DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient and Interactive Data Analytics with WebAssembly

Karl-Pablo Sichert

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient and Interactive Data Analytics with WebAssembly

Effiziente und Interaktive Datenanalyse mit WebAssembly

Author:	Karl-Pablo Sichert
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	André Kohn, M.Sc.
Submission Date:	September 15, 2020

Abstract

Data analytics is resource-intensive. At the same time, users expect that analytics software quickly returns results to their queries. In this thesis, we explore if WebAssembly is a suitable choice to build data analytics products that can satisfy high expectations regarding efficient query execution. WebAssembly is an emerging technology that allows running code with near to native speed in a web browser.

Further, we are interested if data analytics tools can make creating dashboards more interactive while storing them in a concise human-readable representation. We present a dashboard programming language that can be authored both in a graphical and textual mode. The language is modeled as a superset of SQL, can state data requirements and analytical problems in a declarative way, and subsequently create data visualizations.

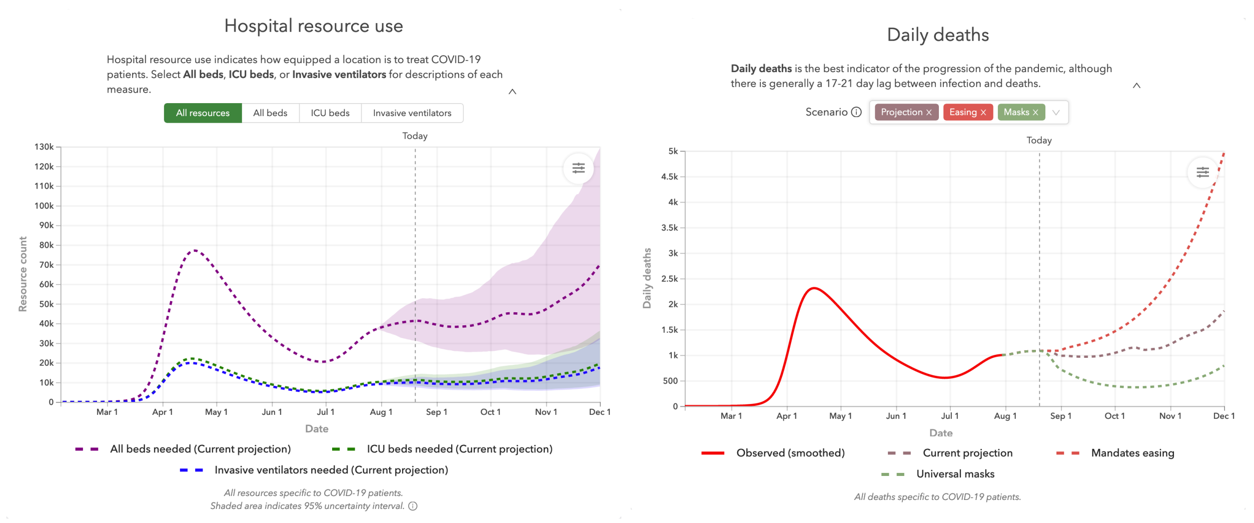
We find that our programming language is equally expressive as other popular browser-based data analytics tools. However, we outperform them by one order of magnitude for common analytical tasks, thanks to running a query engine optimized for analytical query workloads in WebAssembly.

Contents

List of Figures

1 Introduction

Data science is a field where new insights can be produced by analyzing a vast amount of information. These insights are useful for backing up theories by factual data or, more generally, allow individuals and organizations to take a data-driven approach to decision making.



Hospital resource use

Hospital resource use indicates how equipped a location is to treat COVID-19 patients. Select **All beds**, **ICU beds**, or **Invasive ventilators** for descriptions of each measure.

All resources · All beds · ICU beds · Invasive ventilators

All resources specific to COVID-19 patients.
Shaded area indicates 95% uncertainty interval.

Daily deaths

Daily deaths is the best indicator of the progression of the pandemic, although there is generally a 17-21 day lag between infection and deaths.

Scenario: Projection ✕ · Easing ✕ · Masks ✕

All deaths specific to COVID-19 patients.

Figure 1.1: Dashboard for COVID-19 metrics in the USA at https://covid19.healthdata.org. [1]

A notable example of the extensive use of data analytics and visualization has been research around the coronavirus pandemic in 2020. Research institutes, such as the Institute for Health Metrics and Evaluation, published interactive dashboards to inform the public about current threat levels of the pandemic. In Figure 1.1 two graphs are shown that track historical metrics for hospital resource use and daily deaths in the USA attributed to the coronavirus, as well as forecasts for the future development of those metrics depending on which public health measures are taken.

To increase the impact of publishing new insights, suitably presenting results is just as important as the aggregated data itself. To convince the public and sponsors of their research, scientists are motivated to visualize their results so that a conclusion can be drawn without much effort. In the case of the *Daily deaths* graph, the future projections make it pretty clear that introducing a universal mask mandate would be the best course of action. Color-coding the *Masks* measure in green and the *Easing* measure in red further suggest that continuing to ease mandates would be dangerous.

These dashboards highlight a nice property of the web. They are very easy to share because the only thing needed to view them is an URL and a web browser that is installed on virtually every device. However, most of these dashboards send network requests to a centralized database server to resolve analytical queries. The latency induced by fetching query results from the network makes data exploration feel slow.

Recent technological advancements allow running programs in a web browser almost as efficiently as native desktop programs, while still being safe from memory vulnerabilities. This technology is called WebAssembly, and we want to explore it to run a fully-fledged analytical database in the browser. That way, queries can be executed locally, resulting in almost instantaneous results.

While viewing and sharing dashboards is easy, creating dashboards is still a task mainly done by programmers. We want to explore building a data analytics tool that allows data scientists to create dashboards without needing prior programming experience. To accomplish this goal, the tool should offer both a graphical and textual mode to create dashboards.

Throughout this thesis, we aim to answer how efficiently our data analytics tool can process analytical queries and how interactive the dashboard creation process is. We build up our arguments in the following order:

In Chapter 2, we first introduce the data analytics and visualization domain. We look at how analytics desktop software is built, which capabilities web browsers provide for creating graphical user applications, and how databases are optimized for analytical query workloads. Regarding the capabilities of web browsers, we emphasize how WebAssembly enables high-performance computing.

In Chapter 3, we cover related work regarding data analytics applications in web browsers.

In Chapter 4, we cover how several compilers work as a prerequisite to understand how the interactive dashboard tool in Chapter 5 is built.

In Chapter 5, we are concerned with building an interactive dashboard tool, introducing the system architecture, and a concept for a dashboard programming language. We will present the grammar of the programming language and which features it offers.

In Chapter 6, we compare our system with similar data analytics tools in a qualitative and quantitative evaluation.

In Chapter 7, we present possible directions for our future work.

In Chapter 8, we reiterate our goals and how we approached them, and conclude how our evaluation relates to our goals.

2 Data Analytics and Visualization

To assess the landscape of data analytics and visualization software, we first look at the status quo of analytics desktop software. Subsequently, we examine which capabilities web browsers provide to build software, with a focus on graphical interfaces and high-performance code execution. Lastly, we investigate how databases are optimized for analytical query workloads.

2.1 Analytics Desktop Software

Software that involves handling large amounts of data, doing intensive computations or rendering complex graphics has been traditionally developed with native development toolkits for each operating system.

Using the low-level interfaces that operating systems provide has been the only way to make programs performant when analytics software first emerged. Computational power of hardware was more limited than today and the web platform did not yet provide interfaces to use hardware capabilities efficiently.

Targeting specific platform features however also greatly reduces the portability of software. Either a lot of complexity needs to be introduced to adapt and maintain code for each platform, or the software can only run on a limited number of supported systems. For that reason, many software publishers offered their products for Microsoft Windows only since they could not justify investing development efforts into making their programs compatible with other operating systems, e.g. Linux and macOS, with a smaller market share.

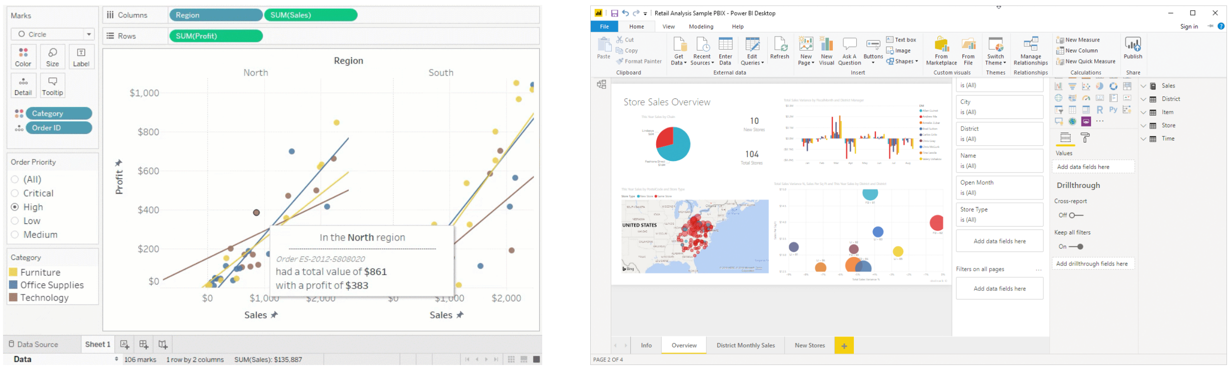


Figure 2.1: Tableau [2] and Power BI [3], desktop software for business intelligence and analytics.

The way desktop software is built and distributed has also been very inflexible. Because software has been distributed without the ability to upgrade it in smaller increments, lengthy cycles of development and testing were completed before a polished software product has been released. That way, new features and improvements have only been provided in release cycles of typically a year or longer.

Classical desktop products for business analytics by Tableau and Microsoft can be seen in Figure 2.1. These products provide tools to formulate database queries for analyzing data, visualize data in the form of interactive diagrams and charts, and to generate business reports. Data can be loaded from various sources, either from local spreadsheet and database files or external databases and data warehouses. Query processing is then preferably handled by a local database or by a powerful analytical database server when the data sets are too large to fit into the desktop client.

By now, these companies also offer web-based solutions as a companion for their business analytics products to take advantage of the flexibility and portability of the web platform. The online tools provide a similar set of functionality as their desktop counterparts, but rely on a server for most of query processing.

One example of this is Tableau Online [4], which additionally offers features to create and share dashboards via the web. Since fast in-memory processing in web browsers is still a very recent technological advancement, they do not take full advantage of it yet. Tableau features a collection of publicly shared dashboards [5], where one can observe that these dashboards are to some extent interactive. However, they rely on network requests to externally evaluate queries and render elements and can be slow to return results.

