INDEXSQL

CBD Project

05/18/2019

Daniel Diment Rodriguez

Contents

Introduction

What is IndexSQL?

IndexSQL is a wrapper that helps with the use of IndexedDB by making it SQL like.

What is IndexedDB?

IndexedDB is a low-level API for client-side storage of significant amounts of structured data, including files/blobs. This API uses indexes to enable high-performance searches of this data. While Web Storage is useful for storing smaller amounts of data, it is less useful for storing larger amounts of structured data. IndexedDB provides a solution.[1]

What is the problem with IndexedDB?

As IndexedDB is a low-level API, it is quite powerful, but it is not simple. Here is an example of how a single insert works:

```
let requestDB = indexedDB.open("IndexSQL",1);
requestDB.onerror = function(event) {
    log.warn("Database error: " + event.target.errorCode);
};
requestDB.onsuccess=function(event){
    let db=event.target.result;
    let transaction = db.transaction(["persons"], "readwrite");

    transaction.onerror = function(event) {
        console.warn("Database error: " + event.target.errorCode);
    };

    transaction.oncomplete = function(event) {
        //SAVED
    };

    var objectStore = transaction.objectStore("persons");
    var request = objectStore.put({Name:"pipo", ect…},DBUtils.name);
}
```

Here is an example of the same operation in IndexSQL:

```
var indexSQL=new IndexSQL("demo");
indexSQL.execute('INSERT INTO Persons (Name, IsMale) VALUES ("pipo",
true);', function(result){
    //SAVED
});
```

[1] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

End-User documentation

You can access a online copy of the end-user documentation in https://dandimrod.github.io/IndexSQL/docs .

Documentation of the Javascript IndexSQL

Class that represents a single IndexSQL database

Parameters

- `dbName` **String** The name of the database you want to open.

execute

It executes several IndexSQL queries.

Parameters

- `query` **String** A string containing the queries you want to execute, every query has to finish with a semicolon ";". For additional information about the use of queries visit https://dandimrod.github.io/IndexSQL/docs/#SQL-syntax
- `callback` dbQueryCallback Callback with the response after the queries are executed.

drop

It drops the IndexSQL database. Afterwards the object is unusable and a new IndexSQL object has to be created.

backup

It backs up the database by returning the whole Javascript object that contanins the database.

Parameters

- `callback` dbBackupCallback Callback with the backup.

restore

It restores an old backup of the database. Warning! The system wont perform any checks on this backup. An incorrect restore will make the library fail.

Parameters

- `backup` **Object** The backup of the whole database.

tableResultHeader

The headers of a result table

Type: Object

Properties

- `name` **String** Name of the header.
- `datatype` **String** The datatype of the header.
- `constraints` **string** The constraints of the header, separed by commas ",".

tableResult

The result table of a query.

Type: Object

Properties

- `header` **Array<**tableResultHeader**>** It is a list of the headers of this table and it's properties.
- `values` **Array<Array>** It is a list of the rows returned by the query, it consists of a list of the values with the same order as the header.

queryResult

The result of a IndexSQL query. It will contain at least one of these properties.

Type: Object

Properties

- `message` **String?** A message, it usually means that the query has been executed sucessfully.
- `warn` **String?** A warn, it means an error was detected, but it won't stop a transaction. Example: Dropping a table that does not exist.
- `error` **String?** An error, it means that the query was erroneus, it will stop a transaction.
- `result` tableResult**?** It represents a table generated as a result of a query.

dbQueryCallback

Callback for queries execution.

Type: Function

Parameters

- `results` **Array<**queryResult**>** An array with the results of the queries.

dbBackupCallback

Callback for backup execution.

Type: Function

Parameters

- `backup` **Object** The backup of the whole database.

