ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Database Library for WebTigerJython

Bachelor Thesis

Selin Barash

September 1, 2022

Supervisors: Prof. Dr. Dennis Komm, Alexandra Maximova
Department of Computer Science, ETH Zurich

For my beloved grandparents Nevriye and Sebahattin

Abstract

UNICEF's State of the World's Children 2017: *Children in a Digital World* report reveals that one in three internet users is younger than 18 years and 71% of 15–24-year-olds are online, making them the most connected age group worldwide. In a world into which the children are born as digital natives, we want them to understand the concepts onto which our technology is based. That is why computer science education starting from early ages has gained immense importance over time and became a part of the curriculum of public schools of all German speaking cantons of Switzerland and Liechtenstein.

The textbook "Einfach Informatik: Programmieren 7–9" teaches students core concepts of programming, and with this algorithmic thinking, using TigerJython, a Python dialect and programming environment. WebTigerJython is a web-based programming environment closely mimicking TigerJython, which can be used together with the textbook "Einfach Informatik: Programmieren 7-9". In this thesis, we extend WebTigerJython with a database library implementing all the methods introduced in the database chapter of the textbook and providing easy-to-understand error messaging. With our implementation, students will be able to solve the related exercises in the textbook using the WebTigerJython platform. We also conducted an evaluation lecture teaching the core functionalities of the database library to a small class and collected feedback. Taking this feedback into consideration, we enriched the library with new methods and functionality such as importing and exporting database tables in form of CSV files.

Acknowledgements

Words cannot express my gratitude to Alexandra Maximova, my advisor, who was my biggest support during the entire process. She was always there to help, clarify my questions, and come up with great suggestions. Without her guidance and commitment, this endeavor would not have been possible.

I am also very grateful for Prof. Dr. Dennis Komm, who was always very generous with his time and offered us help whenever we needed it. It was a great pleasure to work with him and discuss various aspects of my thesis.

I would like to express my deepest gratitude to my parents and my little brother, whose unconditional love and support always make me feel like the luckiest person on the planet.

Lastly, I could not have undertaken this journey without Ahmet Özüdoğru, whose encouragement and belief in me and my project kept me motivated along the way.

Contents

Chapter 1

Introduction

Data means knowledge and power. To take part in today's world, it is crucial to have a solid grasp of data and its management. Therefore, we want to foster an understanding of data and databases in students starting from early ages. Being familiar with data management and its processing will help them later to provide indisputable evidence to back up their arguments and make more informed decisions. At the end of the day, children own the future and rather than mere consumers, we want to raise creators who can analyze their world's problems correctly, find smart solutions, and thus make this world a better place.

1.1 Motivation

With the recent changes to the Swiss school system (i.e., Lehrplan 21 [1]), computer science started to be taught in the German-speaking cantons in Switzerland in the context of the new school subject "Medien und Informatik". The Center for Computer Science Education at ETH Zurich (ABZ) [2] produced a ten-volume textbook series for computer science classes from fifth up to ninth grade.

The textbook "Einfach Informatik: Programmieren 7–9" [6] belongs to this ten-volume textbook series and covers the core objectives of Lernplan 21 with respect to programming on the secondary school level. Its last chapter is dedicated to databases, which aligns with one of the core objectives of Lernplan 21: *The competence of structuring, recording and searching for data*.

With the exception of this chapter, all other parts of the textbook are supported by WebTigerJython [19], a web-based programming IDE for a Python dialect deleveloped by the ABZ. Taking into consideration that Lernplan 21 explicitly includes a teaching goal about databases, we saw the need for an educational tool tailored towards teaching basic concepts of databases to

secondary school students. We therefore enhanced the existing programming environment WebTigerJython with a database library.

1.2 Background

The ABZ and the Chair of Information Technology and Education have developed a spiral curriculum to teach computer science at all school levels. Programming starts already in kindergarten and primary school with Turtle Graphics using the programming language Logo in the programming environment XLogoOnline [18]. The programming environment XLogoOnline was developed at the Chair of Information Technology and Education in 2016 with two main goals in mind: First, it should be easy to deploy and second, its user interface should be simple, intuitive, and free from unnecessary clutter [7].

For higher grades, Turtle Graphics is reiterated and deepened in a real-word programming language: Python. In the later chapters of the textbook "Programmieren" of the "Einfach Informatik" series [6], the programming skills developed using Turtle Graphics are applied to other domains, for example working with lists or databases.

The programming environment TigerJython [9], also developed at the Chair of Information Technology and Education, follows the same design goals as XLogoOnline. It is a desktop application that can be used to work through all the exercises of the aforementioned textbook. However, it needs to be installed on the students' devices, which is not always feasible in a school setting. Therefore, an online programming environment was developed, closely mimicking TigerJython: WebTigerJython [19].

WebTigerJython is a web-based educational Python environment in accordance with TigerJython and the textbook "Einfach Informatik: Programmieren 7–9". It is a single-page web application, which can be run on any device that has a modern browser on it. Currently, WebTigerJython supports all the exercises from the textbook except the last chapter on databases.

Taking into consideration that databases are an inseparable part of our surrounding technology, we wanted to give students the opportunity to learn and practice the corresponding concepts in the classroom. Consequently, we implemented the database library of "Einfach Informatik: Programmieren 7–9" in WebTigerJyhon so that the exercises about databases can be addressed by the students. In the next section, we will talk more in detail about our contributions to the WebTigerJython programming environment.

1.3 Main Contribution

The main contribution of this thesis consists of designing, implementing, testing, and integrating a database library called `database1` into WebTigerJython. Except for a single exercise in the aforementioned chapter, all the other exercises can be solved using our library. Its implementation is explained in Section 3 .

In addition to the implementation, we managed to have an evaluation lecture with secondary school students, whose results will be discussed in detail in Section 4. This lecture helped us to detect some concepts that can be clarified further in the textbook and to target future work.

Besides, we enriched the current version of the library with some functionality that the textbook does not introduce but that might be useful for the students while coding. The additional methods such as `renameColumn()`, `removeColumn()` and `addColumn()` might prove very helpful while coding. Some other methods such as `tableInfo()`, `getDatabases()`, and `removeDatabase()` were created observing their necessity from students' comments and experiences during the evaluation lecture.

Moreover, the possibility to load data from and save data to database tables in WebTigerJython has been implemented. This way, it will be possible for students to import larger databases, which their teacher will provide for future exercises, and they will be able to save the current state of their database to a CSV file.

Chapter 2

Related Work

In this chapter we give an overview of two IDEs that this thesis builds on, namely WebTigerJython, a web-based IDE, and TigerJython, its desktop version. We also introduce different educational softwares that are being used to teach database concepts. From now on we refer to the textbook "Einfach Informatik: Programmieren 7–9" as the textbook.

2.1 WebTigerJython

The Center for Computer Science Education at ETH Zurich, ABZ [2], developed WebTigerJython [19], a simple graphical user interface and a free, easy-to-use programming environment. Along with the textbook, it teaches programming to high school students who have never programmed before, utilizing Turtle Graphics and the Python programming language. All functionality required for the tasks listed in the textbook, with the exception of the final database chapter, are supported by the IDE.

2.2 TigerJython and `database1` Library

TygerJython is a desktop application that provides an educational developement environment for Python programming [9]. It is built on Jython, which is a Python implementation that runs on the Java platform. All the exercises in the textbook are covered by it, including the database chapter.

The `database1` library implemented in TigerJython is written in pure Python. It includes five classes: `Column`, `Row`, `BaseTable`, `Table`, and `Database`. Each `Column` and `Row` object is bound to a unique `Table` object. Both of them have index and name attributes. To get the column or the row, the index is used. The name field functions as a caption.

The `BaseTable` class includes getter and setter methods for the existing rows and columns in the table. The `Table` class extends the `BaseTable` class. It implements methods to append and remove columns and rows, and to sort and print the table.

The `Database` class extends the Python dictionary object. Its most crucial method for assuring persistence of the variables is the `save` function(). Since TigerJython is a desktop application, it has access to computer's file system. The `pickle` library is used to serialize the object. When the object is serialized, it is written into a file that has the database's name as filename. Then, it is saved on the hard-drive using the `dump()` method of the pickle library. Every time a value is set in the database, this function is called.

When the database object is initialized, the `load` function() is called in its constructor, which then first checks whether the database file with the name that is passed to the constructor already exists. If that is the case, then this database already exists. To implement persistence, the data in this existing file is deserialized and returned.

Even though the `database1` library is implemented in TygerJython, it is not possible to simply use that implementation in the WebTigerJython environment. As mentioned above, the library is implemented in Python, which cannot be run natively in web browsers. Beside that, a web application is not allowed to access any part of the computer's file system autonomously due to security reasons. Thus, the way that TigerJython's `database1` library implements persistence would not be possible in web browsers.