2.2 Graphical User Applications in Web Browsers

In this section, we highlight how web-based analytics software can improve the status quo of inflexible desktop software. Further, we go into detail how the feature set of desktop software can be fully replicated in web browsers, both in terms of graphical rendering capabilities and high-performance code execution.

2.2.1 Distribution Model

A big advantage of using web browsers as a platform for running applications is the distribution model. Because of the vast access to information and online services the world wide web provides, virtually every computer and smartphone has a web browser pre-installed, or one that has been installed by the user's own preferences.

In contrast to desktop and mobile applications, executing web applications requires no prior installation and no privileged access to the operating system. This is due to the fact that web browsers provide a basic set of functionality and interact with the operating system via interfaces that are designed to be safe even when running untrusted code.

The web platform greatly lowers the barrier for users to use an application and therefore provides access to a large amount of people who could potentially be interested in using our system for data analytics. This does not only apply to users who want to use our system to create dashboards, but also enables creators to easily share dashboards with their peers.

2.2.2 HTML, SVG and CSS

While the first web browsers were limited to displaying documents with formatted text and images, their capabilities have significantly grown since then. Nowadays, they are capable of arranging complex layouts not necessarily constrained to texts, and support a series of media formats for audio, video and graphics.

The general structure and contents of a web page are described by the *Hypertext Markup Language Format* (HTML) [6] in a tree-like structure. HTML also allows to directly embed other markup and programming languages to render more complex content.

Among those, the *Scalable Vector Graphics* (SVG) [7] format can be used to draw sophisticated vector-based shapes. Vector-based graphics have the advantage over pixel-based graphics that they can be scaled arbitrarily without losing resolution, if modeled appropriately. Further, their text-based format can be compressed easily and shapes based on mathematical equations can be expressed very concisely, making SVG a format that requires very little memory when stored. However, SVG can be more computationally expensive to render, since the shapes must be rasterized first before being painted on screen.

To style the appearance of nodes or their positioning on the document, nodes can either be annotated inline or obtain styles using *Cascading Style Sheets* (CSS) [8]. Styling nodes using CSS has the advantage that content and styling can be separated and that a single rule in the stylesheet can apply the same style to a multitude of nodes. To specify which nodes should be styled, so called CSS selectors are used.

Figure 2.2 shows how a graphic can be composed by using HTML with embedded SVG and CSS. The `h1` and `svg` nodes are centered by applying a style rule on the parent `body` node. How concise curves can be expressed with

SVG can be noticed by inspecting the `path` element. The line is plotted using a cubic Bézier curve by specifying merely four coordinates.

```html
<body>
    <h1>A Simple Line Plot using HTML, SVG and CSS</h1>
    <svg viewbox="-110 -61 221 121">
        <!-- Axes -->
        <line x1="0" y1="-60" x2="0" y2="60" />
        <line x1="-110" y1="0" x2="110" y2="0" />
        <!-- Arrow heads -->
        <polygon points="-1,-56 0,-61 1,-56" />
        <polygon points="106,-1 111,0 106,1" />
        <!-- Curve -->
        <path d="M-100 0 Q -50 100, 0 0 T 100 0" />
    </svg>
</body>
```

```css
body {
    display: flex;
    flex-flow: column;
    align-items: center;
    font-family: Latin Modern Roman;
}

svg line {
    stroke: black;
    stroke-width: 0.5px;
}

svg path {
    stroke: #0065bd;
    stroke-width: 1px;
    fill: transparent;
}
```

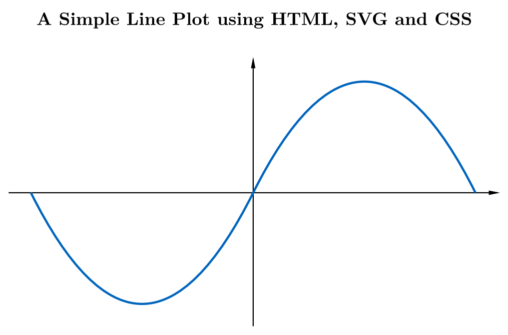


Figure 2.2: *On the top left*: HTML body for a document that contains a line plot.
On the top right: CSS rules for styling the document.
On the bottom: Rendered result of the HTML document with CSS rules applied.

Internally, the browser processes these nodes into a data structure, the *Document Object Model* (DOM) [9]. The DOM is first constructed with nodes statically present in the HTML document, but it is possible as well to create and manipulate nodes via a programmable interface.

These document nodes, and specifically SVG elements for their elegant properties of modeling geometric shapes, form the basis of the charts and diagrams in our dashboards.

2.2.3 JavaScript

JavaScript, or by its formal name ECMAScript [10], can be considered the lingua franca of the web. Up until very recently, it has been the only general purpose programming language that could be executed in all major web browsers. The scripting language was created to allow users to interact with elements on a web page and to dynamically inject content.

Still today, JavaScript has an exclusive position in web browsers. Various interfaces to communicate with the host system from within a web page can only be accessed from JavaScript. While these interface are exposed to JavaScript, the actual code communicating with the operating system is written in low-level system languages. The list of *application programmable interfaces* (APIs) that are used in our dashboards include:

Fetch API When accessing a dashboard, the application provides all logic necessarily to process and display data. This logic does not change while the user interacts with the dashboard. However, data sources may change frequently and are therefore loaded on-demand via the *Hypertext Transfer Protocol* (HTTP) to provide up-to-date information. The Fetch API is used to load data from the network into memory.

File API Local files may also be analyzed with the dashboard tool. This is useful both for large files that are stored locally so they do not need to be transferred over the network or files that are not exposed to any network service for compliance or privacy reasons. In that case, the File API is used to load data directly from the local file system.

Document API Interface elements and graphics need to be rendered dynamically depending on the dashboard configuration and input data. The Document API is used to create and manipulate the DOM nodes as described in Section 2.2.2.

Event Handler API Exploring data goes beyond simply displaying diagrams and charts. It should also be possible to interact with them, e.g. zoom in on specific ranges, change input parameters and show detailed metrics when hovering a diagram. To run actions on keyboard, mouse or touch input, event handlers are attached to the DOM via the Event Handler API.

WebAssembly API Queries should be evaluated as fast as possible to make interaction with data fluid. To minimize latencies, queries are executed locally in the browser. WebAssembly makes it possible to compile a database engine into machine code with only little overhead, achieving performance comparable to running native binaries. The WebAssembly API provides an interface to compile and instantiate such binary modules. We provide a full example how to interact with this API in Section 2.2.4.

2.2.4 WebAssembly

In our dashboards, we want to emphasize on quick execution of database queries with low latency. For that reason, we process queries in-memory, once the data has been loaded from the network.

One way to accomplish this would be to write a query engine in JavaScript or to compile an existing database to JavaScript. However, JavaScript is not a good target for code that deals with large amounts of memory operations, because it is a dynamic scripting language where access to memory is managed by a runtime.

In JavaScript, memory is managed by *garbage collection*, which means that memory allocated from objects is only periodically freed. This is especially expensive for code that does a lot of allocations because available memory would run out fast, and the garbage collector would need to be invoked often to clean up no longer used memory for new allocations. Fine-grained access to memory, which is the basis for many optimizations in analytical databases as described in Section 2.3, would also not be possible because we do not have control over memory layout and how it is accessed.

Additionally, numerical calculations that are used in arithmetic operations or to compute memory addresses can be slow in JavaScript. This is because JavaScript does not distinguish between integer and floating point numbers where the latter generally require more CPU cycles when used in arithmetic operations. To adhere to the language semantics, the JavaScript runtime is forced to execute floating point operations even when the programmer knows in advance that his algorithm only deals with integers.

When JavaScript still was the only language that could be run in web browsers, software engineers from Mozilla researched how a subset of JavaScript could be used to better emulate low-level code. They found that the language specification allows coercing floating point numbers to integers via bitwise operations. Specifically, `x = x | 0;` guarantees that `x` is an integer and `z = (x + y) | 0;` adheres to integer arithmetic if both `x` and `y` are integers. By the use of the typed arrays API, memory layout and access can be controlled on a byte level. These two techniques guarantee that this subset of JavaScript can be heavily optimized by mapping it closely to machine operations and has been formally specified as asm.js [11].

The concept of asm.js then influenced the creation of a new programming language with the goal to be portable, fast and safe: WebAssembly [12]. These goals are accomplished in the following way:

Portable. To ensure code can be compiled for a large number of targets, WebAssembly uses a virtual instruction set architecture. This instruction set defines a number of low-level operations, which are not supposed to run on hardware directly, but are designed to closely map to instructions of most common hardware.

The instruction set architecture expresses computation by a stack machine: Instructions operate on an implicit stack, where arguments can be placed onto and consumed from when needed. This mechanism provides a simple, well-defined calling convention that can be translated to the target platform's calling convention. Additionally, optimal register allocation can be chosen by the compiler for each hardware architecture individually.

This instruction set design ensures that the semantics of the language are well-defined while allowing each platform to take full advantage of their hardware capabilities. Due to the low-level representation, many high-level languages can use WebAssembly as a compilation target.

For our dashboard tools that means that we can run it on mobile and desktop devices, and different CPU architectures like x86-64 and ARM.

Fast. As a portable intermediate language, WebAssembly aims to be fast in multiple regards: Speed of runtime execution and speed of compilation.

Since the virtual instruction set was designed in a way that most operations can be directly be mapped to hardware instructions, runtime code achieves similar performance to high-level code that was directly compiled to the target architecture.

WebAssembly refers to two things: A textual representation and a binary-encoded format for modules, which can be converted from each other. The textual format aims to make module contents readable for humans, while the binary format is more compact and can be consumed faster by machines.

When the binary format was specified, special emphasis was taken on that it would be possible to compile WebAssembly to machine code in a single compilation pass. In a browser context this means that functions can be compiled incrementally while the module is streamed from the network, allowing code to be run directly after it has finished downloading. This greatly reduces the time to first interaction for users.

Safe. From a user's point of view, a WebAssembly module loaded from the internet contains untrusted code. Therefore, a security model needs to be employed that limits possible adverse effects of running untrusted code.

To prevent memory of a process running a WebAssembly module to be corrupted, a simple sandbox model is applied. By design, WebAssembly modules can only operate on continuous chunks of memory so that bounds checks can be performed by simply validating if the requested memory address is within the range of memory assigned to the module.

Additionally, formal validation rules are checked while compiling that guarantee that function calls can not leave the stack in a corrupted state. The call stack and a table containing functions that can be called dynamically are stored in memory inaccessible to WebAssembly, ensuring that no maliciously injected code can be invoked. Dependencies to functions that could communicate with the operating system need to be explicitly declared, further limiting the attack surface.

These measures mitigate most common attacks that can be carried out on corrupted memory, while still being able to distribute high-performing low-level code to users.