Documentation of the IndexSQL language

(The symbols < and > means the parameter is optional)

CREATE TABLE

Syntax: “CREATE TABLE table_name (
column1 datatype <constraints>,
column2 datatype <constraints>,
....
PRIMARY KEY (columnName),
< FOREIGN KEY (columnName) REFERENCES foreignTableName(foreignTableColumn)>
);”

It creates the table table_name. It will contain the columns, with the datatypes and constraints specified, the constraints are optional, and they are:

- `NOT_NULL` The column value won’t be null
- `UNIQUE` The column value won’t be repeated
- `DEFAULT` If the column value is not specified, it will be the default
- `AUTO_INCREMENT` If the column value is not specified, it will be the the next one, only available with datatype number.

The possible datatypes are:

- NUMBER
- STRING
- BOOLEAN

IndexSQL will accept other SQL datatypes like VARCHAR but it will assign them one of the primitive datatypes and it won’t check the length of the columns data.

The primary key is a column that will act as the key to the whole column. It will be NOT_NULL and UNIQUE by default.

The foreign keys will be a series of keys that are related to another column in another table. It means that during the insertion, it will check if the value of the column exists on the foreign table.

DROP TABLE

Syntax: “DROP TABLE table_name;”

It will delete the table_name table and all their values. This action cannot be reverted.

TRUNCATE TABLE

Syntax: “DROP TRUNCATE table_name;”

It will delete all the values of the table_name table and leave it clean. This action cannot be reverted.

TABLES

Syntax: “TABLES;”

It will return a list with all the tables of the system.

INSERT INTO

Syntax: ”INSERT INTO table_name <(column1, column2, column3, ...)>
VALUES (value1, value2, value3, ...);”

It will insert the values into the table_name. The columns are optional, if they are not specified, it will take the values in the order the table was originally created.

If the columns contain the primary key, and the primary key matches any other primary key of the table. It will rewrite the contents.

SELECT

Syntax: ”SELECT <DISTINCT> column1, column2, ... FROM table_name <WHERE condition> <ORDER BY column1 ASC|DESC, column2 ASC|DESC, ... ;>”

It will return the values selected. To select all columns, you can write an asterisk (*) instead.

The DISTINCT parameter will make any rows that are the same to collapse into one.

The WHERE and ORDER BY statements are optional.

To see more about the WHERE statement, check this part of the documentation.

The ORDER BY will order the columns by the list provided, ASC means ascendant and DESC is descendant. If it’s not specified it will be ASC. If the columns are the same, it will check the next column in the list.

UPDATE

Syntax: ”UPDATE table_name SET column1 = value1, column2 = value2, ... <WHERE condition;>”

It will update the columns with the new values provided.

The WHERE statement is optional, if it is not specified, it will update all the rows of the table.

To see more about the WHERE statement, check this part of the documentation.

DELETE

Syntax: ”DELETE FROM table_name <WHERE condition>;

It will delete the columns.

The WHERE statement is optional, if it is not specified, it will delete all the rows of the table.

To see more about the WHERE statement, check this part of the documentation.

START TRANSACTION

Syntax: “START TRANSACTION;”

It starts a transaction, any operation performed during the transaction won’t be recorded into the database. Any errors encountered during a transaction will halt the execution of all the queries, including the ones outside the transaction.

END TRANSACTION

Syntax: ”END TRANSACTION:”

It ends a transaction, committing all the changes made during the transaction to the database.

The WHERE statement

Syntax: ”WHERE condition”

It will match the columns that check the conditions. It supports a series of operations:

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal.
AND	Logic and
OR	Logic or
NOT	Logic not
TRUE	Logic true
FALSE	Logic false
IS	Similar to =
NULL	Matches empty values
()	Used to separate between conditions

The following are some examples of WHERE statements:

- WHERE Age = 1
- WHERE Name = “Pipo”
- WHERE IsMale = TRUE
- WHERE Age > 5
- WHERE NOT (Age > 5 AND IsMale = TRUE)

SQL Coverage

From the standard SQL syntaxis, IndexSQL covers:

- [x] SQL Transactions
- [x] DB operations
- [x] CREATE TABLE Statement
- [x] DROP TABLE Statement
- [ ] ALTER TABLE Statement
- [x] SQL Constraints
- [x] SQL Comments
- [x] SELECT
- [x] SELECT DISTINCT
- [x] WHERE
- [x] AND, OR and NOT Operators
- [x] ORDER BY Keyword
- [x] INSERT INTO
- [x] NULL Values
- [x] UPDATE Statement
- [x] DELETE Statement
- [ ] TOP, LIMIT or ROWNUM Clause
- [ ] MIN() and MAX() Functions
- [ ] COUNT(), AVG() and SUM() Functions
- [ ] LIKE Operator
- [ ] SQL Wildcards
- [ ] IN Operator
- [ ] BETWEEN Operator
- [ ] SQL Aliases
- [ ] SQL Joins
- [ ] INNER JOIN Keyword
- [ ] LEFT JOIN Keyword
- [ ] RIGHT JOIN Keyword
- [ ] FULL OUTER JOIN Keyword
- [ ] Self JOIN
- [ ] UNION Operator
- [ ] GROUP BY Statement
- [ ] HAVING Clause
- [ ] EXISTS Operator
- [ ] ANY and ALL Operators
- [ ] SELECT INTO Statement
- [ ] INSERT INTO SELECT Statement
- [ ] INSERT INTO SELECT Statement
- [ ] CASE Statement
- [ ] NULL Functions

Browser compatibility

IndexSQL has been tested in the following browsers:

Chorme, Chrome for Android, Firefox, Firefox for android

Architecture documentation

Here we are going to see how IndexSQL works by reviewing the architecture.

JSON-SQL Architecture

IndexedDB is a database that stores JSON objects, that means it is incompatible with relational databases, so a JSON model had to be designed to be compatible with SQL. Here is an example of such a model:

```
{
 "Orders;OrderID;PersonID:Persons(PersonID)": {
    "OrderID;0;NUMBER;AUTO_INCREMENT,NOT_NULL": {
      "0": 0
    },
    "OrderNumber;1;NUMBER;NOTNULL,": {
      "0": 1
    },
    "PersonID;2;NUMBER": {
      "0": 0
    }
  }
}
```

As we can see the tables are the first elements of the JSON. The properties of the table are specified on the table head, separated by semicolons (;). First there is the name of the table, then the name of the primary key and then a list of the foreign keys, separated by commas (,) and with the format column:foreignTable(foreignColumn).

Inside the tables, there is the columns. On the columns keys we can see the properties separated by semicolons (;): The name of the column, the order of the column in the table, the datatype of the column and the constraints, separated by commas (,).

This is how IndexedDB can store a relational database.