2.3 Educational Software for Databases

In the following, we will discuss some of the tools and platforms that are being used to teach databases.

Tools

Professional database tools such as HeidiSQL, phpMyAdmin, and Microsoft Access are used as a "learning software" in the classroom [4].

- **HeidiSQL**
  HeidiSQL [5] is an administration tool that allows its users to view and edit data and structures stored in one of the database systems MariaDB, MySQL, Microsoft SQL, PostgreSQL, or SQLite. It provides a GUI to create and edit tables as well as to browse and query the table data.
- **phpMyAdmin**
  phpMyAdmin [12] is a software tool written in PHP that aims at handling the administration of MySQL and MariaDB over the web. Many

operations such as managing databases, tables, columns, relations, users, and permissions on MySQL and MariaDB are supported by this tool. These operations can also be performed via the user interface while the user can still execute SQL queries directly.

- **MicrosoftAccess**
  Microsoft Access [11] is a database management system from Microsoft that combines the Access Database Engine with a GUI and software development utilities.

The database management tools that we have mentioned above are not designed and created for educational purposes. They are industry-level systems that are used to manage big data. Being a professional tool, they include a lot of advanced functionalities that would not be necessary for secondary school computer science education. The variety and sheer number of buttons and interfaces would be overwhelming for the students without understanding their real functionality [4].

Moreover, having a second tool besides WebTigerJython to teach and learn would add another layer of complexity for students and teachers. The latter would need to learn these tools thoroughly to be able to help the former in case they perform an action that is not expected or there is a problem with the configuration.

Another point is that these tools require a configuration process in each device. They need some database system to be already installed to connect and to manage. Therefore, additional effort should be made to setup these tools by the school computer administrators before students start using it.

Also, they need a good understanding of databases and SQL for their competent use, which makes them hardly possible to use for introductory level lessons.

Platforms

In the following, we will introduce the published research on educational platforms for databases. Even though there is a number of such platforms, they mostly refer to higher education. Overall, it is possible to observe two main approaches in these platforms.

The first approach supports students to learn via interactive examples, illustrating basic SQL concepts. The platforms following the first approach use multimedia technologies for the illustrations. QueryViz [14] is a visualisation tool that reduces the time needed to read and understand SQL queries. It creates a table of a provided query and enables the user to visualize the relations between different tables as you can see in Figure 2.1.

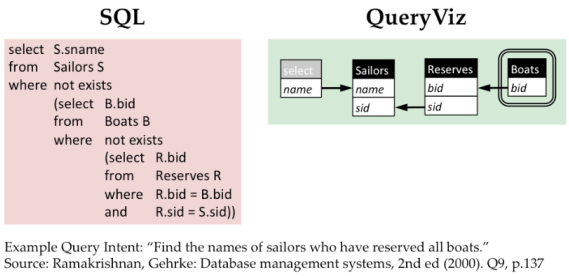

Figure 2.1: Visualisation of a nested query in QueryViz

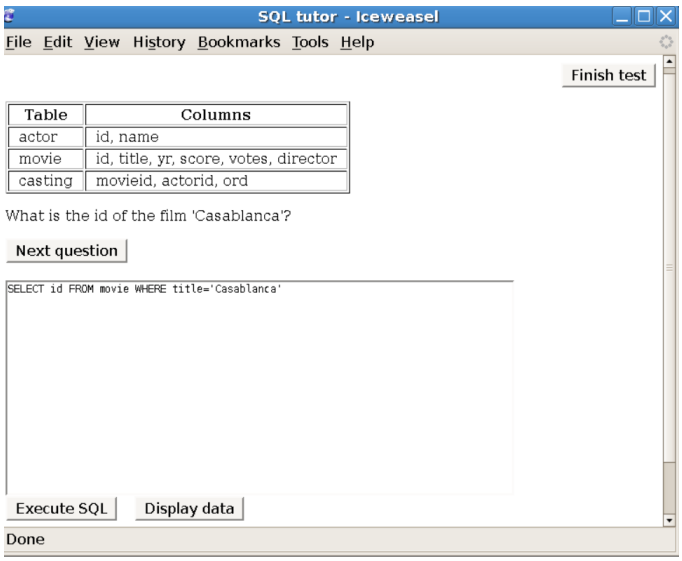

Figure 2.2: Exercise screen on SQLTutor

The second approach supports hands-on exercises, letting students learn by doing. In this approach, the students are given a question and a scenario with some tables, and they are expected to find the right query to answer the question. Their solutions are then evaluated and given feedback by the platform. Two examples of this approach would be SQL Tutor [20] and SQLator [15]. SQLTutor provides a gamified way to learn SQL. It contains two different tutorials: SQLTutor and SQLZoo. SQLTutor uses a different kind of datasets and questions whereas SQLZoo uses the ones from SQLzoo.net, which is implemented independently from SQLTutor. In Figure 2.2, you can see one of the exercise screens for SQLTutor.

SQLator is another online learning workbench that teaches SQL. It includes couple of default databases and the student chooses a query to work on and writes an SQL statement to solve the selected query. SQLator evaluates the SQL statement and provides the result; either "correct" or "incorrect". If the SQL statement was incorrect the learner may have several attempts to solve

Figure 2.3: Exercise screen on SQLator

this problem by typing in a new version of the solution. If the students is unsuccessful in solving the query after all the attemps, she or he may also access a correct solution to the problem. The Figure 2.3 illustrates the exercise screen of SQLator.

All the tools we mentioned above require a certain knowledge of SQL and are mainly targeted to teach this specific language. Most of them also use a GUI to perform the queries. Firstly, we want to focus more on the important concepts behind databases rather than on a programming language such as Python. It is already challenging for children to gain expertise in a single language. Adding another one would add quite some complexity to the current curriculum.

Also, many of the aforementioned tools use a GUI to perform database operations. The textbook is tailored for secondary school students and we believe that on this level, it would be more beneficial for the students to get familiar with full-text coding instead of using GUI. Thus, they can be better prepared for real-life coding experience.

Chapter 3

Database Library

In this chapter we discuss the design and implementation details of our database library.

3.1 Design

In this section we will talk about the process that lead to the current design of the database library we implemented.

As discussed in the Section 1.1, our main goal is to teach database concepts with a simple syntax. To achieve that, we started investigating curricula of secondary schools and high schools around the world that teach databases. We wanted to see which kind of tools and languages they use before creating our own library. There, as also mentioned in Section 2.3, we observed that most of the curricula use professional tools like HeidiSQL [5], phpMyAdmin [12] and Microsoft-Access [11] and SQL to teach database concepts. The database management tools mentioned here are not designed for educational purposes. They include a lot of advanced functionality that would not be feasible for secondary school computer science education. In particular, the variety and vast number of buttons and interfaces would be quite confusing for students without understanding their real functionality [4].

One of the concepts we originally wanted to teach was querying a database. Since we did not want to teach students yet another language (SQL) on top of Python, we considered creating a simple object-oriented syntax for queries, inspired by the numpy library. OQL (Object-Oriented Query Language) [10] is one of the real-life examples for an object-oriented query interface for traditional relational database systems.

Figure 3.1 shows an example of a possible object-oriented query. By calling `getField()` on a student table, we return the corresponding column in the table and the `greaterThan(18)` method filters the students who are older

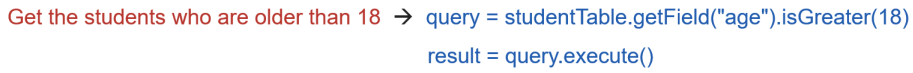

Figure 3.1: An example of an object-oriented query

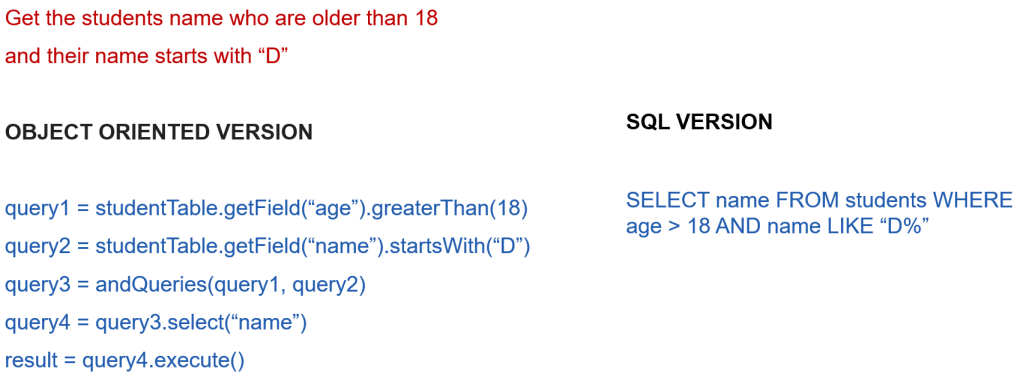

Figure 3.2: Comparison of a nested query with object-oriented and SQL syntax

than 18. When there is a single condition which is not nested, the query does not seem very complicated.