To understand how the virtual instruction set of WebAssembly works in practice, we define a set of functions in C and inspect the corresponding output when compiled to WebAssembly.

The functions defined in Figure 2.3 carry out simple operations: `add` sums up two integers, `store` saves an integer to a specified memory location and `has_ultimate_answer` loads an integer from a specified memory location and prints `Yes!` if that integer equals `42`. A function to print a null-terminated string is declared as `extern` and is expected to be externally linked to the program.

For the corresponding WebAssembly module we can notice that on the top level the module is organized in sections. The first section contains references to external objects and functions that must be provided when the module is instantiated. For one, a reference to a continuous piece of memory that our module can operate on and a reference to the print function are expected.

Next, a data section provides static data to be copied into memory when the module is instantiated. In our case, this section contains the string `Yes!` that is used in the `has_ultimate_answer` function.

The rest of the module is occupied by function definitions that are subsequently exported, so that they can be called externally after being instantiated. The function signatures contain information about parameters, local variables, and return values, so that it can be statically validated that the implicit stack is in a consistent state when the function body is entered and left. E.g. `$add` expects at least two integers `$a` and `$b` on the stack before the function body is entered and must leave one integer on the stack, which is the return value of the function.

```c
extern void print(char*);


int add(int a, int b) {
    return a + b;
}


void store(int* address, int value) {
    *address = value;
}


void has_ultimate_answer(int* address) {
    if (*address == 42) {
        print("Yes!");
    }
}
```

```wat
(module
    ;; Import linear memory
    (import "env" "memory" (memory $env.memory 1))
    ;; Import function to print a null-terminated string
    (import "env" "print" (func $print (param i32)))
    ;; Data segment starting at address '1024'
    (data (i32.const 1024) "Yes!\00")
    ;; Function definitions
    (func $add (param $a i32) (param $b i32) (result i32)
        ;; Push summands onto stack
        local.get $a
        local.get $b
        ;; Consume summands and push sum onto stack
        i32.add
    )
    (func $store (param $address i32) (param $value i32)
        ;; Push address and value onto stack
        local.get $address
        local.get $value
        ;; Consume arguments and write value to address
        i32.store
    )
    (func $has_ultimate_answer (param $address i32)
        block $if
            ;; Push address onto stack
            local.get $address
            ;; Consume argument, load value at address and
            ;; push value onto stack
            i32.load
            ;; Push constant '42' onto stack
            i32.const 42
            ;; Consume two values and push '1' onto stack if
            ;; operands are not equal, '0' otherwise
            i32.ne
            ;; Consume value and jump to end of 'if' block, if
            ;; value is non-zero
            br_if $if
            ;; Push constant '1024' onto stack
            i32.const 1024
            ;; Call print function
            call $print
        end
    )
    ;; Export functions
    (export "add" (func $add))
    (export "store" (func $store))
    (export "has_ultimate_answer" (func $has_ultimate_answer))
)
```

Figure 2.3: *On the left*: Exemplary C code with simple function definitions.
On the right: WebAssembly module with same semantics as the C code. The module has been generated by a C compiler, converted to the WebAssembly Text Format and annotated for clarity.

If we look at the instructions in the function body itself, we can see that parameters and local variables can be accessed via `local.get`, which subsequently push the value onto the implicit stack. Other instructions can then consume values from the stack to carry out operations, e.g. `i32.add` consuming two integers and pushing the sum back, `i32.store` taking an address and a value to write and `i32.ne` consuming two values and pushing back `1` if the values are not equal or `0` otherwise.

Control flow can be expressed by `block` and `loop` labels in WebAssembly. These can also be nested and conditionally exited or repeated. E.g. in our `$has_ultimate_answer` function we want to exit the `$if` block, if the value does not equal 42. This can be accomplished by the `br_if` instruction which jumps to the end of a block if the consumed value is non-zero.

To demonstrate the interaction between WebAssembly and a host language, we use JavaScript in Figure 2.4 to instantiate a WebAssembly module.

```javascript
// Create memory that can be shared between WebAssembly and JavaScript
const memory = new WebAssembly.Memory({
    initial: 1, // 1 page à 64KB
});

// Function that prints a null-terminated string at a given address
const print = (address) => {
    const end = Math.max(address, new Uint8Array(memory.buffer).indexOf(0, address));
    console.log(new TextDecoder().decode(memory.buffer.slice(address, end)));
};

// Download, compile and instantiate WebAssembly module
const { instance } = await WebAssembly.instantiateStreaming(fetch("./wasm.wasm"), {
    env: { memory, print },
});

// Access exported WebAssembly functions
const { add, store, has_ultimate_answer } = instance.exports;

// Call function in WebAssembly
const result = add(41, 1);

// Store result in WebAssembly memory
const address = 0x1234;
store(address, result);

// Check if the result at the memory address is the
// Ultimate Answer of Life, the Universe, and Everything
has_ultimate_answer(address); // Yes!
```

Figure 2.4: Example how to interact with the WebAssembly module in Figure 2.3 from JavaScript.

First, a continuous piece of memory is allocated that can be accessed both from JavaScript and a WebAssembly module. Such shared memory is essential in our dashboard tool to exchange more complex data structures between the query engine in C++ and the JavaScript-based visualization. We can e.g. serialize a query string into this memory so that it can be accessed by the query engine, which then computes result rows, serializes them using a commonly understood format, and returns the memory address back to JavaScript.

Thereafter, we define a print function that can display a string to the console. At the same time, this serves as an example how a more complex data structure, in this case a null-terminated UTF-8-encoded string, can be deserialized from a memory buffer in JavaScript.

In the next step, we use the `WebAssembly.instantiateStreaming` API to load the module from network, validate and compile it in a single pass and instantiate the module. That way, we do not have to save an intermediate buffer and can directly interact with the module once the download has finished. In this API function call, references to `memory` and `print` are provided in an import object to the module.

Once the module has been instantiated, we can use exported functions from WebAssembly as if they were regular JavaScript functions. That way, we can combine access to privileged system API from JavaScript with the predictable performance of WebAssembly.

Understanding WebAssembly at the lowest level is beneficial to know where the performance improvements compared to JavaScript come from. However, for our dashboard tool we do not directly write source code at this low level but use compilers to generate WebAssembly modules and JavaScript modules for serialization and deserialization of data structures. More detailed information about these compilers can be found in Chapter 4.

2.3 Databases for Analytical Query Workloads

Applications whose primary purpose is analyzing and visualizing data have different requirements for their database engine than a system for handling live customer orders would have. The database workloads for the former are commonly classified as Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP) for the latter. [13]

OLAP database queries are predominantly concerned with reading and associating large amounts of entries and aggregating them. In contrast, OLTP queries need to often lookup and insert data, both at arbitrary locations. In regard to database implementation, these requirements are conflicting in many cases and cannot be optimized both at the same time without making any compromises.

Since we mainly focus on dashboards, we have a database workload that heavily reads from multiple data sources and executes computationally expensive queries to aggregate the results. In the following sections, we examine which technological choices are taken in database design to optimize OLAP workloads.

2.3.1 In-Memory Databases

Optimizing a system means speeding up work or even avoiding it altogether if it can be omitted under certain conditions. As a first step, potential bottlenecks need to be identified by observing where most of the time is spent during execution. This information can then be used to find parts in the system, where optimizations would have the most impact and should be addressed first.

In the case of classical disk-based database systems, profiling query execution reveals that interaction with memory (I/O operations) can take up a significant share of execution time, especially for read-heavy workloads. Traditionally, databases stored all data on external storage (such as spinning hard drives and later solid-state drives) and used main memory only for the working memory required during query execution.

Considering the memory hierarchy in a classical computer architecture in Figure 2.5, one can notice that access latency increases exponentially, the further away from the CPU the data resides. Specifically, the difference between fetching uncached data from main memory and external storage amounts to roughly five orders of magnitude. This illustrates well why keeping data exclusively in main memory can lead to significant performance improvements.

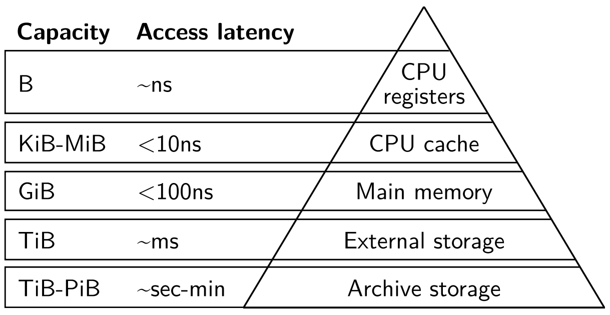


Figure 2.5: Memory hierarchy in a classical computer architecture.

Eliminating access to external storage from a database system is not only attractive for performance reasons. Buffer management, which means loading data segments from external storage into working memory and writing data back at a suitable time, requires careful implementation regarding correctness and efficiency. Removing this complexity can therefore reduce the effort to maintain the system implementation.

Historic reasons for using external storage in databases were mostly economical: main memory modules with enough capacity to entirely fit a typical data warehouse did simply not exist or were prohibitively expensive compared to external storage media. Only in the last one or two decades has it become feasible to realize a database system that would rely solely on main memory, as advancements in the manufacturing process of memory modules increased transistor density, making them both cheaper and larger in capacity.

The feasibility to design purely in-memory database systems heavily influenced the database landscape in the last decade. New commercial systems emerged, such as Hekaton [14] by Microsoft and SAP HANA [15], as well as research databases, such as HyPer [16] at TUM and MonetDB [17] at CWI.

The latest research database of CWI, DuckDB [18], also uses an in-memory design. Since it has been built specifically as an embeddable analytical database and matches well with our criteria, we use DuckDB as our primary query engine. The database is embedded into our C++ runtime for the dashboards and operates on a continuous piece of memory in WebAssembly.

2.3.2 Columnar Data Storage

To understand how the memory representation of data stored in a database can affect query performance, one needs to consider how the underlying hardware is utilized during query execution.

When data is fetched in an operating system on modern CPU architecture, various caches exist between the levels of the memory hierarchy. These caches exist to hide the high latencies of data fetching compared to available arithmetic processing resources. Starting from external storage, the operating system keeps a cache in main memory for data blocks of recently accessed files. Next, when data is loaded from main memory into CPU registers, the CPU loads adjacent memory regions into multiple levels of caches implemented in hardware. Namely, these are the L1-L3 caches, where L1 is the cache with lowest latency and smallest in capacity, and L3 the comparatively largest cache with highest latency.

Specifically for in-memory data processing we can exploit the fact that the CPU loads adjacent memory addresses, so called cache lines, ahead of time. If we maximize the density of relevant data that is loaded into the cache lines, we need to fetch data from lower levels in the memory hierarchy less often. That means that we are more rarely penalized with high latency costs and therefore need less time overall to load all required data.