IndexSQL Architecture

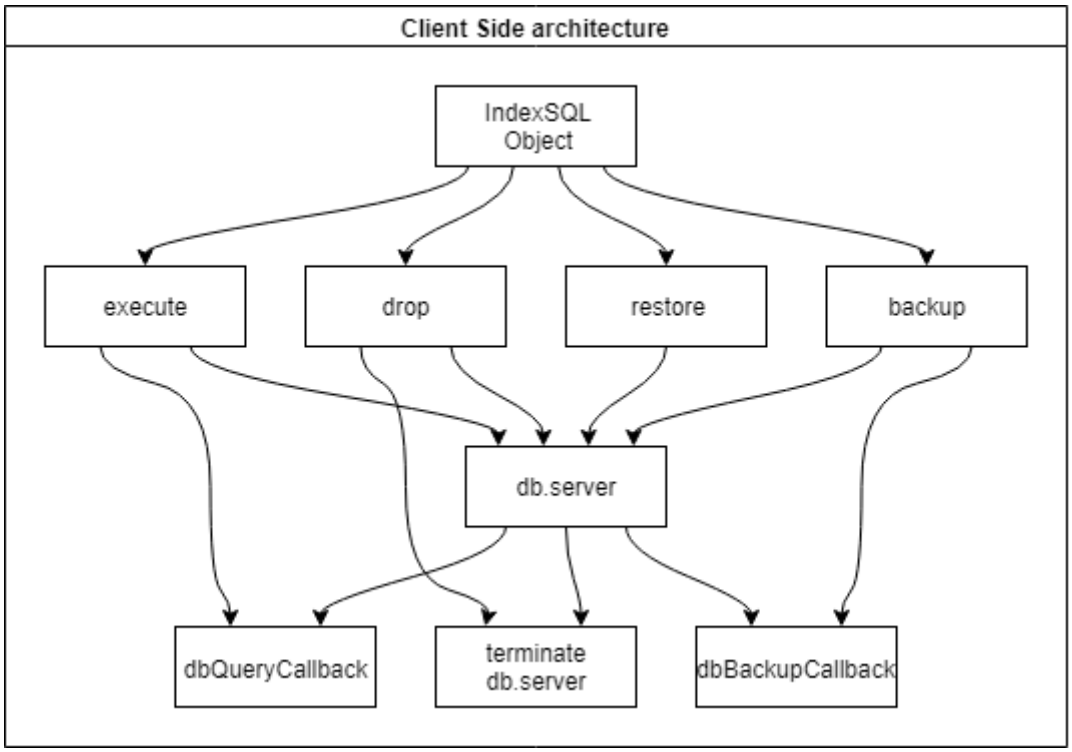
As IndexSQL must perform a lot of operations to go over all the columns of the tables many times, this means that its execution time is longer. For any other JavaScript code this would mean that the HTML DOM would be blocked, and the page would be unresponsive for the end user. You can test what DOM block is entering the JavaScript console on any webpage while loading and typing this:

```
for (var i = 0; i < 1000000; i++) {console.log('blocked!')}
```

To prevent this from happening, IndexSQL uses a technology called Web Workers. This technology allows multithreading on JavaScript. Giving that, IndexSQL will have two treads working together: the client one, that is the one where the execution of the page happens and the server one, where IndexSQL makes their operations and communicates with the database.

IndexSQL Client Architecture

Here we can see a diagram of the client architecture:



Here we can see how the different functions of the IndexSQL Object interacts with each other.

All these functions are firstly created as a part of the private db object that contains all the information of the database.

Web Worker

Firstly, the server is created and stored inside the db object with this:

```
db.worker = new Worker(URL.createObjectURL(new Blob(["(" + db.server.toString().slice(0, -1) +
 "createDB('" + dbName + "');})()"], { type: 'text/javascript' })));
db.worker.onmessage = function (e) {
    if (e.data.end) {
        db.worker.terminate();
        db.end = true;
        return;
    }
    if (e.data.backup) {
        db.backupCallback(JSON.parse(e.data.backup));
        return;
    }
    db.callbacks[e.data.id](e.data.response);
    db.callbacks[e.data.id] = undefined;
};
```

The web worked is created using the code of the db.server function. Also, the function of createDB(dbName) is called in this point so the server can prepare the database for the extraction.

At this point the message receiver of the worker is also instantiated.

If it detects the end instruction, it means that the database is dropped as shown by the diagram, so it closes the server and makes the database unusable by setting db.end to true.

If it detects that the backup instruction, it calls the db.backupCallback and sends the backup object, that was stored as a string during the transaction, as a javascript object.

If none of these happens, it means it is the result of a query, so it checks the query id, calls the db.queryCallback of the query itself. For memory management, after the use, the db.callback is set to undefined so the JavaScript garbage collector can dispose this object.