However, the syntax can become quite complicated even when only one additional condition is added. For example, in Figure 3.2, we want to get all students who are older than 18 and whose names start with ”D”. If we compare the object-oriented version with the SQL version, we can see that the former one is very long and complex even with two conditions. The more nested the query is, the longer it will take to express it using this version. Conversely, SQL syntax is much closer to natural language. It seems much easier to read and understand. Also, students are not familiar with the class concept. Therefore, calling multiple functions on the query object could be hard to explain.

After this comparison we considered a hybrid approach where students can pass SQL queries to Python methods. In this case, the WebTigerJython parser should be extended with syntax highlighting for SQL strings. However, this would again mean that the students would need to learn an additional new language. While even mastering a single programming language is challenging, it would be complicated to introduce a completely new language. Also, the time dedicated to this chapter in the curriculum is very limited and it would not be realistic to teach all the concepts about relational databases.

Considering all of the above, we decided to implement the syntax, used in the current database chapter in the textbook using a SQL library internally. We especially wanted to use an SQL library in a way that our library could be

easily extended in the future. For instance, at higher classes like in highschool, SQL might actually become a part of the curriculum.

The current database chapter in the book covers two main aspects of databases: persistence and tables. In the beginning of the chapter, databases are defined as systems to save data persistently. In the second part of the chapter, tables are introduced as a structure to save data in columns and rows.

The functionality of the database library in the book is implemented in TigerJython which we discussed in Section 2. Since TigerJython is a desktop application, it has to be downloaded and installed on every computer in order to use the database library. Since the installation process and maintaining the latest version of the application is cumbersome for teachers and administrators, we wanted to implement the database in WebTigerJython which is a web application that students can access easily using their browser.

Even though both libraries implement the same functionality, the library in TigerJython and the library we implemented in WebTigerJython are technically very different. The `database1` library in TigerJython is written in Python. The database object in this library is essentially a Python dictionary that stores data as key-value pairs. Since desktop applications have access to the file system of the computer, the library implements persistence by writing and reading the content of the database from and to files.

On the downside, web applications cannot run raw Python code client-side. They also cannot access the file system of the computer. They need JavaScript to run in the browser. That is why we decided to use the Skulpt framework to compile the Python code to JavaScript.

3.2 Implementation

In this section we will explain how we implemented the new database library for WebTygerJython.

3.2.1 Skulpt

WebTigerJython uses the Skulpt framework in order to compile Python to JavaScript. To make our library backwards compatible with WebTigerJython, we used Skulpt to implement our library as well.

We added a Skulpt module named `database1`. This module is written in JavaScript and it provides all the database commands used in the textbook. The module carries the same name as the database library in TigerJython for the students to import the library in the same way as in the textbook. Thus, the `database1` module can be imported by the line `from database1 import *`.

```
1 var mod = {}
2 mod.getDatabases = new Sk.builtin.func(function(){...});
```

Listing 3.1: An excerpt from our Skulpt `database1` module with a global function

In the beginning of the module, we initialize a dictionary called *mod*, which includes all the functions that we can call and all the classes that we can initialize in that module.

Global Functions in Skulpt

Some methods in the `database1` module such as `getDatabases()` are global and they are therefore not bound to any class.

Listing 3.1 shows an example of a global function of the `database1` module. In line 1, the empty dictionary we mentioned above is created. In line 2, the global function is created by calling `Sk.builtin.func`. This call adds the functions in the module. When a user writes `getDatabases()` in their Python program after importing the `database1` library, the `getDatabases()` is called on Skulpt's side.

Classes in Skulpt

Another key function for building up a module is `Sk.misceval.buildClass`. This function allows to create different classes in the module. In our `database1` module we have three classes: `Database`, `List`, and `Table`. These classes implement the functionality of the Python classes that students create in WebTigerJython.

When creating Python classes, it is possible to use special methods [13] to overwrite the functionality of some operators. These methods are not meant to be invoked directly by the user, but the invocation happens internally from the class on a certain action. Special methods start and end with double underscores. For example, whenever a new instance of a class is created, the `__init__` method is called, or when `print` function is invoked the `__str__` method is called internally.

The Skulpt framework also implements these special methods and allows to overwrite them. This allows flexibility with respect to how to structure the class.

The `buildClass()` method as in Listing 3.2 takes four parameters: `globals`, `function`, `name`, and `bases`. The `mod` dictionary is always passed as a global parameter, the function parameter is a function that represents the `class` object, `name` is the external name of the class, and the bases are the classes that the object is inheriting [16].

```
var mod = {}
mod.Database = Sk.misceval.buildClass(mod,function($glb, $loc){
$loc.__init__ = new Sk.builtin.func(function (self,
    databaseName){...});
$loc.__setattr__ = new Sk.builtin.func(function (self, key, value)
    {...});
$loc.__getattr__ = new Sk.builtin.func(function (self, key) {...});
}, "database1", []);
```

Listing 3.2: An excerpt from implementation of Database class in `database1` module

```
from database1 import *
DB = Database("my_database")
DB.x = 1
y = DB.x
```

Listing 3.3: An example use case where a variable is set and get from the database object using `database1` library in WebTigerJython

In line 2 of Listing 3.3, a database object is instantiated calling the `__init__()` method internally. In the next line, a variable `x` with value 1 is set in the database object calling `__setattr__()`. Lastly, in line 4, by calling the `__getattr__()` method on Skulpt's side, the value of `x` is retrieved and stored in a variable y.

Suspensions

As we will explain in Section 3.2.3, we make use of indexed database to implement persistence in our library. The operations that we perform on indexed database are asynchronous. Therefore, in some methods such as retrieving data from indexed database, we should wait until the asynchronous operation is completed and we retrieve the data successfully.

In Skulpt, suspensions are used to suspend a function to return a value before a specific execution is completed. In plain JavaScript, the `await` operator is widely used to achieve this. However, using `await` in this scenario does not work in Skulpt.

Another way to handle asynchronous processes in JavaScript is using `promises`. One can wrap a process with a promise and specify an action when the process gets resolved or rejected. Using solely `promises` could also not implement the intended behavior in Skulpt. Skulpt uses both suspensions and promises to wait until an asynchronous operation is completed and return it in a method. We will describe two different approaches to achieve that in the following.

```
1
2  $loc.exampleFunction = new Sk.builtin.func(function (self) {
3
4  var myPromise = new Promise(function(resolve, reject) {
5    setTimeout(() => resolve("done"), 1000);
6  })
7  var suspension = new Sk.misceval.Suspension();
8
9  suspension.data = {
10         type: "Sk.promise",
11         promise: myPromise
12 };
13
14 suspension.resume = function () {
15   if (suspension.data["error"]) {
16     throw suspension.data["error"];
17   }
18   return suspension.data["result"];
19 };
20
21 return suspension;
22 }
```

Listing 3.4: First way of using suspension

The first way of using suspensions is displayed in Listing 3.4. Let's assume that we want the `exampleFunction()` to return the value that the promise defined in lines 4–6 resolves or rejects. In order to guarantee this, firstly we initialize a suspension object like in line 7. Then, we define the data of the suspension as in lines 9–12. We set the promise that we want to wait for to resolve as the promise field of the data of the suspension.

Later, in lines 14–19, we define the resume field of the suspension. This is the function that is triggered when the promise set in line 11 is executed and resolved or rejected. The `if`-block in lines 15–17 specifies the behavior when the promise is rejected. Otherwise the `resume` function returns what the promise resolves. At the end, the value that the `resume` function returns propagates to the return statement of the `exampleFunction()`.

The second way of using suspensions is shown in Listing 3.5. The function `Sk.misceval.promiseToSuspension()` is a built-in Skulpt function that executes the same logic as we described in Listing 3.4. Since it makes the code shorter and more compact, we used this second approach to return the result of asynchronous operations in our functions in the `database1` library.

```
1
2 $loc.exampleFunction = new Sk.builtin.func(function (self) {
3
4 var myPromise = new Promise(function(resolve, reject) {
5   setTimeout(() => resolve("done"), 1000);
6 })
7
8 return Sk.misceval.promiseToSuspension(myPromise);
9 }
```

Listing 3.5: Second way of using suspension

3.2.2 SQL.js Library

As discussed in Section 3.1, we also want the database library to be useful for future scenarios like teaching SQL in higher classes. We also want to reduce the overhead for creating and maintaining tables manually. Using an SQL library, we would not need to implement functions such as ordering, querying, or removing in an optimized way, which is out of scope for this thesis. That is why we supported SQL in the background using the SQL.js library.

SQL.js is a JavaScript SQL database library. It allows the user to create a relational database and query it entirely in the browser. As of the writing of this thesis, it is the most used and maintained SQL library to use in JavaScript.

