.

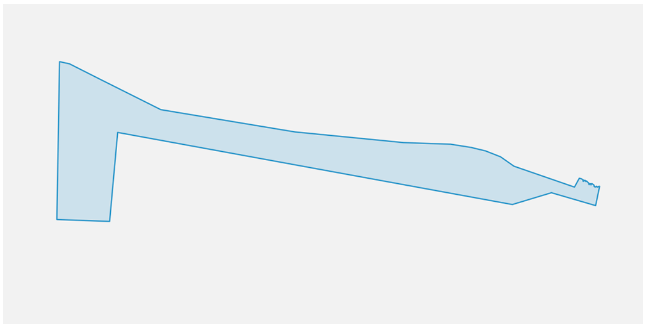

Figure 7. The result of a polygon clip operation, performed on one Polygon feature with another, both of which can be seen in Figure 6.

Union

Takes two existing features and combines them into a single feature. Figure 8 shows two adjacent Polygon features, and Figure 9 shows the resulting feature after performing the union operation on these two Polygon features. The user will also have to specify the attributes for the new feature, if they should have any.

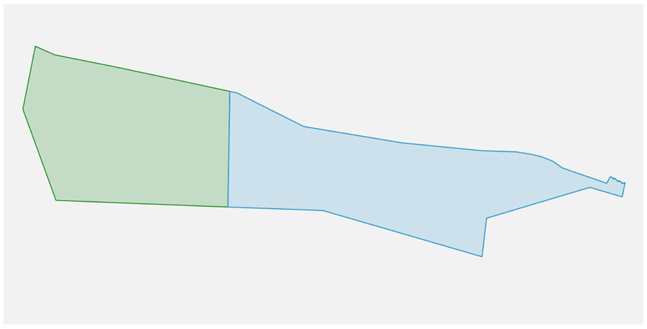

Figure 8. Two adjacent Polygon features. The blue feature is available in Appendix 1, and the green feature is available in Appendix 4.

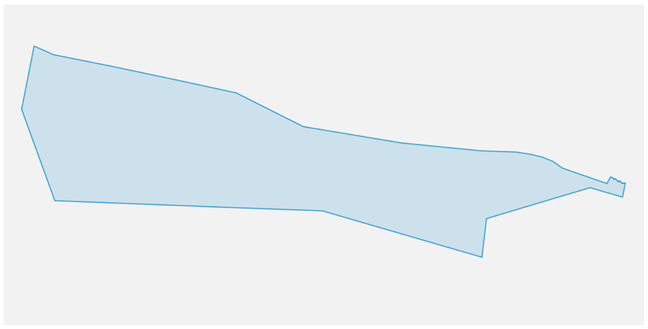

Figure 9. The result of a union operation performed on two Polygon features, both of which can be seen in Figure 8.

Real-world performance evaluation

Since the system will probably begin with a quite large dataset, and not build it up feature by feature, it is important that bulk insertion of features is performant. In order to test this, a sample GeoJSON file with 100,000 random points around the globe was inserted into the database. Another important performance aspect that deals with a lot of data is to retrieve all features within a bounding box, in order to display them for a map for example. To test this, a bounding box that just about covers Europe was used to retrieve all points within it. For this random dataset, about 3,800 points were within this area.

After the initial data has been inserted into the database, most operations are made one at a time. As previously mentioned, the important part here is that the system doesn’t feel choppy or the user needs to look at a frozen user interface, and the metric used to compare this will be the 100 milliseconds found by previous studies. To benchmark the spatial operations, an arbitrary Polygon feature, available in Appendix 1, was designed to be representative of an average feature from a real dataset. Each operation was performed 100 times to eliminate environmental variables, and then the average time was used as the benchmark result.