Execute

Here we can see the code of the query function, that is renamed to execute afterwards:

```
db.query = function (query, callback) {
    if (db.end) {
        console.warn("This database has been dropped already");
        return;
    }
    db.worker.postMessage({ querys: query, id: db.id });
    db.callbacks[db.id] = callback;
    db.id++;
};
```

As we can see firstly it checks if the database has been dropped, to stop any use of the function.

Afterwards it sends to the server the query and the id of it.

Then the callback is stored for later use and finally the db.id gets incremented for the next query to come.

Drop

The drop function is quite simple:

```
db.drop = function () {
    if (db.end) {
        console.warn("This database has been dropped already");
        return;
    }
    db.worker.postMessage({ drop: true });
};
```

After checking if it has been dropped already, it sends the instruction drop to the server.

Restore

In the code of the restore function, we can see:

```
db.restore = function (backup) {
    if (db.end) {
        console.warn("This database has been dropped already");
        return;
    }
    db.worker.postMessage({ restore: JSON.stringify(backup) });
};
```

First a check of the closing database.

Afterwards it sends the instruction store to the server, with a string representing the backup object.

Backup

The backup function contains:

```
db.backup = function (callback) {
    if (db.end) {
        console.warn("This database has been dropped already");
        return;
    }
    db.backupCallback = callback;
    db.worker.postMessage({ backup: true });
};
```

It basically stores the callback for later use and sends to the server the backup instruction.

The db Object

```
let db = {};
let requestDB = indexedDB.open("IndexSQL", 1);
requestDB.onerror = function (event) {
    log.warn("Database error: " + event.target.errorCode);
};
requestDB.onupgradeneeded = function (event) {
    db = event.target.result;
    let store = db.createObjectStore("databases", { keypath: "name" });
};
db.server = function () {···
};
db.id = 0;
db.callbacks = [];
```

The db object is instantiated at the start, with the creation of the database (If it doesn't exist already), then the db.server is created, that contains all the code for the server and the db.id that controls how many queries has been executed, to call the callbacks of the db.callback array.

The IndexSQL Object

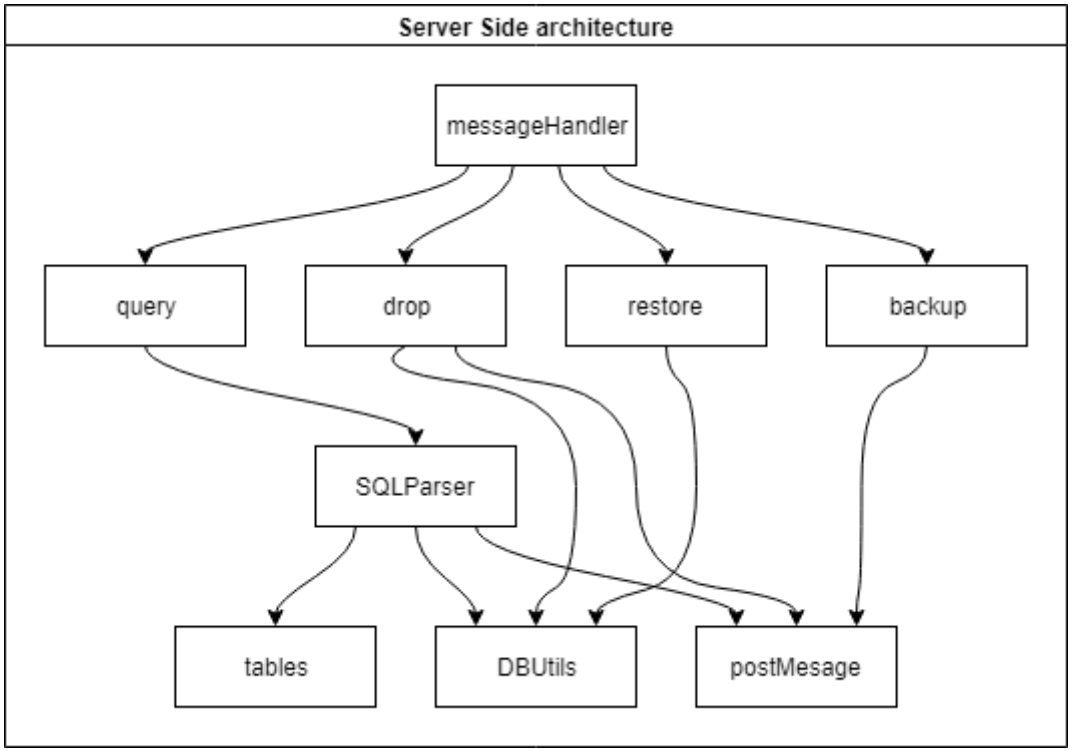
```
this.execute = db.query;

this.drop = db.drop;

this.backup = db.backup;

this.restore = db.restore;
return this;
```