Advantages and Disadvantages

Since Swiss schools do not always have a fast internet connection and running the code on a server might cause potential security issues, WebTigerJython is implemented as a client-side only web application [19]. We also wanted to preserve this feature of the application. This is why one of the biggest advantages of SQL.js for our purposes library is that it is built for and works entirely on the client side. It does not require any server-side process to work.

Another point is that, in contrast to MySQL and PostgreSQL, it does not require any third-party software to run. This makes it easier to setup. After downloading the respective JavaScript file and adding it to the `index.html` file, it is ready to use.

However, the library had a drawback that caused a technical challenge to overcome: the changes made to the databases created by the SQL.js library are not persistent [17]. This means that all the modifications on the database

```
1  const SQL = await initSqlJs({
2    locateFile: file => `https://sql.js.org/dist/${file}`
3  });
4
5  const db = new SQL.Database();
6
7  let sqlstr = "CREATE TABLE hello (a int, b char); \
8  INSERT INTO hello VALUES (0, 'hello'); \
9  INSERT INTO hello VALUES (1, 'world');";
10 db.run(sqlstr);
11
12 const stmt = db.prepare("SELECT * FROM hello WHERE a=:aval AND
       b=:bval");
13
14 while (stmt.step()) {
15  var row = stmt.getAsObject();
16 }
17
18 const binaryArray = db.export();
```

Listing 3.6: Examples of most commonly used functionalities of the SQL.js library

disappear when the browser is reloaded. The reason is that SQL.js uses a virtual database file stored in the browser memory which is not persistent.

Nevertheless, the library allows to import an existing SQLite file and export the created database as a JavaScript typed array. To implement a persistent database, we combined this feature with the indexed database API that we will talk about in the next section .

Usage

The `initSqlJs` function in line 1 initializes the SQL.js library asynchronously. In line 2, the location of the files that will be used for initialization should be specified.

When the library is initialized, a new database object can be created as in line 5. It is possible to put an existing SQLite database file in the constructor of the database object to import a specific database instead of creating an empty one.

In order to run an SQL query, first the query should be defined as a string. SQL.js library also supports executing a single SQL string that contains multiple statements as you can see in lines 7–9. In line 10, the query is run without returning anything.

If the query returns a value that will be used later, then a second approach is used to execute this query. As in line 12, first the SQL statement is prepared

	Session Storage	Local Storage	Indexed DB
Capacity	5MB-10 MB	10 MB	250 MB
Accessibility	Same Tab	Any window	Any window
Expiry	On tab close	Forever	Forever

Figure 3.3: Indexed database and other web storage API

and it can be executed step by step. In line 14, we loop through each row of the table. Using the `getAsObject()` function in line 15, each table record is transformed into a dictionary object that can be used later to fetch the values of the columns in that record.

After performing some operations on the database, it is possible to export its current state using the export function as in line 18.

3.2.3 Indexed Database

In the previous subsection, we mentioned that the databases created by SQL.js library are non-persistent. However, one of the most important concepts that we want to teach in the database chapter of the textbook is the persistence of databases.

To understand the intended behavior, we can have a look at Listing 3.3. When we run the code in Listing 3.3 for the first time, we want the value `x` to be saved persistently in the database object in line 2. So that, even if we remove the line 3 and run this code snippet again, it should be possible to get the value of the variable `x` from the database and assign it to variable `y`.

Comparison of Different Web Storage API

We implemented the intended behavior using an indexed database. Indexed databases are one of the existing web browser storage APIs and we use them to store the database content we create using SQL.js. There are several ways to use the web storage API: session storage, local storage, and indexed database. In the following, we will discuss the reason why we chose an indexed database instead of other options.

Firstly, they have different expiration conditions. The session storage stores data only for the session, meaning that it is stored until the browser or even the current tab is closed. However, we want to store the data across the tabs and also across sessions. Local storage and indexed database do not have an

expiration date unless they are explicitly cleared using browser settings or via JavaScript.

Another point is that when using session storage the storage data is only accessible from the same tab. So, if the user will open another tab and try to access the same database, this would not be possible using a storage data. We want the data to be consistent between tabs. For the reasons above, storage data is not suitable for our use case.

Local storage and indexed database differ in capacity limits. The local storage can save up to 10MB data. However, some CSV files can be potentially larger. Especially for future use cases in high school, a data storage limit of 10 MB is too restrictive. For this reason, we decided to use indexed database, which has an upper limit of 250 MB.

We assume that each student has their own account, so that he or she would not be able to use another student's indexed database. We created a function called `removeDatabase()`. A database can be easily removed if necessary by specifying the name of the database in the function, e.g., `removeDatabase("my\_database")`.

Indexed Database in `database1` Library

When the first database since the import of the library is created, we open a database called `mainDatabase` in the indexed database. If it is the first time that the database is being opened, an event called `onupgradeneeded` is triggered, otherwise an event called `onsuccess` is triggered. An object store can only be created and modified in the `onupgradeneeded` handler [8]. This is a technical limitation. Outside of this handler, it is possible to add, remove and update the data in the object store but it is not allowed to change anything related to object store's properties. Hence, we create an object store named `databases` in that handler.

Both the index and key path of the object store share the same named `name`. It is important to specify the index and key path to perform queries on the object store later. In each row of this object store, the name of the database is mapped to its content which is the dumped SQLite database file. Each data that we add to the object store must contain the defined key path. This is the only restriction related to adding data to the object store. If the object store has a defined key path, the data must contain the defined keypath [21].

You can see the structure of the database we created in the indexed database in Figure 3.4. Two databases with names "database_1" and "database_2" are stored in the `databases` objects store.

Each time a new database is created, the `__init__()` method of the `Database` class is called. There are two scenarios when this happens. The first scenario

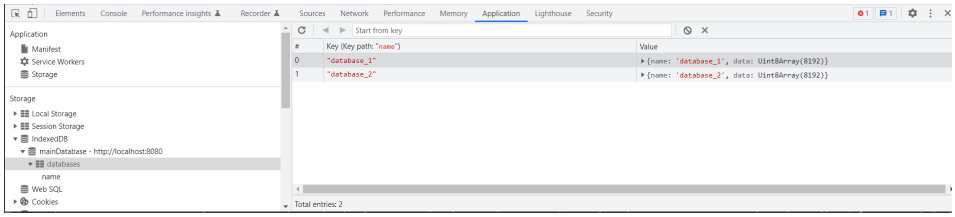

Figure 3.4: Indexed database structure with two stored databases

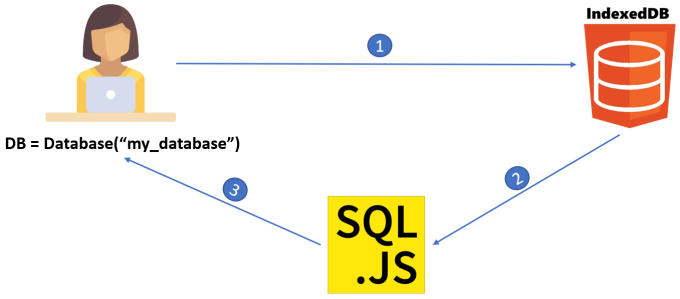

Figure 3.5: First scenario by initializing the `Database` object

is illustrated in Figure 3.5. As the first step, the `__init__()` method checks whether a database with this name already exists in the indexed database. If that is the case, it retrieves the content of that database and initializes the database object with the existing content returning the database object as third step. Then it adds the database object to a dictionary in the `Database` class mapped to its name.

We use this dictionary to prevent importing the database each time a `get()` method is called to retrieve an existing value in the database. Instead, we create a copy of the database, save it to a local variable, and then perform each of the functions that modify the database content also on this local copy of the database. Thus, we are able to read any value from the local copy without importing anything from the indexed database since it contains exactly the same modifications as the database we should have imported. This reduces the number of asynchronous operations we perform and consequently increases performance.

The second scenario displayed in Figure 3.6 is when the check in step 1 is performed and it is concluded that no database with the name provided in the `__init__()` method exists in the indexed database. In that case the indexed database returns `False` in the second step and a new empty database object is created using SQL.js. In each new database, we create a table called `variables`. This table contains three columns: `name`, `value`, and `isTable`. After creating this table, we export the database content and put it in the

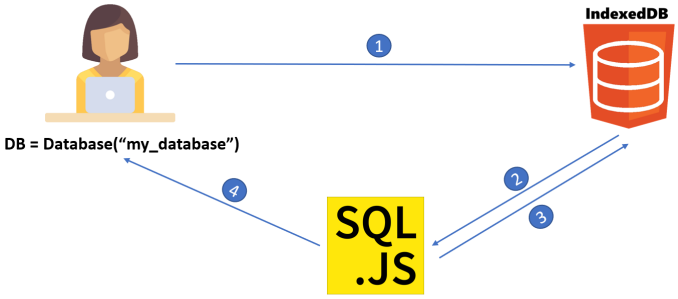

Figure 3.6: Second scenario by initializing the `Database` object

```
1 DB = Database("my_database")
2 DB.x = 5
3 print(DB.x)
```

Listing 3.7: Examples of a possible race condition scenario

object store mapped to the database name as in the third step.