- The buffer operation was performed by buffering the Polygon feature with 100 meters. See Figures 2 and 3 for a graphical representation of the feature before and after the operation.
- The line clip operation was performed with a simple LineString feature consisting of only two points; one on each side of the Polygon feature being clipped. It represents the user splitting a polygon in two by drawing a simple line over it, for example. The LineString feature used in this operation is available in Appendix 3. See Figures 4 and 5 for a graphical representation of the features before and after the operation.
- The polygon clip operation was performed with a simpler four-sided Polygon feature, representing the user clipping away part of the feature by drawing a simple polygon on the screen, for example. The polygon used in this operation is

available in Appendix 2. See Figures 6 and 7 for a graphical representation of the features before and after the operation.

- The union operation was performed against another Polygon feature, designed the same way that the first Polygon feature was designed. It represents the user wanting to merge two existing features, for example. The Polygon feature is available in Appendix 4. See Figures 8 and 9 for a graphical representation of the features before and after the operation.

RESULTS

All benchmarks whose results are revealed in this chapter were performed on two different devices, one desktop device and one mobile device, unless stated otherwise. Their specifications are described below.

- Desktop device: A computer running Windows 10 with an Intel Core i5-7400 processor and 16GB of memory, using version 59 of the Firefox web browser.
- Mobile device: An Apple iPhone X, which has an Apple A11 Bionic processor and 3GB of memory, running iOS 11.3 and using the Safari web browser.

Choosing the indexing method

Indexing method	Insertion	Retrieval
R-tree	68s	1.0s
Quadtree	65s	1.0s

Table 1. Results of the indexing method comparison benchmarks. The times are averages of 10 runs of each operation. This benchmark was only run on the desktop system.

Real-world performance evaluation

Operation	Desktop	Mobile
Insert 100,000 points	15s	17s
Retrieve points within Europe	10s	22s

Table 2. Results of the insertion and retrieval benchmarks. The times are averages of 10 runs.

Operation	Desktop	Mobile
Buffer	50ms	72ms
Polygon clip	38ms	36ms
Line clip	59ms	52ms
Union	50ms	55ms

Table 3. Results of the different spatial operations. The times are averages of 100 runs.

DISCUSSION

Results

One interesting thing is that the insertion benchmarks from Table 1 were faster than the retrieval benchmarks on the mobile device, while the opposite on the desktop. It shows that different architectures and browser implementations can result in very different performance metrics.

Method

When choosing the indexing method I would ideally have liked to compare several methods. However, due to time constraints, I needed to continue with the rest of the implementation. So, there might be more performance found in a system like this if an even more optimal indexing method exists.

The indexing method benchmarks were made very early on in development, and it was only later that I realized that it is more relevant to run the benchmarks on a mobile device. But, due to time constraints, and since a lot of work already had been put into the chosen implementation, I chose to not rerun the benchmarks on a mobile device.

The real-world performance evaluation was not as real as initially planned. Since the system in which the system developed in this paper is to be used at Foran hasn't been developed yet, I could not use that for testing. The feature operations are very much representative of a real-world scenario. However, I did not have access to a dataset large enough for the initial bulk insertion and retrieval benchmark. What I instead did was to create a sample GeoJSON file with 100,000 random points around the globe. This distribution isn't really typical, but more of a worst-case scenario for the R-tree structure since it tries to group features as closely together as possible. This is not all bad though, since knowing the worst-case performance of the system is very useful.

Reliability

While the results of this study are quite concrete and should be the same for any similar implementation, whether the system is feasible for a specific application will be dependent on the requirements of that specifications. The system meets Foran's requirements but may not meet all. That is also why I after my SRQ specifies what I mean by *performant enough.*

The work in a broader context

By being able to cache large datasets even for web applications, many large data transfers can be avoided. Should it be the norm for demanding applications to use these storage technologies it could have a positive impact by reducing the load on the cellular networks. This both save money in the industry, as well as reduce energy consumption; not only would data centers experience less strain, but mobile devices themselves would consume less energy since reading from an onboard storage doesn't require nearly as much energy as firing up power hungry cellular antennas.

CONCLUSIONS

The purpose was to develop a layer above IndexedDB that represents a spatial database and examine whether it is a feasible solution, meaning that it performs within the specified targets. The layer has been completed with the desired functionality, so all that remains is to evaluate its performance.