The IndexSQL Object is just the renaming of the 4 functions we saw of the db object and making the class constructor return the object itself.

(If you have to review this on the code, take in mind there is a lot of documentation declared in this part too).

IndexSQL Server Architecture

First, we are going to see a diagram of the server architecture. As it is so complex, we need to simplify it as much as we can.



Post message is just the interface to send information to the client side.

The createDB function

```
function createDB(name) {
    DBUtils.name = name;
    DBUtils.load();
}
```

This function is called at the start of the server, it saves the name of the database to open in the DBUtils variable and calls for the load function.

The messageHandler Function

```
function messageHandler(e) {
    if (e.data.drop) {
        DBUtils.drop();
        return;
    }
    if (e.data.backup) {
        postMessage({ backup: JSON.stringify(data) });
        return;
    }
    if (e.data.restore) {
        data = JSON.parse(e.data.restore);
        DBUtils.save();
        return;
    }
    SQLParser.parse(e.data.querys, e.data.id);
}
onmessage = messageHandler;
```

Here we can see the handling of all operations except for the query.

The drop function calls DBUtils.drop and then returns.

The backup gets the data from the global variable data, makes it as a string and sends it back to the client.

The restore overwrites the data variable and saves it with DBUtils.save();

Finally, if the data is a query, it will be sent to the SQLParser.

The DBUtils Object

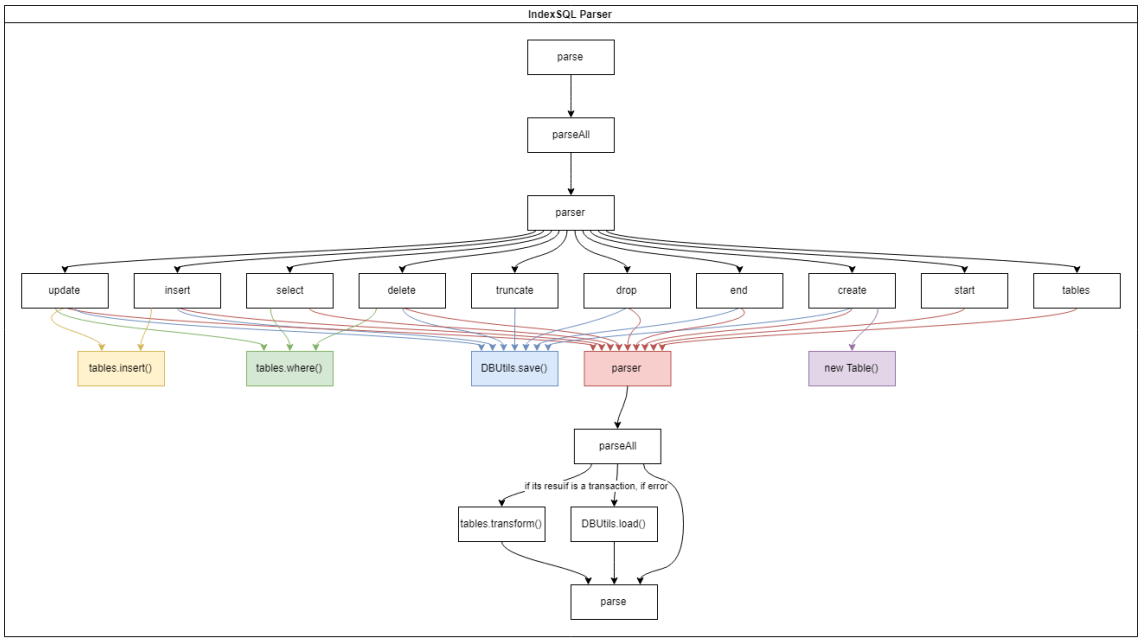
This object takes care of the interactions with the database.

```
var DBUtils = {
    name: "",
    loaded: false,
    transaction: false,
    save: function () {···
    },
    load: function () {···
    },
    drop: function () {···
    }
};
```