Next time when a student creates a database with an existing name in the indexed database, he or she gets the already saved database content as in the first scenario. He or she modifies it with their commands, and each time he or she sets a value, the modified version of the database content is exported to the indexed database to persist the changes.

3.2.4 Promise Queue

The indexed database operations are asynchronous. By creating the object store, adding data to it, or updating it, we should wait until the operation is completed. However, on WebTigerJython students call the `database1` library methods synchronously and sequentially, which means that we should not run into any race conditions since the students would not be able to make sense of the results they are observing.

For instance, it is possible to run into a race condition in a scenario like in Listing 3.7 if we do not add further handling. First, we initialize our database object. Then, we set the value of the variable `x` to 5 in line 2. In line 3 we print the value of `x`. All of these operations are happening asynchronously.

To fix this issue, by initializing the database, we open the indexed database, have a look at the object store and check whether this database already exists. Since all these operations are asynchronous, we do not wait until they are completed but move on running the next line (as you can observe in Figure 3.7), where We want to set the value of `x` to 5. However, it is not possible to

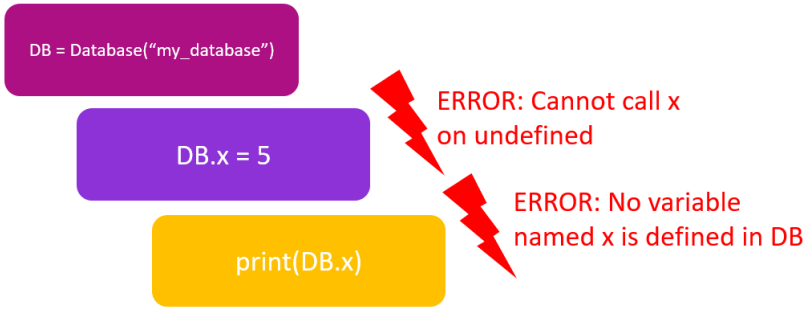

Figure 3.7: Race condition without adding any handling for sequential methods

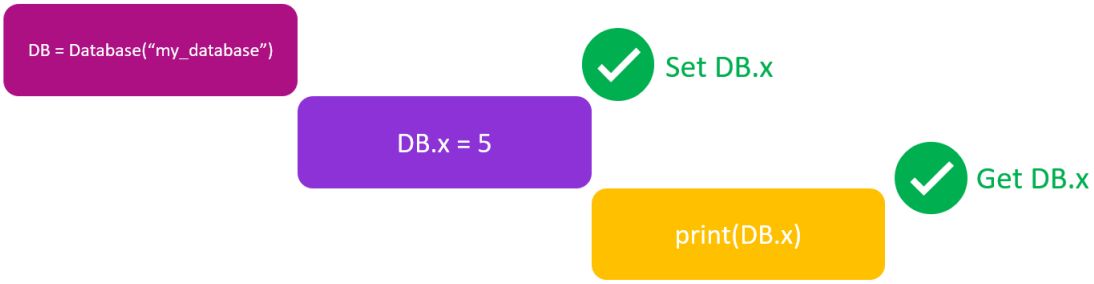

Figure 3.8: Asynchronous operations are executed sequentially after adding a promise queue

perform this action if the database that we want to operate on is not ready yet. Also, in the third line, we are printing the value of `x`. In the previous line, by setting the value of `x`, we open the object store and add or update the new value and export the database. These operations also run asynchronously. So, if setting the value of `x` is not completed yet and the updated version of the database is not loaded to the object store, we would get an error or the previous value of the variable `x`.

Not to confuse students with the results, we should perform the operations and error messages sequentially without any race conditions as shown in Figure 3.8. In the current setup, this does not happen automatically. We need to guarantee that only when the current operation is completed we can move onto the next operations .

To solve this problem, we used a dynamic promise queue, which is essentially a FIFO queue that processes the operations in order.

Except from the `__iter__()` and `__str__()` methods, we wrap each method on Skulpt's side into a function that does not take any input parameter and returns a promise that contains the current wrapped method's code in its execution code, and enqueue this wrapping in the promise queue. This is

important since when the promises are created, their execution code runs automatically but as we want to put the operations in order, we need to pause this execution. This is why we do not simply add the promise itself but a wrapped version of it.

When we enqueue the wrapping we mentioned above, the enqueue function returns another promise. This promise is the wrapper promise of the real promise that we would like to execute. In the execution code of this wrapper promise, a dictionary with three elements is pushed to the queue. The first element of the dictionary is the real promise. The second and third elements are the resolve and reject functions of the wrapper promise. After enqueueing this dictionary, the dequeue function is called.

The dequeue function first checks whether there is any other process left in the queue. If not, it returns. After that, it checks whether there is a promise that is currently being executed. In that case, the new promise should wait until the current one is resolved or rejected. If the queue is available to process the next element, then the first element of the queue is dequeued after which the `workingOnPromise` flag is set and the execution of the promise is started.

The popped element is a dictionary that includes the three elements we discussed above. Firstly, the first element is called, which means that the real promise whose result we are waiting for is executed. If it returns successfully, the `then()` method is executed, otherwise the `catch` block is executed. In the `then` block, the `workingOnPromise` flag is unset so that other promises can be executed later. Then the resolved value of the executed promise is assigned to the wrapper promise's `resolve()` method, so that it will be returned at the end. At the end, the `dequeue()` function is called to evoke other promises waiting for this promise to be completed in the queue. In the `catch` block the `workingOnPromise` flag is unset as well, but this time the error is sent to the upper wrapper function to return in the `reject()` function.

Chapter 4

Evaluation

In this section, we will talk about the lecture that we conducted to evaluate our library. Firstly, we will give an overview about the lecture setup, then describe the questionnaire we handed in to the students at the end of the lecture and discuss its results. Besides, we will share our observations about the lecture and explain the adjustments we made to our library according to these observations.

4.1 Evaluation Setup

We evaluated our database library with nine students in Cantaleum Zurich in an 90-minutes lecture. There were six male and three female students in the class whose ages were from 12 to 14. The students had previous experience with Python programming. In the first lecture, we mainly talked about the relevance of databases in our world and explained the main functionalities of the library introduced in the textbook. To explain the main functionality, we gave the students basic exercises that they could try themselves. The introductory exercises are shown in Listing 4.1. We will talk about them in more detail in the next paragraph. We covered the following topics during the first lecture:

- Creating a database and saving a variable in the database permanently
- Creating a permanent and non-permanent table
- Appending records to a table
- Creating a `for`-loop that runs over the records in the table

After brainstorming with the students about why persistence might be important for databases, we wanted to show them how we can create it using our database library. The first exercise is displayed in the lines 2–4 in Listing 4.1. It was to create a database with a name that students chose themselves

```
1  from database1 import *
2  #Exercise1
3  DB = Database("my_database")
4  DB.x = 10
5  print(DB.x)
6
7  #Exercise2
8  students = Table("Name", "Favorite number")
9  students.append("John", 23)
10 students.append("Jane", 24)
11 print(students)
12
13 #Exercise3
14 DB.students = Table("Name", "Favorite number")
15 DB.students.append("John", 23)
16 DB.students.append("Jane", 30)
17 print(students)
18
19 #Exercise4
20 for x in DB.students:
21   if x["Favorite number"] > 25:
22     print(x["Name"])
```

Listing 4.1: The exercises we went through during the first lecture

and to save a variable `x` in it that has value 10. For them to realize that the value is persistently saved, we deleted line 3 and printed the variable `x` again. Even though the value assignment did not happen, they were still able to see the value of `x` printed.

In lines 6–9, we can see the second exercise. As discussed in Section 3.1, organising data in tables was another important concept of the database chapter in the book. Firstly, we created a table with the students on a PowerPoint slide, and then coded that table in WebTigerJython. The students learned how to use the `append` function to add records to the table.

However, the table in the second exercise was non-persistent. As a third exercise, we showed them how to modify the code to persist the table in a database. The code in lines 11–14 displays this third example. Like in the first exercise, we deleted the lines 12 and 13 and then printed the table again to showcase the persistence. As the last exercise which you can see starting from line 16, we explained how to create a `for`-loop that would run over the records in a given table, which creates a base for the challenge we prepared for the second lecture.

During the second lecture we prepared an engaging challenge for the students to apply what they learned during the first lecture. Before the lecture, we

(Nick)name: ..... Klasse: .....