To demonstrate how storing database entries column-wise instead of row-wise can utilize cache lines more efficiently, we first create a table that contains cities by the SQL schema specified in Figure 2.6. For simplicity, we assume that each attribute takes up the same amount of space in memory, that we have one cache level, and that a single cache line can hold exactly four attributes.

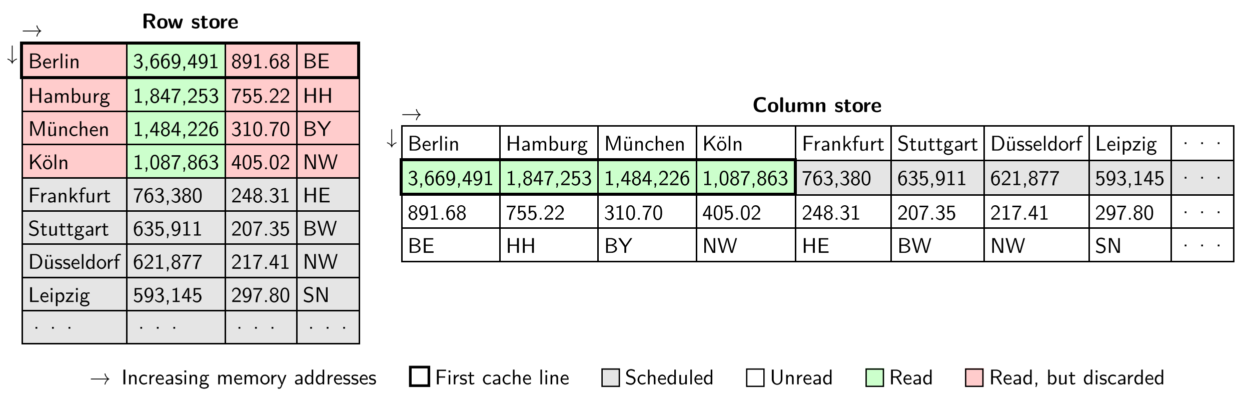
Next, we execute the SQL query in Figure 2.7 that reads the *population* attribute of each row in the *city* table and returns the sum of all *population* values. Effectively that means that the database engine is forced to consider every single row in the table, since each row could affect the result.

```sql
CREATE TABLE city (
    name VARCHAR(26),
    population INT,
    area FLOAT,
    subdivision_code CHAR(2)
);
```

Figure 2.6: Exemplary city SQL schema.

```sql
SELECT SUM(population) FROM city;
```

Figure 2.7: Exemplary aggregating SQL query.



Berlin	3,669,491	891.68	BE
Hamburg	1,847,253	755.22	HH
München	1,484,226	310.70	BY
Köln	1,087,863	405.02	NW
Frankfurt	763,380	248.31	HE
Stuttgart	635,911	207.35	BW
Düsseldorf	621,877	217.41	NW
Leipzig	593,145	297.80	SN
. . .	. . .	. . .	. . .

Berlin	Hamburg	München	Köln	Frankfurt	Stuttgart	Düsseldorf	Leipzig	. . .
3,669,491	1,847,253	1,484,226	1,087,863	763,380	635,911	621,877	593,145	. . .
891.68	755.22	310.70	405.02	248.31	207.35	217.41	297.80	. . .
BE	HH	BY	NW	HE	BW	NW	SN	. . .

Figure 2.8: Simplified memory layout and memory accesses during query execution for row- and column-oriented databases. Both databases contain the same entries for the schema shown in Figure 2.6 and execute the query stated in Figure 2.7.

In Figure 2.8, memory layout and memory accesses during query execution are shown for two different database systems. The one on the left stores a complete record for each row subsequently in memory, while the right one stores all attributes of one column subsequently before storing the next column.

For the column store, we can observe that after fetching the first attribute, all other attributes in our cache line are also relevant to our query result and will eventually be loaded into the arithmetic unit of the CPU to form the total sum. As a result, for each value we need to fetch from main memory, we can load the three subsequent values with a vastly lower latency from the CPU cache. As we can recall the memory hierarchy in Figure 2.5, these subsequent loads are roughly ten times faster.

The row store however does not make good use of the cache line. In fact, it is only able to utilize one single attribute within the cache line for the query result. Therefore, in this example, we do not benefit at all from the cache the CPU provides and each attribute needs to be loaded from main memory at full cost.

Due to good utilization of cache lines, column-oriented databases perform especially well when a lot of subsequent values are read, e.g. when filtering or aggregating a whole column. This is a common use case in data analytics and in queries that an interactive dashboard produces. DuckDB uses a column-oriented data store which made it a favorable candidate in our choice over other embeddable in-memory databases such as SQLite [19].

2.3.3 Vectorized Query Execution

When a database accepts a SQL query, the query is first transformed into a relational algebra tree internally. Relational algebra is a construct that allows to reason about database queries in set theory. Reasoning on a set-theoretical level about queries allows us to perform logical optimizations without changing the result set. This is very useful because we can apply these logical optimizations to transform a relational algebra tree into a simpler but semantically equivalent tree that can be executed more efficiently.

After the database has produced an optimized relational algebra tree, the tree needs to be transformed into something that can be executed by a machine. Several strategies exist, but we will focus on operator-centric execution only. Operator-centric means that the relational algebra tree does not only exist conceptually, but is also instantiated physically in code and is traversed during query execution to produce the result set.

To explain this method of computation, we guide through an example. For that, we reuse the *city* schema from Figure 2.6 and extend it by a *subdivision* table in Figure 2.9. Next, we want to fetch a list from the database that contains all cities that have a population of at least 10,000. Additionally to the names of the cities, we want the name of the subdivision they are located in to be displayed. The SQL query that formulates this request can be found in Figure 2.10.

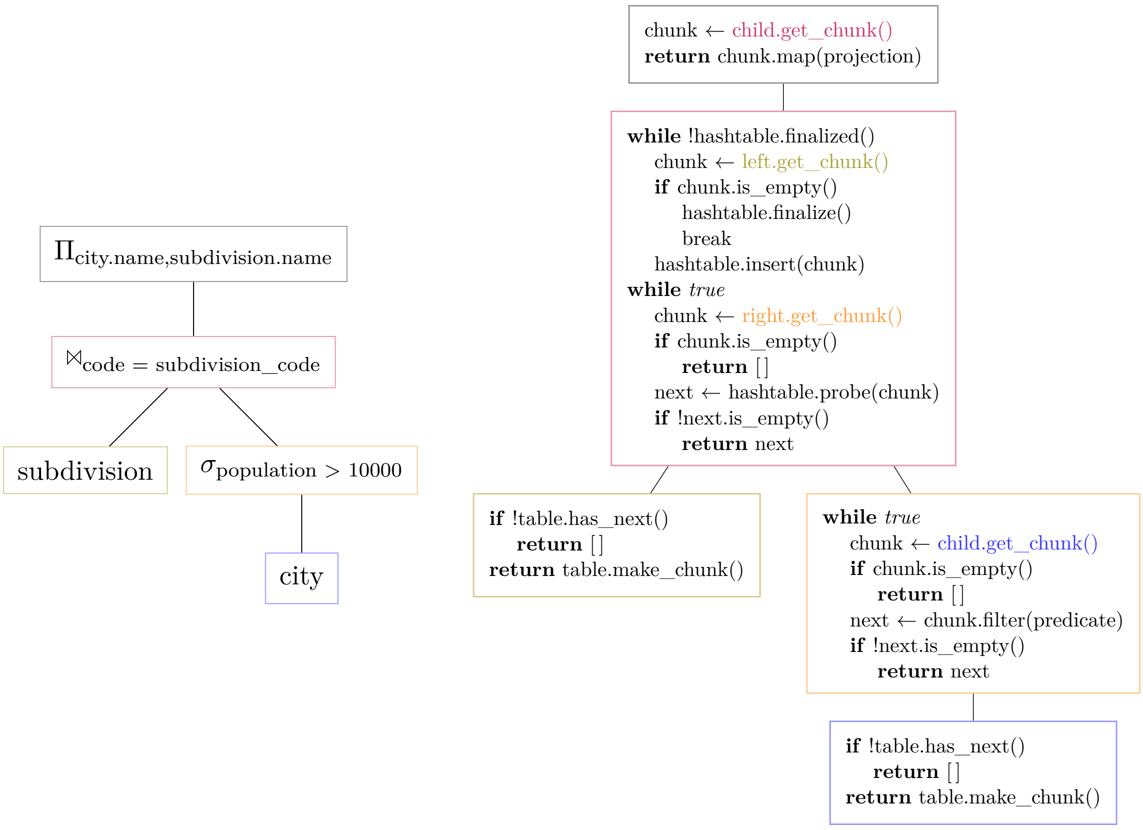
After parsing the query and transforming it into a relational algebra tree, the database optimizes the tree into one that most likely looks like in Figure 2.11 on the left. Concretely, the selection is pushed down, so that filtering out all cities that do not have at least 10,000 inhabitants happens before we try to match them with the subdivisions. Also, the sides of the join are swapped, because the database estimates that the number of subdivisions is smaller than the number of cities that match our criterion. That way, the data structure utilized to find matches is filled with less entries and therefore consumes less memory.

```sql
CREATE TABLE subdivision (
    code CHAR(2),
    name VARCHAR(22)
);
```

Figure 2.9: Exemplary subdivision SQL schema.

```sql
SELECT city.name, subdivision.name
FROM city
JOIN subdivision
ON subdivision_code = code
WHERE population > 10000;
```

Figure 2.10: Exemplary join SQL query.


$\Pi_{\text{city.name,subdivision.name}}$
$\bowtie_{\text{code = subdivision\_code}}$
subdivision
$\sigma_{\text{population} > 10000}$
city

```
chunk ← child.get_chunk()
return chunk.map(projection)
```

```
while !hashtable.finalized()
    chunk ← left.get_chunk()
    if chunk.is_empty()
        hashtable.finalize()
        break
    hashtable.insert(chunk)
while true
    chunk ← right.get_chunk()
    if chunk.is_empty()
        return []
    next ← hashtable.probe(chunk)
    if !next.is_empty()
        return next
```

```
if !table.has_next()
    return []
return table.make_chunk()
```

```
while true
    chunk ← child.get_chunk()
    if chunk.is_empty()
        return []
    next ← chunk.filter(predicate)
    if !next.is_empty()
        return next
```

```
if !table.has_next()
    return []
return table.make_chunk()
```

Figure 2.11: *On the left*: Optimized relational algebra tree for query shown in Figure 2.10. *On the right*: Pseudocode for vectorized query execution. The colored boxes highlight where the implementation corresponding to the relational algebra operator can be found. Colored text indicates which child operator is called.