By comparing the benchmarking results with the initial specifications by Foran, the conclusion is that the system indeed meets the performance requirements. The numbers referred to in this chapter will be from the benchmarks on the mobile device, since this is where the system is mostly intended to be used.

For bulk insertion of a dataset of adequate size, the resulting time of 17 seconds, see Table 2, is well within the specified minute. The searching for features is a bit on the slow side with its 22 seconds. However, as discussed in the previous chapter, this was kind of a worst-case scenario, and performance will typically be better than this. As for the feature manipulation, the slowest operation was the buffer operation, and that result of 72 milliseconds, see Table 3, is within the target of 100 milliseconds; hence all of them are meeting the requirements.

Answer to SRQ

Can IndexedDB be performant enough to store a spatial database on a mobile device?

The results presented in this paper show that, with specifications similar to those of Foran Sverige AB, IndexedDB indeed is performant enough that it can be used to store a spatial database. This enables more options for cross-platform implementations – native code might not be needed for storage demanding spatial applications to run with adequate speed. For Foran it means that for this system they can use a single code base to target practically all platforms, and the users can use whichever device they like.

Future work

One thing I would have liked to do if I had more time is to run the benchmarks on many more devices. I only had access to my own mobile device, but it would be interesting to see a comparison with a number of different devices. I don’t really know if the difference in performance between my desktop device and mobile device is due to hardware or browser implementations; this is something that could have been seen with a more significant number of testing devices. Doing a comprehensive study where all major browsers on multiple platforms are compared might be an option for a thesis for someone in the near future. It would, together with this paper, contribute even more to a field of increasing popularity.

Contributions

This study has provided a concrete example of how, thanks to Progressive Web Apps, web technologies can be used for things that were not previously possible. Before IndexedDB, these storage possibilities just didn’t exist in the browser. By leveraging web technologies, a multi-platform system with a single codebase can be developed even for storage demanding geospatial systems, and in the process, you are avoiding the higher-friction app installation action.

REFERENCES

[1] A. Russell, *Progressive Web Apps: Escaping Tabs Without Losing Our Soul,* Infrequently Noted, 2015.

[2] R. B. Miller, "Response time in man-computer conversational transactions," *AFIPS Joint Computer Conferences,* pp. 267-277, 1968.

[3] S. K. Card, G. G. Robertson and J. D. Mackinlay, "The information visualizer, an information workspace," *Proceedings Of The SIGCHI Conference: Human Factors In Computing Systems,* vol. April, pp. 181-186, 1991.

[4] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," Association for Computing Machinery, 1984, pp. 47-57.

[5] T. Lee and S. Lee, "OMT: Overlap Minimizing Top-down Bulk Loading Algorithm for R-tree," *CAiSE Short Paper Proceedings,* Vols. 74, June, pp. 69-72, 2003.

[6] R. A. Finkel and J. L. Bentley, "Quad trees, A data structure for retrieval on composite keys," *Acta Informatica,* vol. 4, no. 1, pp. 1-9, 1974.

[7] F. De Kerchove, J. Noyé and M. Südholt, "Extensible modules for JavaScript," *Proceedings Of The ACM Symposium On Applied Computing,* no. April, 2016.

[8] F. Johansson, "Designmönster i Javascript," Networked Digital Library of Theses & Dissertations, EBSCOhost, Uppsala, 2011.

[9] S. Ryu and S. Kang, "Formal Specification of a JavaScript Module System," *Acm Sigplan Notices,* vol. October, pp. 621-638, 2012.

[10] I. Malavolta, G. Procaccianti, P. Noorland and P. Vukmirovic, "Assessing the Impact of Service Workers on the Energy Efficiency of Progressive Web Apps," *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft),* pp. 35-45, 2017.

[11] E. Nilsson and A. Lagerqvist, "The performance of mobile hybrid applications: An experimental study," Networked Digital Library of Theses & Dissertations, EBSCOhost, Jönköping, 2015.