Was hast du lieber?	
☐ Waffeln	☐ Pancakes
☐ Videospiele	☐ Sport
Wo würdest du lieber in die Ferien gehen?	
☐ Hawaii	☐ Alaska
Wenn du eine übermenschliche Fähigkeit haben könntest, was hättest du lieber?	
☐ Unsichtbar sein	☐ Gedanken lesen

Figure 4.1: Question sheet consisting of four either-or questions we prepared for the challenge

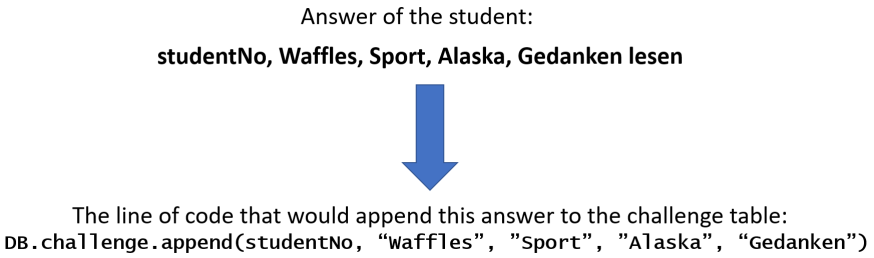

Figure 4.2: The conversion from students answer to append function

prepared a sheet consisting of four either-or questions (see Figure 4.10) and let students fill it in. The goal was to create a table called challenge in a database and append all the answers to this table to perform some queries on it later. We distributed the filled sheets among the students and explained them how to convert the answers in the sheets to an `append` function. Figure 4.2 displays how the conversion from students answer to append function works.

After each student turned their portion of the answers into arguments for an `append` function, they sent these lines to their common group chat. Later, all the students copied these lines to their coding environment. When we made sure that they all saved the records in the challenge table, we gave them a few tasks, which were finding out how many people liked one of the answers of either-or questions for each question. For example, for the first group the task was to find out how many people liked waffles instead of pancakes, for the second group to find out how many people preferred doing sports

Question	Yes	No
Was it easy to work with ?	2	7
Have you understood how the library works ?	6	3
Did you have confusing errors that you did not know how to handle ?	7	2

Figure 4.3: Results of three yes-no questions in the questionnaire

instead of playing video-games etc.

At the end of the lecture, the students were supposed to present their findings and how they got the respective answers, but we had to skip this part due to time constraints. We also handed out a questionnaire about the experience with the database library after the lecture.

4.2 Evaluation Findings

In this section, we will discuss the content and results of the questionnaire about our database library.

4.2.1 Questionnaire

The questionnaire consists of ten questions of three different types. There are three yes-no questions, two rating scale questions and five open-ended questions about the database library. In the following, we will list all the questions and their results.

Yes-No Questions

The first yes-no question was whether it was easy to work with the library. As you can observe in Figure 4.3, two students answered that it was easy to work with it while seven of the students found it hard to work with. As the second yes-no question, we wanted to know whether the students understood the working principles of the database library. The majority of the students replied that they understood how the library works. The last yes-no question was about getting confusing errors that they did not know how to handle. Seven students stated that they came across confusing errors while two students answered that they were to able understand the error and fix the problem.

Figure 4.4: The rating scales in the questionnaire

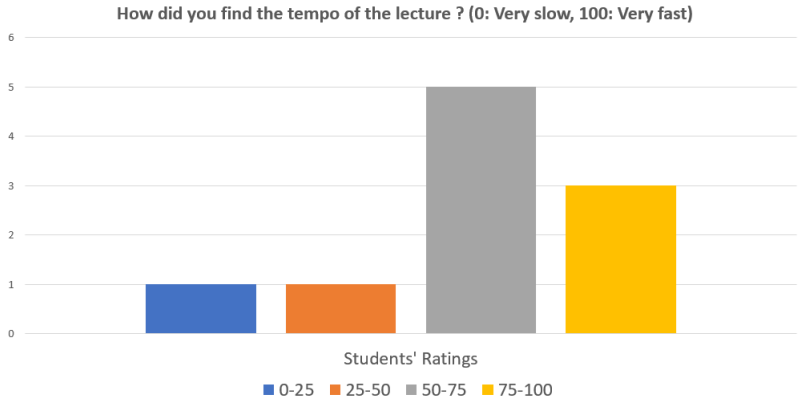

Figure 4.5: Students' ratings about the tempo of the lecture

Rating Scale Questions

We asked two rating scale questions about the tempo of the lecture and the level of difficulty. The Figure 4.4 illustrates the rating scale we used. In order to get a numerical value out of the scale, we firstly measured the entire bar length. Then, we measured the distance from the left end of the bar to where the student marked. We divided that value by the total length. Lastly, we multiplied that value with 100 for simplicity.

When we applied the above operation for each answer, we obtained the bar graphs in Figures 4.5 and 4.6. The first graph shows the students' ratings of lecture's speed. The vertical axis of the graph represents the number of students and the horizontal one the percentage we calculated as described above. 0 means that the student found the lecture very slow and 100 very fast respectively. Two of the students rated the tempo of the lecture slower than 50 whereas most of the students thought the lecture was faster than 50. Altogether, students' ratings about the speed are clustered mostly in the 50–100 interval.

The second bar graph in Figure 4.6 displays students' rating with respect to the level of difficulty of the database library. Three students found the difficulty average and below the average whereas six students thought that the library was harder than 50. Overall, the students thought that the lecture was fast and difficult which is understandable considering the density of the material in a 90-minutes lecture. Possible reasons will be discussed in the

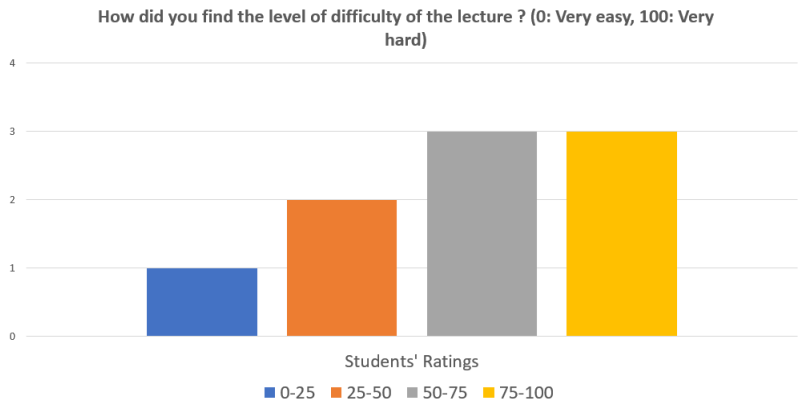

Figure 4.6: Students' ratings about the difficulty of the library

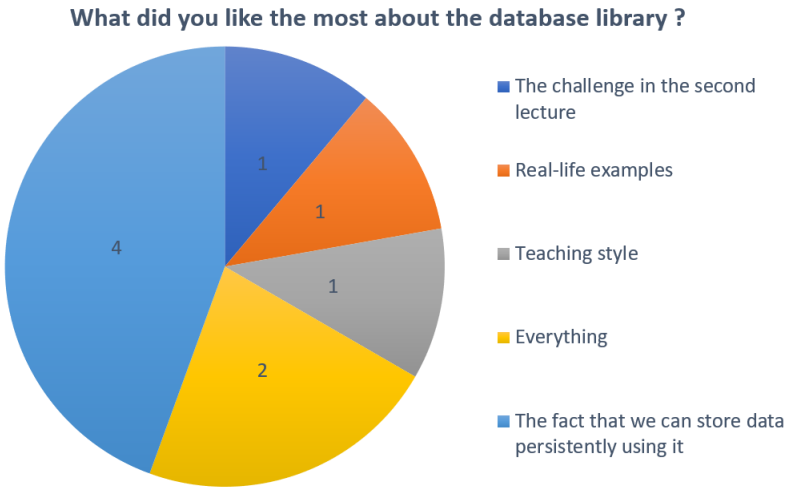

Figure 4.7: Results of the question "What did you like the most about the library"

Subsection 4.2.2.

Open-Ended Questions

The pie chart in Figure 4.7 illustrates the students' answers to the question about their favorite aspects of the library. From the graph it is clear that majority of the students liked the persistence of the database library the most. They commented that it was great to be able to store a value persistently via database library. Two of the students answered that they liked everything about the library. The rest of the students mentioned the assistance during the lecture, the challenge in the second lecture, and learning the relevance of the topic in real-life as what they liked the most.