The most straightforward implementation for traversing the relational algebra tree would start at the root of the tree and repeatedly request the next result tuple from the operator until no more tuples can be produced. Each operator would then recursively request tuples from its children, perform the computation corresponding to the semantics of the operator, and pass the result to the parent. At the leaf nodes of the tree, the actual tables are scanned for entries.

While this model is conceptually very elegant and easy to implement, the convenience comes at great cost. For each tuple that can be contained in the result, we potentially need to call every single operator in the algebra tree once. If we construct queries that return a large amount of rows, the number of function calls alone perceptibly affects performance. Additionally, these frequent context switches between operators lead to very inefficient utilization of cache lines, because we pollute the cache with memory accesses from other operators before we get to load the next entry from memory.

To correct the shortcomings of this technique, while keeping the fundamental idea, we seek to amortize the function calls into other operators by requesting and returning tuples in bulk. This method is also referred to as vectorized execution. In Figure 2.11 on the right we give an outline how an implementation of the relational operators present in our example might look like. We can observe that scanning the relation, filtering, and inserting and probing the hash table all happen in bulk. That way, the amount of function calls is greatly reduced and we can benefit from low latencies when accessing cached values during bulk operations on successive regions of memory.

The vectorized execution model was pioneered and popularized by the MonetDB/X100 [20] project. Due to excellent properties of vectorized execution for analytical queries, this design has also been incorporated into DuckDB. In summary, our requirements to an analytical database overlap very well with the project goals of DuckDB and make it an ideal candidate for our query engine.

3 Related Work

Using web browsers as a platform to analyze data is not a novel idea since the ease of distribution makes it a very attractive target.

When asm.js, a performance-centric subset of JavaScript, and Emscripten [21], a LLVM-to-JavaScript compiler, were released, researchers thought of which use cases would be interesting to showcase this new technology. Among the first demos was SQLite, an embeddable database written in C, running in the browser [22]. In a similar spirit, a prototype for a query engine was written that would compile SQL queries to asm.js and execute them in-browser. [23] However, these demos were more focused on the technical aspect of running SQL in a browser instead of being part of a larger data analytics tool.

Using HTML, CSS and SVG for data visualization has also been well explored. D3 [24] is a tool that can be used to create very sophisticated visualizations through a custom JavaScript API. The API provides imperative operations to create visualizations, can be used to animate transitions and makes user interaction possible by running actions when an event is triggered. This imperative model is very powerful, but pushes a lot of responsibility to the user to synchronize state and to determine when updates need to be executed.

Vega-Lite [25] is a similar tool to D3, but offers a high-level grammar to express layout, data transformations and interactions. Using a declarative grammar allows Vega-Lite to analyze data flow and visualization in the same context, so that actions that need to be run on user interaction can be inferred by the compiler. That way, a small set of granular updates can be applied automatically to visualizations.

While the grammar of Vega-Lite for visualizations and interactions comes close to the expressiveness we imagine for our dashboard tool, it is still very programmer-centric. It accepts specifications in a JSON-format and is initialized via JavaScript.

The goal for our data analytics tool lies more in the intersection of SQL, Vega-Lite and Emscripten. The language to create dashboards should be closer to what is used in the data science domain, as expressive as Vega-Lite and benefit from low-level optimizations by Emscripten and WebAssembly.

4 Compiler Toolchain and Development Tools

In this chapter we want to give some background about the tools that we use to build the interactive dashboard system as described in Chapter 5. Particularly, we want to go into detail in Section 4.3 how a parser for a programming language as described in Section 5.2 is constructed.

4.1 C++/Emscripten

To make parsing of input programs and query processing fast and responsive, we want our code to run in WebAssembly. Instead of writing this low-level code directly, we use a programming language that provides high-level features like classes and a collection of commonly used data structures.

We choose C++ for this task because it allows precise memory management and has a vast ecosystem of tooling and libraries that are suitable for database development. Among those, we use a parser generator for C++ (we go into more detail about this in Section 4.3) and embed the DuckDB query engine into our project.

Emscripten is a tool that can compile a C++ project to WebAssembly and additionally generates glue code that is needed to interface between JavaScript and WebAssembly. It uses a C++ compiler to generate LLVM [26], a low-level intermediate representation of code. The LLVM code is then translated to WebAssembly and a small JavaScript runtime.

Since WebAssembly has no direct access to platform APIs, Emscripten provides an implementation for system calls in JavaScript that are imported in WebAssembly. It does for example implement an in-memory file system in JavaScript to support system calls like `open`, `read` and `write`, and uses the JavaScript `WebWorker` and `SharedArrayBuffer` APIs to emulate multi-threading via POSIX threads.

Using Emscripten allows us to treat the dashboard tool mostly like a regular C++ project that can be run on native architectures, while also being able to target web browsers.

4.2 Protocol Buffers

In Section 2.2.4 we noticed that the only way to exchange data between WebAssembly and JavaScript are function calls or accessing a shared piece of memory. Function arguments and return values in WebAssembly can only contain primitive values, precisely integers and floating point numbers. To exchange more complex data structures, we need to read and write into shared memory and agree on a format for serialization, because e.g. objects in JavaScript and C++ do not share the same memory layout.

One way of specifying a serialization format are Protocol Buffers [27] by Google. Protocol Buffers are a binary message format which aims to be compact and to serialize and deserialize efficiently.

The Protocol Buffers specification language allows defining message types which are used to write key-value pairs into memory that can be reconstructed back to a message structure. From this specification, code can be generated for various programming languages that provide methods to pack and unpack data from a buffer. Serialization and deserialization code is not particularly complex, but still requires implementation effort. Auto-generating this code therefore also removes a potential source of programmer errors.

Another benefit of specifying a serialization format is that we can develop components of a system against a contract instead of relying on implementation details. Since the exact structure of return values is known, it would be possible to swap out the query engine for a more efficient one in the future, as long as it uses the same format to encode query results.

A basic example for a message specification using Protocol Buffers can be seen in Figure 4.1. The specification is a stub for how a query result might be encoded. A message contains fields which can either use one of the pre-defined scalar types or compose custom message types. The scalar types can be used to encode numeric and boolean values, strings, and raw bytes.

```
message QueryResult {
  repeated string column_names = 1;
  uint32 row_count = 2;
  // ...
}
```

Figure 4.1: Exemplary Protocol Buffers message specification.

Field values are optional by default, with the idea that clients accepting the message format are more resilient if they are always forced to consider the case that values could be missing. Additionally, field types can be marked `repeated`, meaning that an array of values for the same field can be provided.

The binary message format merely uses integers to identify fields. Therefore the specification needs to be known by both the encoding and decoding side to interpret a message correctly. In our example, `column_names` is tagged with key `1` and `row_count` with key `2`.

Tags in the binary message format additionally encode a wire type alongside the key. The wire type is stored in the three least significant bits of the field tag and is used by a decoder to determine how many more bytes belong to the field that is being read. There are three classes of wire types: *fixed-width* types, where a known number of bytes will follow, the *length-delimited* type, where an integer containing the byte size of the field precedes the field's data, and the *variable integer* type, where an integer can be encoded by a variable amount of bytes and each byte indicates with the most significant bit if more bytes are following.

```cpp
// Construct new message
auto query_result = QueryResult();

// Set column names
auto& column_names = *query_result
    .mutable_column_names()
;

for (auto& name : std::vector({
    "name", "population"
})) {
    column_names.Add(std::string(name));
}

// Set row count
query_result.set_row_count(42);

// Write serialized message to buffer
query_result.SerializeToString(&buffer);
```

Figure 4.2: *Serializing* example data using auto-generated *C++* library according to message specification in Figure 4.1.

```javascript
// Deserialize message from buffer
const message = QueryResult
    .deserializeBinary(buffer)
;

// Convert message to JavaScript object
const queryResult = message.toObject();

// ['name', 'population']
queryResult.columnNamesList;

// 42
queryResult.rowCount;
```

Figure 4.3: *Deserializing* example data using auto-generated *JavaScript* library according to message specification in Figure 4.1.

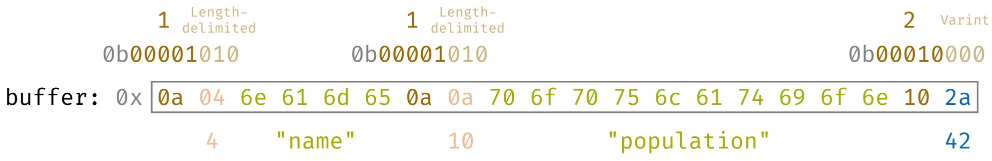


Figure 4.4: Example data in binary format according to message specification from Figure 4.1. Buffer contains message that has been written in Figure 4.2 and is read in Figure 4.3.

In Figure 4.2 and Figure 4.3 an example is given for how the serialization/deserialization libraries are used, that are automatically generated from the message specification in Figure 4.1 by the Protocol Buffers compiler. In particular, a query result message with the table names `name` and `population` and a row count of `42` should be serialized in C++, written to a buffer, and subsequently read and deserialized in JavaScript.

The binary representation of that message can be seen in Figure 4.4. We can notice that the first two values are length-delimited and belong to the `column_names` field, indicated by the key `1` encoded in the five most significant bits of the tag. After a value indicating the byte-length of the following data, a UTF-8 representation of the string is stored. The row count is stored in a single byte and associated by the key `2` and a variable integer wire type.

The actual message types in our dashboard tool are more complex than in this example but work according to the same principle. Specifically, we define a message to transfer a parser tree of the input program and a message to transfers query results from the database, where both specifications contain multiple nested message types.

4.3 Flex and Bison

For our data analytics tool, we want to model a custom programming language that has familiarities with SQL and can be used to declaratively build dashboards.

In programming languages, the first step of accepting and executing a program usually consists of transforming the input text into a structured representation, a so called parse tree. In most implementations, this process is separated into two phases: *Lexical analysis*, where the input text is divided and categorized into a sequence of tokens and *parsing*, where tokens are matched against a set of grammar rules to derive a parse tree.

Flex [28] and Bison [29] are tools to generate programs that perform lexical analysis and parsing, respectively. They were implemented as open source alternatives to Lex and Yacc and are compatible with the programming interface specified in their POSIX manuals [30][31].

Lexical analyzers (also referred to as *lexers*) generated by Flex use deterministic finite automata to produce a token stream and therefore are limited to accepting a regular language. In particular this means that for example nested quoted expressions can not be identified in this stage. To specify which set of tokens should be recognized, a collection of keywords and regular expressions can be provided to Flex. These tokens are the smallest units used in the more powerful parser rules.

```
"SELECT"        { return make_SELECT(); }
"*"             { return make_STAR(); }
","             { return make_COMMA(); }
"FROM"          { return make_FROM(); }
";"             { return make_SEMICOLON(); }
[a-z][a-z0-9_]* { return make_IDENTIFIER(getTokenText()); }
```

Figure 4.5: Exemplary lexer definition in Flex with regular expressions and C++.