APPENDIX 1

```
{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [[592399.7343, 6501021.1675], [592382.142, 6500935.058899999596], [592185.411, 6500992.463899999857],
[591768.047, 6500868.553899999708], [591557.134, 6500805.6469], [591529.493, 6500571.412899999879],
[591422.496, 6500602.452899999917], [590561.357, 6500852.275899999775], [590481.382, 6500855.1549],
[589981.680199999944, 6500874.133], [589994.136599999969, 6501576.813], [590038.6923, 6501567.2752],
[590179.8898, 6501495.11429999955], [590444.9059, 6501362.893199999817], [591041.8802, 6501264.2049],
[591527.4624, 6501215.878399999812], [591736.815199999954, 6501208.546799999662], [591828.0712,
6501194.872], [591892.279499999946, 6501178.410099999979], [591959.067599999951, 6501152.1315],
[592018.648599999957, 6501110.99469999969], [592288.1149, 6501018.217699999921], [592309.9546,
6501057.5263], [592319.3766, 6501055.106399999931], [592326.3728, 6501050.533699999563], [592326.8995,
6501050.052099999972], [592326.3854, 6501045.743099999614], [592326.2548, 6501044.7198], [592329.0747,
6501044.080799999647], [592329.9429, 6501043.877299999818], [592330.6305, 6501046.7268], [592330.7752,
6501047.9983], [592331.1785, 6501048.2131], [592336.1372, 6501047.2169], [592337.345199999982,
6501046.692599999718], [592337.147699999972, 6501046.0734], [592336.9588, 6501045.2363], [592338.636,
6501043.193], [592339.3785, 6501042.5461], [592340.810599999968, 6501044.3398], [592341.1546,
6501045.081799999811], [592343.1065, 6501044.4654], [592348.1319, 6501039.918], [592352.609199999948,
6501034.228199999779], [592352.148799999966, 6501030.942499999888], [592352.0526, 6501030.0436],
[592353.3506, 6501028.7675], [592353.970199999982, 6501028.374099999666], [592356.4006, 6501031.8704],
[592356.958899999969, 6501032.831899999641], [592358.353099999949, 6501032.286799999885],
[592359.344399999944, 6501031.8178], [592359.659, 6501030.514899999835], [592360.9754, 6501028.12239999976],
[592361.7183, 6501027.4495], [592362.1833, 6501027.695], [592363.6352, 6501031.661399999633], [592363.6918,
6501032.7778], [592365.460399999982, 6501033.19], [592368.002, 6501032.586699999869], [592373.3229,
6501028.568699999712], [592375.766799999983, 6501025.7878], [592376.1596, 6501021.6306], [592376.2064,
6501019.738599999808], [592381.5155, 6501017.950899999589], [592384.102399999974, 6501019.982499999925],
[592385.0378, 6501021.592299999669], [592387.1776, 6501021.564], [592388.6171, 6501019.66579999961],
[592389.8479, 6501018.486899999902], [592392.265699999989, 6501018.039099999703], [592393.2465,
6501020.3623], [592398.676899999962, 6501021.3003], [592399.7343, 6501021.1675]]
    ]
  }
}
```

APPENDIX 2

```
{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [
        [590182.804021941614337, 6500496.124051729217172],
        [590252.904221319011413, 6501261.384561598300934],
        [592174.818020915030502, 6500910.883564711548388],
        [592011.250889034476131, 6500139.781371560879052],
        [590182.804021941614337, 6500496.124051729217172]
      ]
    ]
  }
}
```

APPENDIX 3

```json
{
  "type": "Feature",
  "properties": {},
  "geometry": {
      "type": "LineString",
      "coordinates": [
          [590450.061032070894726, 6500532.634572226554155],
          [591110.171242874930613, 6501403.04538116324693]
      ]
  }
}
```