It stores the name of the database, if it's on a transaction and if it's already loaded. It has 3 functions, save(), that saves the data variable into the database, load, that loads the contents of the database to the data variable and drop, that deletes the database and sends a post message with the end instruction to the client.

IndexSQL Parser

As this is such an important part of the project, it has been separated into a different section even though, it is still part of the server architecture. To try and explain how it works, here is a simplified diagram of the parser:



The calls to tables.find() and tables.finder() are so numerous that they had to be removed from the diagram to make it understandable. Also, all the calls to DBUtils.save() are not performed during a transaction, except for the end one that, commits all the changes to the database.

Here we are going to describe what every function generally, but we are not going to review the code, because these functions are over 1000 lines of code long, and it would need more than this document to explain it all.

The SQLParser

Firstly, we have the **parse** function. This is the first function to be called, and it does mainly two things, it deletes all the SQL comments with this line:

```
querys=querys.replace(/(?:\/\*(?:.|\n)*?\*\/|--.*)/g,"");
```

And if the database is not loaded yet, it holds the execution until it is ready.

Next, we have the **parseAll**. It separates all the queries with a split(“;”) and with a for loop, sends them all to the *parser* function.

The **parser** is a basically a switch that classifies all the queries into the desired operation.

Now we can describe all the operations:

- **Update:** It first gets the data out of the query via a regex, as the part after the set is optional it uses a function to get the possible where. Gets the values and the columns affected. Then it compiles the where using *tables.where()*, with a for loop it goes over all the rows that are affected and takes from the database the values that are not affected afterwards it sends it all to the *tables.insert()* to insert that row. Then it tests for transaction and if not, commits the changes to the database.
- **Insert:** It first gets the data out via regex, if the columns are not specified it gets the order via the order parameter of the row key, afterwards the values are extracted and the data is sent to *tables.insert()* for insertion. After checking for transaction, compiles the changes back to the database.
- **Select:** It gets the data with the regex, as the part after the from is optional, it uses a function to extract the data, it gets the columns affected, compiles the where with *tables.where(),* Then it clones the table and removes the rows of the table that does not match the where. Finally, it compiles the order by and sends the clone of the table, the order and the distinct back to the *parser* and to the *parseAll*.
- **Delete:** Delete gets the data out of the regex, uses a function to get the where, uses *tables.where()* to compile it and loops through the database, deleting all rows that matches the where. Afterwards it checks for transaction and commits if necessary.
- **Truncate:** Truncate simply checks the syntax and loops through the columns of the table setting them back to an empty array. Afterwards it checks for transaction and commits.
- **Drop:** Drop checks the syntax and deletes the table object directly from the data variable. Then if it's not on a transaction, it stores the data back to the database.
- **End:** It sets the *DBUtils.transaction* variable back to false and commits all the changes back to the database.
- **Create:** This function checks if the table already exists, extracts the parameter and creates a new table object with the *new Table()* constructor. Afterwards it stores it in the database if needed.
- **Start:** Quite a simple operation that sets the *DBUtils.transaction* variable to true.
- **Tables:** As this is not a standard SQL database, there is no table storing all the names of the table, so I made up the command tables to get a list of all the tables. It loops trough the data object storing all the names of the tables in an object. Afterwards, it makes a virtual table with a single column: tables and sends it to the *parseAll*.