Figure 4.5 shows an example how to instruct flex which kind of tokens should be produced by the lexer. We specify the keywords `SELECT` and `FROM`, the symbols `*`, `,` and `;` and a regular expression for identifiers that may start with a letter and contain letters or digits. Those tokens form the basis to parse a very small subset of SQL in our next example regarding parsers.

In its default configuration, Bison generates deterministic pushdown automatons to parse grammars, which use the LR(1) parsing technique. In this definition, *L* stands for *left-to-right*, which means that the parser starts at the leftmost token and successively consumes tokens to the right to find a matching rule, also called *derivation*. *R* refers to *rightmost derivation*, which means that the parser tries to find the most specific rules first, before constructing more general derivations. This behavior is also known as bottom-up parsing because the resulting parse tree is constructed from the leaf nodes up. Lastly, *1* stands for a *lookahead* of one, meaning that the parser considers one token further out to determine which rule to use next.

The parsing mechanism is implemented using *shift* and *reduce* operations, where shifting means placing the next token on the stack of the automaton and reducing describes the action of replacing a sequence of matching tokens on the stack to form a derivation. Due to the use of deterministic pushdown automatons, the languages accepted by these parsers are equivalent to the class of deterministic context-free languages.

To define which language the parser should accept, a grammar in *Backus-Naur form* (BNF) needs to be provided to Bison. Grammars in BNF consist of a collection of rules (also called *productions*), where on the left-hand side exactly one *nonterminal* describes a sequence of nonterminals or *terminals* on the right-hand side, or a list of alternatives thereof. A terminal is a symbol that may not be further resolved by any rule, and a nonterminal therefore one where further rules may be applied.

In Figure 4.6, we can see a set of productions that define a simplified select statement. To clarify the terminology, e.g. the `columns` nonterminal describes two alternatives: Either it ends in with the terminal symbol `*` or can be replaced by the `column_list` nonterminal.

On the right-hand side of the productions, Bison provides the ability to execute C++ code when a rule is matched. This construct is used to build up internal parse tree structures from the bottom up, which can then be further processed. E.g. the parsed structure could be used directly by a program interpreter or it could be compiler into program code that is executed at a later point.

```
select_statement:
    "SELECT" columns "FROM" IDENTIFIER ";" { $$ = SelectStatement($2, $4); }
;

columns:
    "*"                                    { $$ = SelectStatement::Star; }
  | column_list                            { $$ = $1; }
;

column_list:
    IDENTIFIER                             { $$ = vector($1); }
  | column_list "," IDENTIFIER             { $1.push_back($3); $$ = $1; }
;
```

Figure 4.6: Exemplary parser definition in Bison with BNF rules and C++.

A concrete example of which rules are matched on an input text can be seen in Figure 4.7. In this example, the input text has already been processed by the lexer and is provided as a list of tokens to the parser. The rules are matched from left to right, bottom up. Therefore, subtrees with the root `column_list`, `column_list`, `columns`, and `select_statement` are built in that order. If the parser has able to construct a complete parse tree when no more tokens are outstanding to be processed, the input program is accepted as valid.

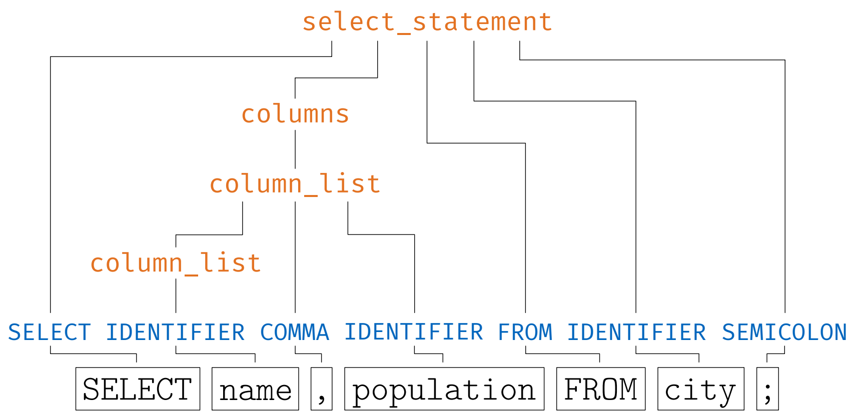

select_statement
columns
column_list
column_list
SELECT IDENTIFIER COMMA IDENTIFIER FROM IDENTIFIER SEMICOLON
SELECT name , population FROM city ;

Figure 4.7: *Upper part:* Derivation of a select statement by the grammar rules defined in Figure 4.6. *Lower part:* Exemplary input text and resulting token list according to rules in Figure 4.5.

In addition to BNF, the *extended* notation *E*BNF has been established.

It extends BNF in a way that productions may specify loops and optional elements on the right-hand side. In BNF, these constructs can otherwise only be modeled unwieldy by using multiple indirections. Nevertheless, both forms are equivalently expressive and can be converted into each other.

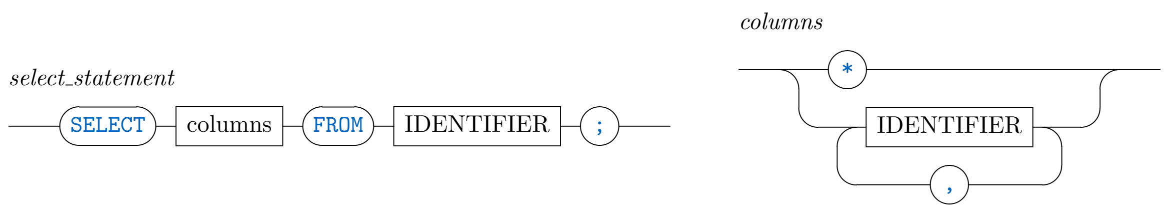


Figure 4.8: Grammar in EBNF which accepts the same language as the one in Figure 4.6, visualized as railroad diagram.

Since grammars in EBNF can be converted into railroad diagrams, which can be interpreted quite intuitively, we use these types of diagrams in Section 5.2 instead of BNF to express the same grammar.

An example of the visual representation of EBNF as a railroad diagram is shown in Figure 4.8, which has been constructed on the basis of the grammar in Figure 4.6. Italic titles indicate the definition of a nonterminal, round boxes contain terminals where the token inside is read verbatim, and squared boxes contain further nonterminals or terminal tokens that are resolved by a regular expression.

5 Interactive Dashboards

Data sets can be explored by observing how the output changes depending on which queries are formulated or how input parameters are chosen. Graphical user interfaces are a great way to make the process of understanding data more intuitive because users can visually interact with regions that are interesting to them. Our goal is to create an analytical tool that can be used by graphical interaction and is distributed via the web so that it is accessible to a large user base.

In this chapter, we present how we built such a data analytics tool that makes both creating and using dashboards interactive. We first give an overview of the system architecture, then how the dashboard programming language is designed, and lastly which features it offers.

5.1 Architectural Overview

The data analytics tool is divided into multiple subsystems that have clearly defined responsibilities. This makes the overall implementation simpler, because complexity is isolated in smaller modules.

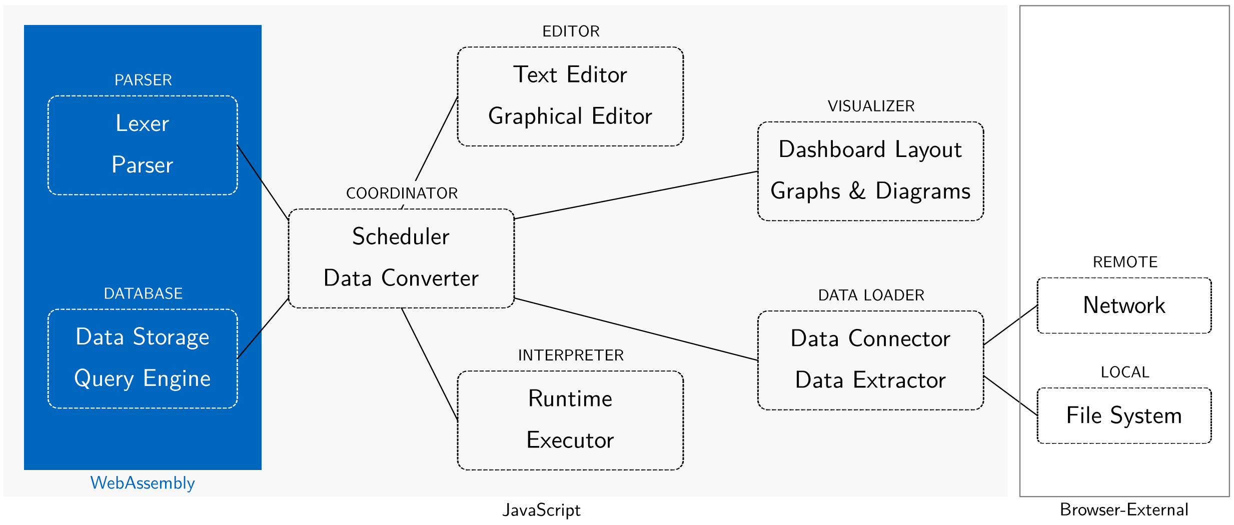


Figure 5.1: Architectural overview of the data analytics tool. The dashed boxes indicate subsystems. Subsystems are connected by a line when they directly interact with each other. The blue area contains submodules that run in WebAssembly and are optimized for performance.

Subsystems communicate via interfaces so that they are coupled to a defined set of functionality instead of relying on implementation details. That way, parts of the system can be refactored and improved without needing to change all components that dependent on it.

Figure 5.1 shows which modules are part of the system. The shaded gray area indicates where memory and program execution is controlled by the browser. Since APIs to communicate with the host system can only be accessed from JavaScript (see Section 2.2.3), submodules that are mainly concerned with user input, graphical output, and loading data from external sources are implemented in JavaScript.

Modules that are self-contained in a sense that they only operate on data input and return a structured result, can be fully implemented as WebAssembly modules. These are located in the blue area and benefit heavily from fast code execution. The reason why only a relatively limited amount of the system runs in WebAssembly is discussed among the limitations in Section 6.3 and inspires future work in Chapter 7.

The subsystems in Figure 5.1 are responsible for the following functionality:

Editor. The editor is the main entry point for user interaction when creating a dashboard. It offers two input modes that operate on the same data structure: A program can be written in a code editor, or can be manipulated via a graphical representation of the program structure. The editor is responsible for capturing text input by keyboard and mouse/touch input by the JavaScript Event Handler API.

Visualizer. Users who explore a dashboard mainly interact with the visualizer. It has two responsibilities: It creates widgets and configures the dashboard layout using the JavaScript DOM API and CSS, and draws charts and diagrams into these widgets using SVG when query results arrive. The JavaScript Event Handler API is used to notify the system when interaction with elements requires evaluating new queries.