APPENDIX 4

```
{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [[[593162.4804, 6506127.3426], [593161.7791, 6506123.373499999754], [593158.4668,
6506107.8376], [593155.741, 6506100.681599999778], [593152.111599999946, 6506090.734299999662],
[593147.8678, 6506086.210099999793], [593023.9899, 6506203.472299999557], [593012.9882, 6506213.8865],
[593009.561299999943, 6506389.541699999943], [593008.7611, 6506389.9605], [593010.045,
6506397.064399999566], [593009.8288, 6506397.8513], [593003.163399999961, 6506422.14], [592999.708699999959,
6506434.0329], [592990.411899999948, 6506464.10099999979], [592988.4957, 6506468.50569999963], [592984.6927,
6506477.2476], [592978.1557, 6506490.853], [592972.6535, 6506500.79349999968], [592975.5036,
6506499.270299999975], [593012.924099999946, 6506515.6353], [593026.8858, 6506557.761099999771],
[593004.821299999952, 6506587.87899999972], [592969.7199, 6506619.28699999955], [592910.9161, 6506696.7082],
[592887.3702, 6506729.701899999753], [592901.088299999945, 6506792.080199999735], [592900.6012,
6506832.5853], [592884.184, 6506874.3457], [592859.522, 6506879.8365], [592821.8405, 6506885.170599999838],
[592808.4209, 6506918.2861], [592776.264899999951, 6506945.389], [592755.7166, 6506969.7379],
[592729.416699999943, 6506991.1239], [592693.4455, 6507094.862499999814], [592700.104399999953,
6507142.68759999983], [592693.7439, 6507190.3562], [592703.2214, 6507240.896399999969],
[592815.795799999963, 6507207.3399], [592816.723899999983, 6507196.5695], [592818.733099999954,
6507189.2582], [592820.583, 6507180.907599999569], [592827.589799999958, 6507169.80719999969], [592833.5507,
6507162.53699999955], [592842.267, 6507154.768799999729], [592846.6052, 6507152.5403], [592851.686699999962,
6507150.8519], [592858.2966, 6507151.962399999611], [592865.4407, 6507156.1698], [592903.152,
6507101.724799999967], [592928.040099999984, 6507077.4281], [592947.124399999972, 6507054.5085],
[592960.5614, 6507019.946299999952], [592968.1946, 6506986.761199999601], [593012.410599999945,
6506919.292299999855], [593035.223099999945, 6506826.970099999569], [593032.817, 6506786.4302],
[593056.1715, 6506769.3491], [593091.7426, 6506698.8826], [593142.826299999957, 6506661.879599999636],
[593159.139199999976, 6506628.7988], [593139.164899999974, 6506605.4095], [593132.2972,
6506574.943699999712], [593148.314299999969, 6506566.455299999565], [593169.8395, 6506581.1824],
[593200.114, 6506590.2274], [593219.1635, 6506570.200899999589], [593236.6794, 6506557.3902], [593246.7883,
6506558.958499999717], [593252.465899999952, 6506562.843], [593253.213, 6506558.238099999726], [593254.0908,
6506547.595099999569], [593254.0307, 6506537.003299999982], [593254.0058, 6506533.8788], [593250.6175,
6506503.876899999566], [593249.6744, 6506495.564399999566], [593247.9057, 6506479.9753], [593241.694,
6506450.939199999906], [593239.4527, 6506445.07], [593237.8284, 6506434.6468], [593229.629599999986,
6506404.5871], [593222.7031, 6506388.2267], [593216.363, 6506369.8738], [593211.3151, 6506347.974899999797],
[593208.850199999986, 6506324.3262], [593207.253799999948, 6506311.5915], [593206.8061, 6506286.4675],
[593205.8174, 6506269.9598], [593206.5827, 6506247.8808], [593207.270499999984, 6506237.4543], [593206.2069,
6506221.976599999703], [593206.076099999947, 6506212.071299999952], [593202.210899999947, 6506195.7477],
[593199.096299999976, 6506184.5568], [593194.4231, 6506173.0661], [593178.168199999956,
6506153.031899999827], [593169.906599999988, 6506143.778599999845], [593167.1501, 6506139.1841],
[593163.7097, 6506134.3002], [593162.4804, 6506127.3426]]]
  }
}
```