On the way back, the **parser** function does nothing, but the **parseAll** has some quirks; firstly if the returned data is of the type "response", that is, a table, it will send it to the *tables.transform()* function with the order by and distinct parameters. Then it will exchange the response from the result, exchange it with the result of the transformation and delete the order and distinct data (if any). This function will also check if there is an error during a transaction and stop the execution. Finally, it will be sent to the **parse** all the data to be returned to the client.

The tables, table object

The tables object is a few functions that assists the parser.

- **tables.find():** One of the most useful functions, it takes a container and a searchName and searches the object inside the container. As seen by the JSON-SQL Architecture, A simple container[searchName] won't work since there is more parameters than the name itself, so it searches matching with the split(";")[0] to retrieve the object.
- **tables.finder():** Similar to *tables.find()* it doesn't retrieve the object, but the entire key. This is useful if you want to check the extra parameters, but you only have the searchName.
- **tables.insert():** It inserts a row inside a table. It checks that all the values complies with all the restrictions and takes the values out of the string according to the datatype. If a primary key is not specified, it will made up an int with autoincrement as the key for the object. To update a row, you just have to overwrite the primary key. It has an extra key parameter, to overwrite objects that doesn't have primary keys.
- **tables.checkDatatype():** Called by *tables.insert()* to check if the datatype of a value is correct and to extract the value from the string itself.
- **tables.transform():** This function is quite important. It converts the result from the JSON-SQL Architecture to the more readable and accessible tableResult object. It also sorts the results afterwards by the ORDER BY specified and applied the DISTINCT if necessary.
- **tables.where():** Another quite important function, it parses from the indexSQL syntax to a condition of JavaScript code stored on a string. Then to test if a row accepts the condition, the function apply() of JavaScript compiles the code and tests the condition.
- **new Table():** This is the constructor for a table object, it makes an empty table with all the constraints according to the JSON-SQL Architecture.

Additional software

Some additional software had to be made for this project, here it's only going to be lightly explained. All the html has been made responsive by hand and no css libraries where used.

The demo.html

The demo is a full script executor and table inspector to accompany IndexSQL, it is quite handy to try it and has multiple features, like recording the time to execute the queries and to try any queries before using them in your code. Quite a neat feature. It uses the codemirror library for the code editor.

The /compiler/compiler.js

This node.js script helps with production as it helps with continuous integration. First it minifies the code from src/IndexSQL.js to a new file src/IndexSQL.min.js so the end user can easily import the script directly from the github repository.

Afterwards it also extracts the JavaScript documentation directly for src/IndexSQL.js to a md file called docs/doc.md Afterwards it converts this md file into html and combines it with the html from docs/not-index.html and saves it into docs/index.html make the documentation page.

Finally, it deletes the docs/doc.md file.

The docs/not-index.html

This file is made as the template for the docs, it has all the end-user documentation except for the JavaScript one that is provided by the compiler. It also has an autogenerated hidden menu that takes all the h tags from the html and makes this nav var to help with navigation. Finally, it also has a button to get to the top quickly.

Deployment

All the project has been released as an open source project at:

https://github.com/dandimrod/IndexSQL

And has been deployed using github pages at:

https://dandimrod.github.io/IndexSQL/

Bibliography

Here we have a list of the notable bibliography and resources used:

- https://www.w3schools.com/sql/default.asp As the main SQL syntax resource.
- https://regex101.com/ As the main regex resource.
- https://jsoneditoronline.org/ As a way to see the JSON properly represented.
- https://codemirror.net/ As the library for the code editor in demo.h