Coordinator. The coordinator is a central module that instantiates all other modules and orchestrates program flow. Each subsystem has a different interface for how its functionality is invoked and which data format is expected. The coordinator allows modules to communicate by converting data to a suitable format. Most of the data conversion is implemented using Protocol Buffers, as demonstrated in Section 4.2.

Data Loader. A dashboard program can choose to load data from various sources. The data loader is concerned with the specific protocols that need to be implemented for each external source. It uses the JavaScript File and Fetch APIs to load data from the local file system or the network.

Database. To evaluate analytical queries, the database is an integral part of our system. It is optimized to quickly process large amount of data and runs in WebAssembly, powered by DuckDB and Emscripten, as described in Section 4.1. Data that is gathered from various sources is be imported and stored in a relational format. The specifics of how the query engine achieves high performance is described out in Section 2.3.

Parser. Dashboard programs are encoded in plain text and need to be converted into a structured representation before they can be executed. The parser is responsible this transformation. It runs in WebAssembly and is implemented using Flex and Bison, as described in Section 4.3.

Interpreter. Interpreting a program conceptually means stepping through a program's instructions one by one and executing them. The interpreter uses the program representation produced by the parser and maps it to actions to be run by other modules. E.g., it notifies the database when to execute a query and instructs the visualizer when to draw query results. It also stores and resolves variables during program execution.

5.2 Dashboard Programming Language

For dashboard creators, the dashboard programming language will be the main interface to configure how a dashboard should behave when it is loaded by a user. In this section, we describe the concept, features and implementation of the language.

5.2.1 Language Concept and Goals

Common workflows in data analytics follow a so-called *ETL* process, which stands for *Extract*, *Transform*, *Load.* We want to adopt this process, but extend it by one step: *Visualize.* That way, data analytics and visualization are unified into one seamless process.

The dashboard programming language has the goal to express the *ETLV* process as elegantly as possible. It should provide constructs to *extract* data from various sources, *transform* it by a suitable schema, *load* it to a database and query it, and finally specify how results should be *visualized.*

The following goals should be satisfied by the language implementation:

Approachable. Anyone who wants to create a dashboard, should be able to do so with reasonable effort. At the same time, we do not want to neglect power users who are experts of their tools. For this reason we offer a hybrid mode for modification: Dashboard programs can be created and edited in a graphical and textual mode at the same time.

Interactive. Forgetting syntax or producing erroneous programs is a regular part of the development cycle. The process of creating and editing programs should be interactive in a way that the user interface provides building blocks, contextually suggests which options are available and reports errors at the location they happened.

Textual. Storing and versioning programs should be straightforward. The "single source of truth" of dashboard programs is therefore human-readable, concise text. This also ensures that programs are maintainable by other developers in the future. Having an explicit, well formed representation for a program is beneficial when considering that programs are read more often than they are written.

Declarative. Dashboard programs should state which results they want to obtain, instead of formulating instructions how to compute them. This allows for automatic optimizations of programs and takes the burden off users that worry about performance instead of focusing on the analytical problem.

Familiar. The programming language should be self-explanatory enough that it is possible to deduce without prior knowledge what a program does. It should be possible to guess how a programs generally needs to be structured by inspecting programs written by other people. That is why we model the language closely to SQL which is commonly used in the data analytics domain.

5.2.2 Grammar and Language Features

The dashboard programming language is modeled as a superset of SQL. That means that every SQL program is a valid dashboard program, but would not yet be useful on its own. Additional statements are needed to produce a complete dashboard that loads and visualizes data.

These additional statements are designed to be familiar with SQL and are formulated as english imperatives. E.g. in SQL a statement to query a table starts with the imperative `SELECT`, and we use the additional imperatives `DECLARE`, `LOAD`, `EXTRACT`, `QUERY` and `VISUALIZE` in our language to begin a statement.

In the following subsections, railroad diagrams are used to express the grammar rules of the language. These railroad diagrams were derived from rules in BNF, that we use to generate a parser automatically. The relation between these diagrams and how the parser is implemented is explained in Section 4.3.

5.2.2.1 Program

At the top level, a dashboard `program` is expressed by a sequence of `statement`s that are finished by a semicolon.

For a `statement`, a number of alternatives are available. Usually, a program declares statements in the following order: input parameters, statements to load data, statements to extract data by a schema, statements to query data and finally statements to visualize data. However, since our programming language is declarative, control flow can be inferred without requiring that statements must be provided in this strict order.

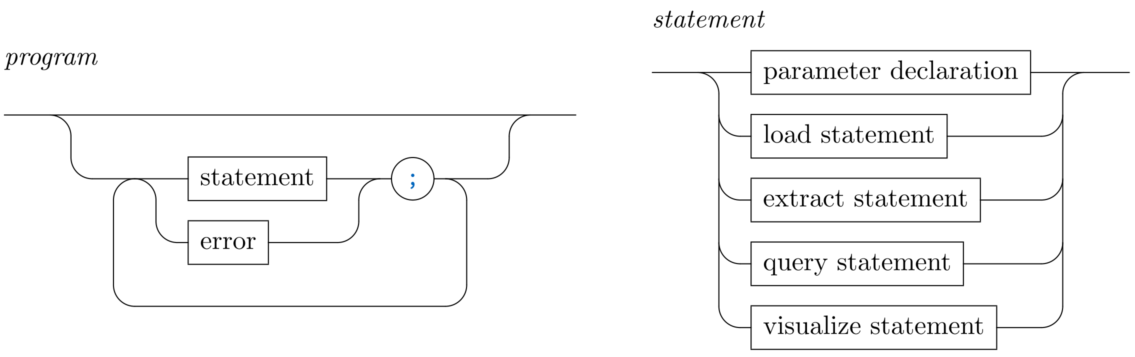


Figure 5.2: Program grammar.

The `program` production in Figure 5.2 also allows for `error`s to appear between semicolons. These are not formally specified, but rather indicate that the parser may ignore invalid input until the next semicolon and resume with parsing afterwards. This is useful in many cases where partially executing a program still produces reasonable output, instead of rejecting it altogether. The user is notified of the error, regardless.

5.2.2.2 Parameter Declaration

To actively encourage dashboard users to explore data by changing input parameters, `parameter declaration`s can be used.

A parameter must be named by an `identifier`, which is defined as either an `identifier literal`, `string literal` or `keyword`. An `identifier literal` is a token that may start with a character and subsequently contain characters, numbers and underscores, e.g `parameter_1`. A `string literal` is text that is quoted in single or double quotation marks. The `keyword` rule simply resolves to all possible keywords in the grammar, so that the parser can correctly choose as an `identifier` in this position, even though the token was classified as `keyword` by the lexer.

Optionally, a second `identifier` can be specified preceded by an `AS` token. In that case, the first identifier will be used as a label on the dashboard and the second one will act as an alias that can be referenced in source code.

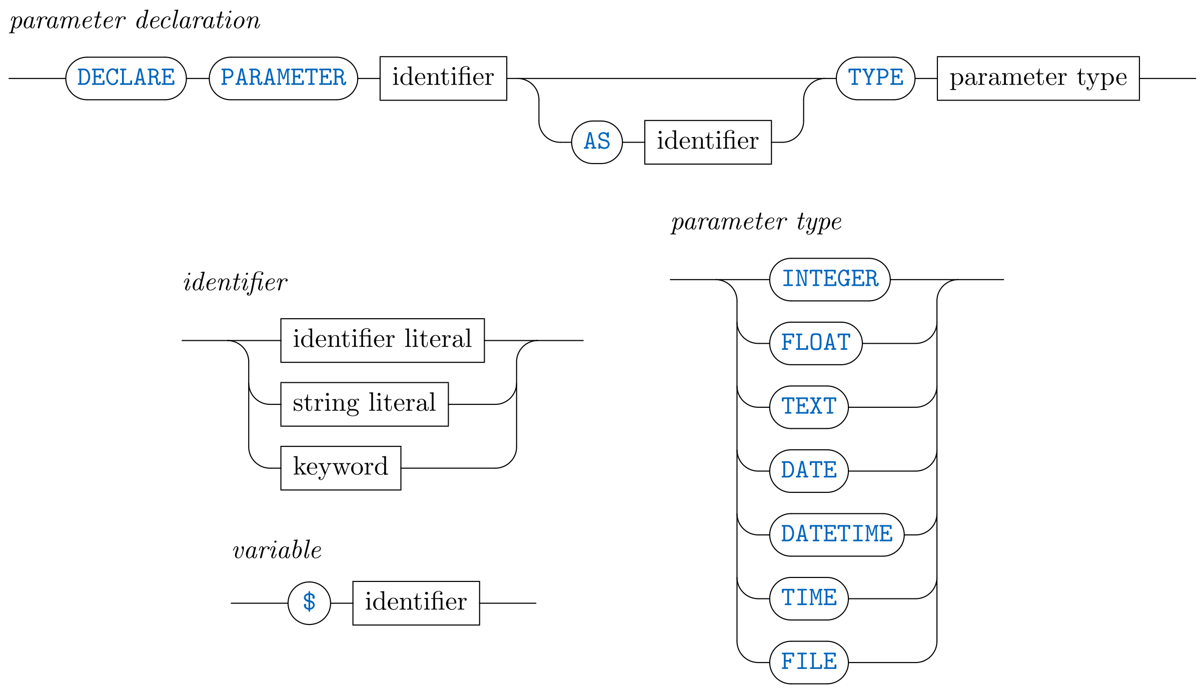


Figure 5.3: Parameter declaration grammar.

A parameter can then substitute all matching variables in the source code at runtime, where a `variable` is a `$` followed by an identifier. Variable substitution happens either by simple string replacement, or more complex operations like file loading, depending on the type of the parameter. Possible `parameter types` are listed in Figure 5.3.

For each declared parameter, a graphical input element is rendered on the top of the dashboard that can be manipulated by the user.

5.2.2.3 Load Statement

Before data can be analyzed and explored, a data set must be imported. Using `load statements`, data can be fetched from a number of sources. Currently, the dashboard tools allows to use a `text loader`, `file loader` or `http loader`, as reflected by the `load method` in Figure 5.4.

A `text loader` represents static data that is embedded as string literal into the program text.

When the `file loader` is used, data will be read from the local file system. A file must be selected by the user through a parameter field of type `FILE`, and the corresponding parameter identifier must be provided as a variable to the file loader.