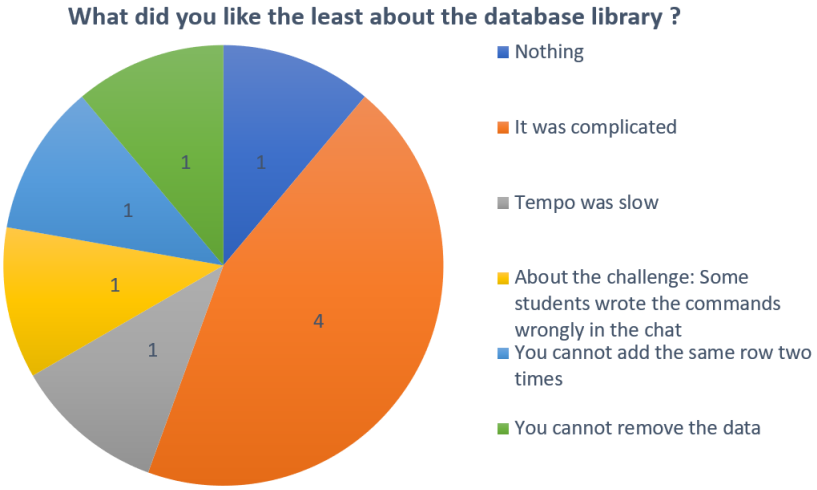

Figure 4.8: Results of the question "What did you like the least about the library"

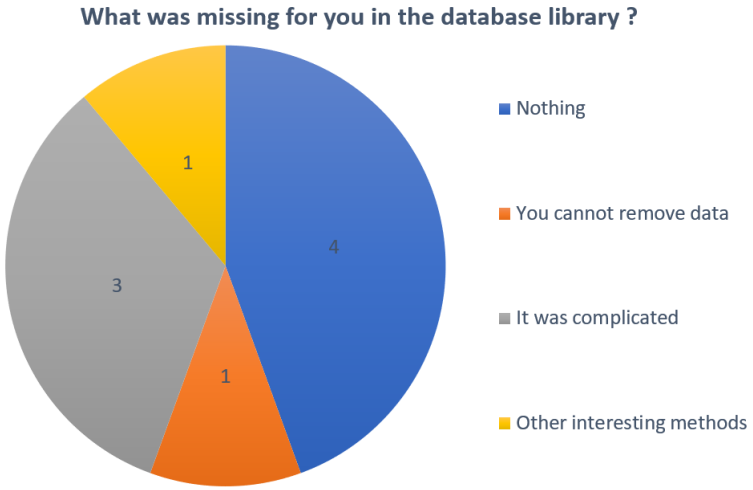

Figure 4.9: Results of the question "What was missing for you in the database library?"

The second open-ended question was about what students liked the least about the library. Nearly the half of the students answered that its complexity was the least pleasant aspect of the library, which is consistent with the results of the bar graph in Figure 4.6. Other than this answer, all the other answers had equal appearance in the questionnaire about this question. Some students found the speed of the lecture slow. Individual students complained about some students not being able to convert the answers in the sheets correctly to an append function. Students also mentioned that they did not like not being able to add the same row twice or to remove the data. Another student commented that there was nothing that annoyed them about the library.

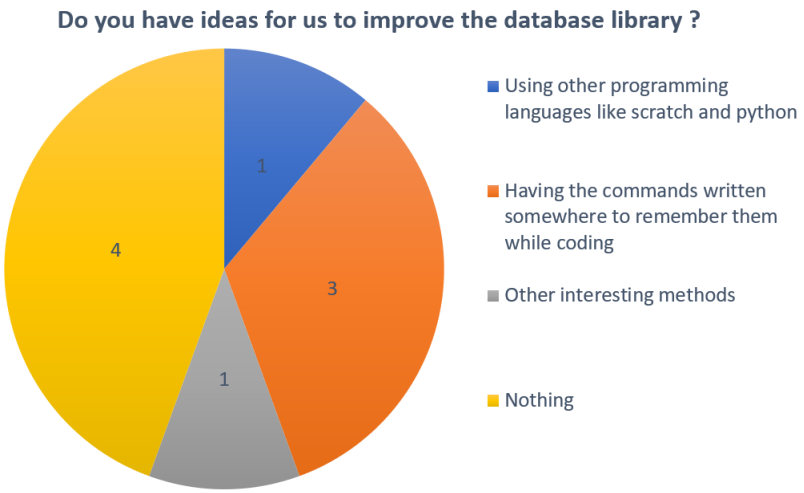

Figure 4.10: Results of the question "Do you have ideas to improve the database library?"

As the third open-ended question, we wanted to know whether something was missing in the database library. Most of the students did not mention any functionality that was missing. Three of the answers mentioned that the library was complicated. The rest of the answers indicated that the students would like to have other interesting methods and a method to remove data from the database.

Overall, we can deduce that the complexity of the library was the most disliked aspect. We will provide some possible reasons why the students think in this direction, explaining the observations we made during the lecture in Section 4.2.2.

Not being able to add the same row twice in a table is one of the conventions we have in the database library. By having this convention, we aimed at preventing the students from running the code a couple of times without realizing the side-effect of appending a record to a table. Also, we already implemented a function to remove a specific record from a table but none of the students asked whether it exists during the lecture. However, the corresponding comment indicated the necessity of this method, which we already included in our library.

Lastly, we asked the students whether they had ideas for us to improve our database library. The majority of the answers did not contain any idea. One third of the answers mentioned that it would be practical to have the commands written somewhere to look them up while coding. Individual answers indicated that one could use other programming platforms and languages like Python and Scratch and add other interesting methods.

We took the idea about displaying the commands somewhere that the stu-

dents can reach easily into consideration. To address this issue, we added the database library commands to the information section where all the methods in WebTigerJython are listed.

Regarding the idea about adding other interesting methods to the library, we could not introduce all the methods we have in the library during the lecture due to time limitation. If these interesting methods are referring to statistical shortcut methods such as sum, average, maximum, etc., then they are not implemented on purpose. One of the main goals of Lernplan 21 is to support children to understand the building blocks of programming, so that they can implement such methods themselves instead of simply using built-in solutions.

Also, one of the core programming principles in the textbook is to promote text coding. We use an enriched version of Python in the WebTigerJython platform. Therefore, we did not act on the comment about using other languages and platforms like Python and Scratch which uses block coding.

4.2.2 Observations

One of the main confusions for students was to understand what happens when a new database instance is created. Even though they understood where and how databases are used in their everyday life, they were not able to understand what happens when two different databases are created. In this aspect, it would be useful to give tangible analogies for them to understand the scope of databases and variables that we save in them. This confusion can be clarified by adding a new section to the current database chapter in the textbook where a respective analogy is introduced and an example is added to reinforce it. We also added a function called `getDatabases()` that returns the list of names of all existing databases.

An example analogy could be mapping two databases to two different cities. The variables and tables stored in the databases can be streets and the records in table can be the buildings on the street. It is possible that two cities have a street with the same name. For instance, both Zurich and Bielefeld have a street called Bahnhofstrasse. Even though both streets have the same name, the buildings on these streets are different; just as two databases that have a variable with the same name but having different values. The coding example in Listing 4.2 could help students to understand the borders of two different databases.

In lines 2–3 and 5–6, two different databases called `my_database_1` and `my_database_2` are created, and a variable with the same name, namely `x`, but with different values, is saved in each database. First, the teacher could let the students guess the results of the print functions in lines 8 and 9 and later he or she could let them code and compare the result with their answer.

```
from database1 import *
db1 = Database("my_database_1")
db1.x = 10

db2 = Database("my_database_2")
db2.x = 20

print(db1.x)
print(db2.x)
```

Listing 4.2: An example which could clarify the confusion about creating multiple databases

This exercise would be helpful for the students to see that two databases are two different entities. The confusion could also be of more general nature and corncern understanding the scope of the variables. Learning about local variables could be useful to differentiate between two different database entities.

Another source of confusion was that without visual feedback, students had a hard time to understand the side effects of `createTable()` and `append()`. Our library does not allow to create a table with an existing name or append a record with the same values to the same table. So, if a student executes the line `DB.table = Table("Column1")` for the first time, the table is created with the respective columns in the database. The second time the student runs the same program, the same line will throw an error indicating that this table already exists. The same logic applies to appending a record to a table. The exact same line of code once running successfully and once throwing an error was not very intuitive for students.

An editor in which students can see the modifications they create in the database would be ideal to solve this issue. Due to time limitations, we were not able to implement such a tool. However, the print function applied to an instance of `Table` class prints each record of the table to the console. Also, calling the methods `getVariables()` and `getTables()` on a database object would return the names of the tables and values stored in that database. In addition, we have the `getTableInfo()` function that returns a string including the name of the table, the number of columns, and the columns in order. Using these methods students have a chance to check the state of their databases and tables.

In addition, the challenge that we created for the second lecture also had some operational obstacles. As one of the students mentioned in the questionnaire, some of the students were not able to transform the information on the sheet to an append function call correctly. It would be very helpful if there was a file that included all the information and the students could use that file for their exercise. To prevent this kind of issue in the future, we implemented an

import and export database table functionality. Using import functionality, the students can import a CSV file into a table in a database and perform queries on it.

The last point is that the students really seemed to enjoy the introduction of the lecture where we talked about the relevance of databases for our daily lives. We started the lecture asking whether they know where all the photos and stories in their social media accounts or the scores on their favorite online games are stored. Also, we created a mind-map together to note down everything that comes to our minds when talking about databases. Seeing their participation and excitement in that part, we think that the introduction to the database chapter in the book can be enriched by more recent and relevant examples.

Chapter 5

Conclusion, Limitations, and Future Work

As mentioned in the introductory chapter, computer science has become a part of the curriculum of obligatory schools of all German speaking cantons of Switzerland and Liechtenstein in the context of the new school subject "Medien und Informatik". An easily accessible web-based IDE called WebTigerJython, which can be used on multiple kinds of devices via modern browsers, has been implemented for the lectures of this subject. It allows the students to solve all the exercises in the textbook except from the last chapter related to databases.

5.1 Conclusion

In this work, we firstly investigated the current curricula and existing tools to teach databases. Due to reasons we expressed in Sections 2 and 3.1, we decided to implement all the methods that the database chapter in the textbook offers. We conducted an evaluation lecture where we had the opportunity to let students work with our library and observe their interaction. Besides, we asked them to fill out a questionnaire that we extensively analyzed in Section 4.

The major challenges of this thesis were discovering the Skulpt environment which does not provide a very extensive documentation, implementing persistence using the SQL.js library which is inherently non-persistent and performing sequential execution of the asynchronous processes. We explained in detail how we solved all these issues in Section 3.2, sharing our learnings with everyone who will improve WebTigerJyhon in the future.

Even though the database chapter is the last chapter of the textbook and it is unfortunately hard to make time for it because of the already dense program, it was great to observe the students' excitement about databases

and their real-life usage. The students seemed interested hearing that they use databases on a daily basis in their social media accounts and favorite online video games. Even after a 90-minutes lecture they understood how to store data persistently such that the day after our lecture a student asked whether they could use the library in a robotics context to save the scores of the game they coded. Considering all this and the undeniable power of data in today's world, we believe that databases are an engaging topic for students.

With our work we hope to contribute to computer science education in a way that they can get a firm grasp of database concepts. We find it extremely important that children will start to gain an understanding of these concepts at an early age. This way they will be able to query data, analyze it, and thus stick stronger to facts than false information.

5.2 Limitations

Our database library implements all the methods used in the current database chapter, however, it has a few limitations. In this section, we discuss these limitations and describe possible solution approaches.

5.2.1 The Last Exercise In the Textbook

The only exercise that cannot be solved using our library in the database chapter is the seventh exercise, which is displayed in Listing 5.1. In the beginning of this exercise, it is mentioned that it is allowed to append a record that has number of values smaller than the column number of the table. For instance in line 12, a record without the value of the "Schnitt" column is appended to the table. However, this approach does not allow students to leave out the value of any other column than the last one. So, if the student wants to leave out the value of column "Noten" and only append the values of "Fach" and "Schnitt" in the append function, the value for the "Schnitt" would be saved in the "Noten" column.

A possible solution for this issue could be introducing Python keyword arguments and specify the column name as a label for each value in the append function. However, Python keywords are not introduced in the textbook. To leave out such confusing scenarios, we only let appending a record to a table if the number of values in the append function is the same as the number of the columns of the table and else throw such an error: "The table X has Y columns but only Z values are supplied. Please use tableInfo() command to check the structure of your table. If no value should be inserted, you can enter None."

```
1  from database1 import *
2
3  def berechne_schnitt(eintrag):
4      summe = 0
5      anzahl = 0
6      for note in eintrag["Noten"]:
7          summe += note
8          anzahl +=1
9      eintrag["Schnitt"] = summe/anzahl
10
11 noten_table = Table("Fach","Noten","Schnitt")
12 noten_table.append("Informatik", [5.5,6,5,5.5])
13 noten_table.append("Informatik", [4,4.5,4.25])
14
15 for eintrag in noten_table:
16     berechne_schnitt(eintrag)
17
18 print (noten_table)
```

Listing 5.1: The seventh exercise in the database chapter

Another limitation about this exercise is directly setting a column value as in line 9 in Listing 5.1. In the current implementation, in the for loop of the `Table` class, we return a built-in Skulpt dictionary to iterate over. In the built-in Skulpt dictionary the set operation is not persistent. It would therefore not be possible to modify the column value persistently in line 9 in Listing 5.1 as expected.

We can solve this issue by creating a customized `Dictionary` class in the current Skulpt module. This class would extend the built-in Skulpt dictionary and overwrite the `set()` method of the dictionary to export the database content after modifying the table column value.

5.2.2 Error Messaging in Global Methods

There are two global methods in the `database1` library as discussed in Subsection 3.2.1, namely `getDatabases()` and `removeDatabases()`. Currently, even though we could throw an error when there are no existing databases, the error is not propagated to the student's screen. We can only see the error message in the developer console.

Besides, we call the `databases()` method of the `IDBFactory` interface to get the list of all the databases from the indexed database in both of these methods. This method is supported by all modern browsers except for Firefox [3]. Therefore, these two methods do not work on the Firefox browser.

5.3 Future Work

As the evaluation results in Section 4 show, a simple database editor, compatible with the database library, would be very useful for the students to see the side-effects of operations such as creating a new database or adding a record to an existing table. The editor should be capable of illustrate the data in table form, and display the imported and exported tables.

In addition, we do not support to save list of lists in the database currently. A support for multi-dimensional lists can be added to the database library as a future work.

Bibliography

[1] Lehrplan 21. Modul Medien und Informatik. https://v-fe.lehrplan.ch/index.php?code=b|10|0&la=yes. Accessed: 2022-08-18.

[2] ABZ. Nachhaltige Vermittlung von Wissen im Bereich Informatik. http://www.abz.inf.ethz.ch/. Accessed: 2022-08-18.

[3] MDN Web Docs. Idbfactory.databases(). https://developer.mozilla.org/en-US/docs/Web/API/IDBFactory/databases. Accessed: 2022-08-31.

[4] Andreas Grillenberger and Torsten Brinda. eledSQL – a new web-based learning environment for teaching databases and SQL at secondary school level. In *Workshop in Primary and Secondary Computing Education (WIPSCE '12)*, pages 101–104. ACM, 2012.

[5] HeidiSQL. Administration tool for databases. https://www.heidisql.com. Accessed: 2022-04-15.

[6] Juraj Hromkovič and Tobias Kohn. *Einfach Informatik. Programmieren*. Klett und Balmer Verlag, 2018. School Textbook for Grades 7–9.

[7] Juraj Hromkovič, Giovanni Serafini, and Jacqueline Staub. XLogoOnline: A single-page, browser-based programming environment for schools aiming at reducing cognitive load on pupils. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives (ISSEP 2017)*, volume 10696 of *Lecture Notes in Computer Science*, pages 219–231. Springer, 2017.

[8] javascript.info. On creating object stores. https://javascript.info/indexeddb. Accessed: 2022-08-01.

[9] Tobias Kohn. *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. PhD thesis, ETH Zürich, 2017. Accessed: 2022-08-01.

[10] Zhongling Li. Object-oriented query language. http://www.cs.columbia.edu/~sedwards/classes/2006/w4115-spring/proposals/OQL.pdf, 2006. Accessed: 2022-08-19.

[11] Microsoft. DBMS developed by Microsoft. https://www.microsoft.com/en-ww/microsoft-365/access. Accessed: 2022-04-15.

[12] phpMyAdmin. Software tool written in PHP to manage database over the web. https://www.phpmyadmin.net. Accessed: 2022-04-15.

[13] PythonDocs. Python docs special methods. https://docs.python.org/3/glossary.html#term-special-method. Accessed: 2022-07-27.

[14] QueryVis. Queryvis, visualization for SQLqueries. https://queryvis.com. Accessed: 2022-08-18.

[15] Shazia Sadiq, Maria Orlowska, Wasim Sadiq, and Joe Lin. SQLator – an online SQL learning workbench. https://dl.acm.org/doi/pdf/10.1145/1026487.1008055. Accessed: 2022-08-18.

[16] SkulptDocs. buildClass explanation. http://skulpt.org/docs/index.html. Accessed: 2022-07-27.

[17] SQLJsLibrary. Documentation of sql.js library. https://sql.js.org. Accessed: 2022-07-30.

[18] Jacqueline Staub. XLogoOnline – a web-based programming IDE for Logo. Master's thesis, ETH Zürich, 2016. Accessed: 2022-08-01.

[19] Nicole Trachsler. WebTigerJython – a browser-based programming IDE for education. Master's thesis, ETH Zürich, 2018.

[20] SQL Tutor. Web-based interactive tutorial of SQL. https://www.gnu.org/software/sqltutor/manual/sqltutor.html. Accessed: 2022-08-19.

[21] web.dev. On key paths. https://web.dev/indexeddb/. Accessed: 2022-08-01.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.