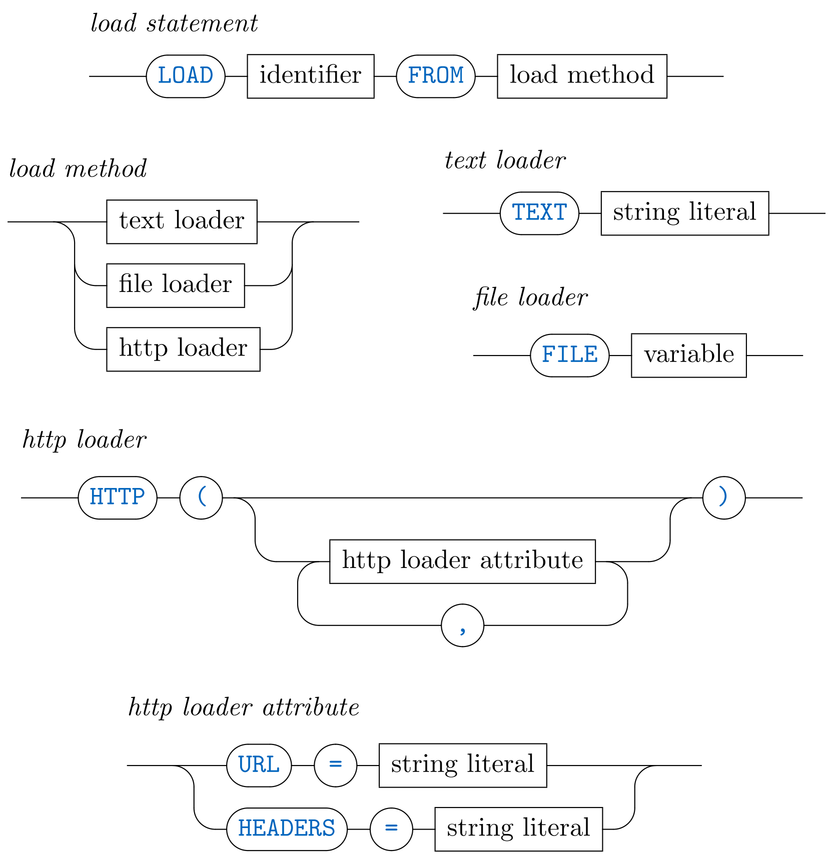


Figure 5.4: Load statement grammar.

To load data from network, the `http loader` can be used. It allows fetching content via HTTP, where options like the URL and request headers can be configured in `http loader attributes`.

An identifier needs to be provided to the load statement so that the raw data that has been fetched can be referenced for further processing.

5.2.2.4 Extract Statement

When data is fetched via a load statement, it initially is a raw buffer without further interpretation. The buffer needs to be decoded via an `extract statement` first before it can be loaded as structured data into the database.

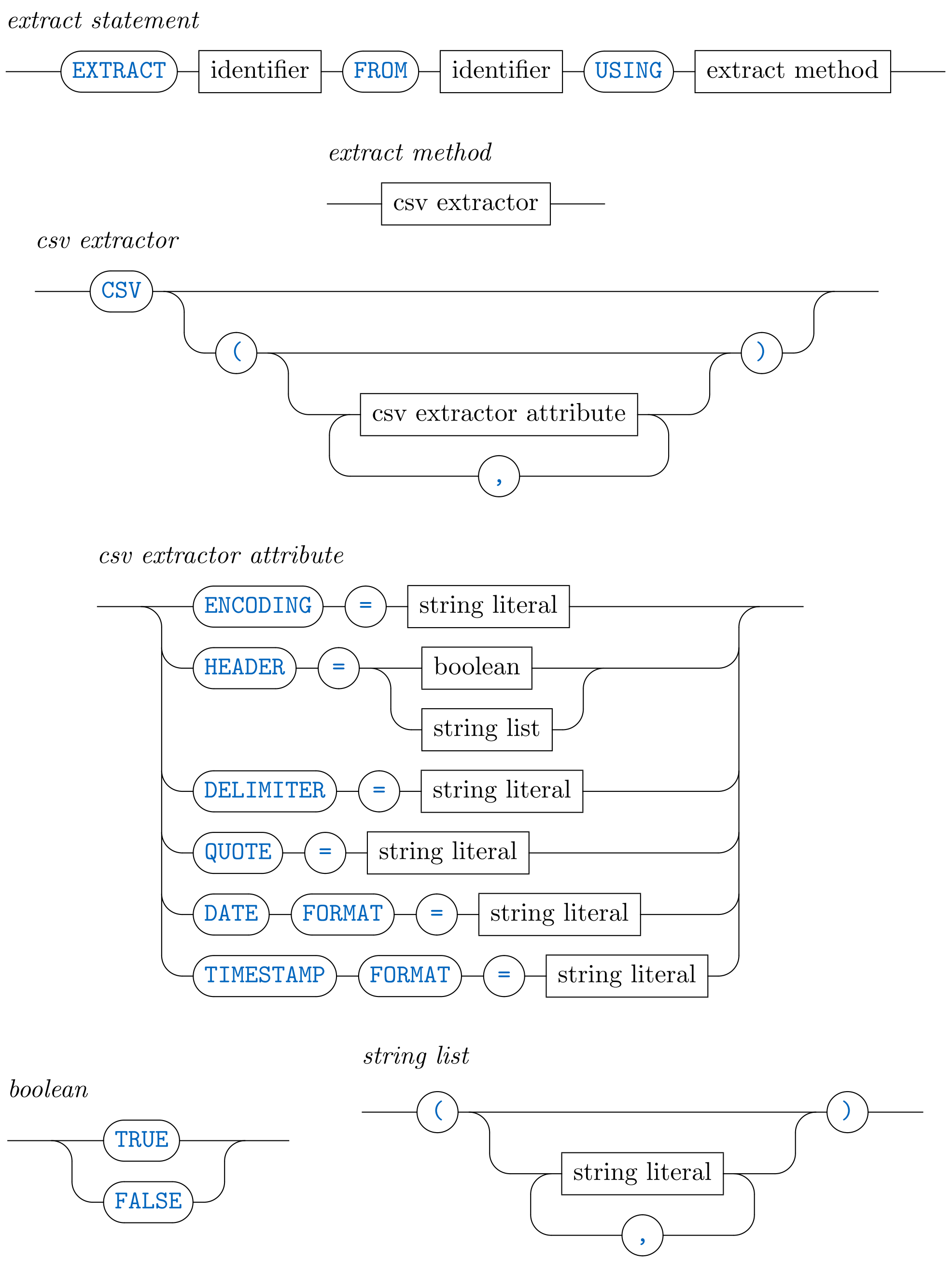


Figure 5.5: Extract statement grammar.

In this first version of the dashboard tool, the only available `extract method` is the `csv extractor`. However, it is easy to imagine how other methods to extract data can be added in the future, e.g. providing methods to extract data from JSON or allowing developers to provide custom modules that transform a buffer into structured data.

The `csv extractor` allows customizing how the data that is read is handled by specifying `csv extractor attributes`. The full list of available options can be found in Figure 5.5. They can be used to change the delimiter that is used to separate values, specify a list of columns names for the tabular data, or change the format of date and timestamp strings and quoted values.

An extract statement needs to specify two identifiers: The first one is used to reference the resulting structured data after it has been extracted, whereas the second identifier determines which load statement in our program provides the raw buffer.

After successful extraction a table is created in the database that contains the structured data.

5.2.2.5 Query Statement

Once data has been imported into the database, analytical `query statements` can be formulated to produce results that reveal more complex relationships within the dataset. This can be accomplished by e.g. joining multiple relations or aggregating values.

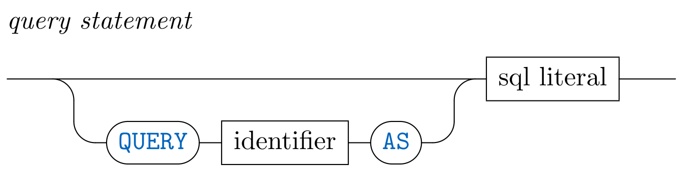


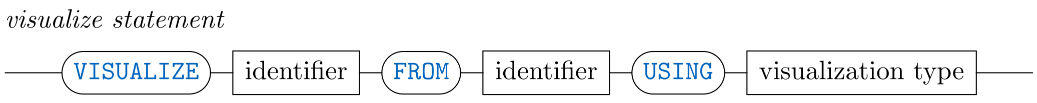
Figure 5.6: Query statement grammar.

The `query statements` are in fact only an instrument to provide an identifier for SQL queries, as can be interpreted from the grammar in Figure 5.6. To avoid needing to implement all rules of the SQL grammar, we tokenize entire SQL statements by a simple heuristic. A `sql literal` token is recognized by the lexer using a regular expression that searches for strings that begin with `SELECT` or `WITH` and end with a semicolon. That way, SQL queries and common table expressions can be embedded.

Using a full-fledged relational database engine and allowing SQL to query data provides great benefits. The query language is flexible and supports many features, builds on vast research of query optimization and allows developers to reuse their existing knowledge on SQL. If they are not well versed in SQL yet, existing standards and documentation can be used as resources.

5.2.2.6 Visualize Statement

After all query statements have been evaluated, the results need to be visualized in a suitable way. A `visualize statement` specifies in which way the data should be displayed.



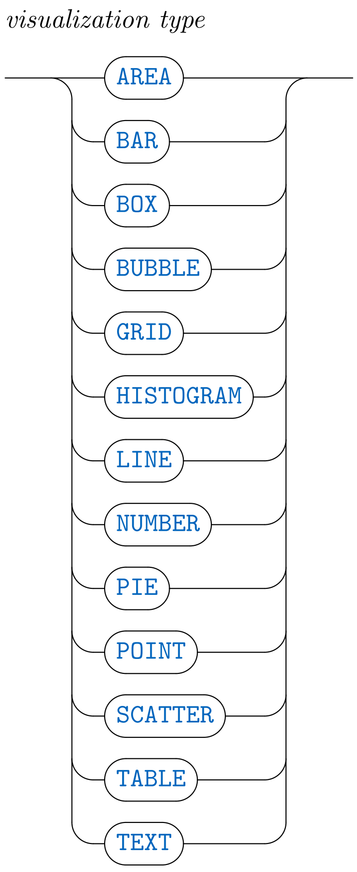


Figure 5.7: Visualize statement grammar.

The `visualization types` in Figure 5.7 lists all methods that are available to create a widget on the dashboard. Currently, the visualization mechanism simply defers to an existing JavaScript library with ready-made components for data visualization.

In particular Vega Lite was chosen, only that none of its data processing capabilities are used. Data is pre-processed in our in-memory query engine and Vega Lite is used purely for the last step of drawing graphical elements.

In future releases, more granular options for customizing visualization can be implemented, instead of relying on these default options. The dashboard language would then be extended to accept new attributes.

A visualize statement expects two identifiers: The first one is used to display a label on the dashboard widget, while the second identifier specifies which query result or relation should be visualized.

6 Evaluation

To evaluate our work, we take both a qualitative and a quantitative approach. The results of the evaluation are presented in this chapter.

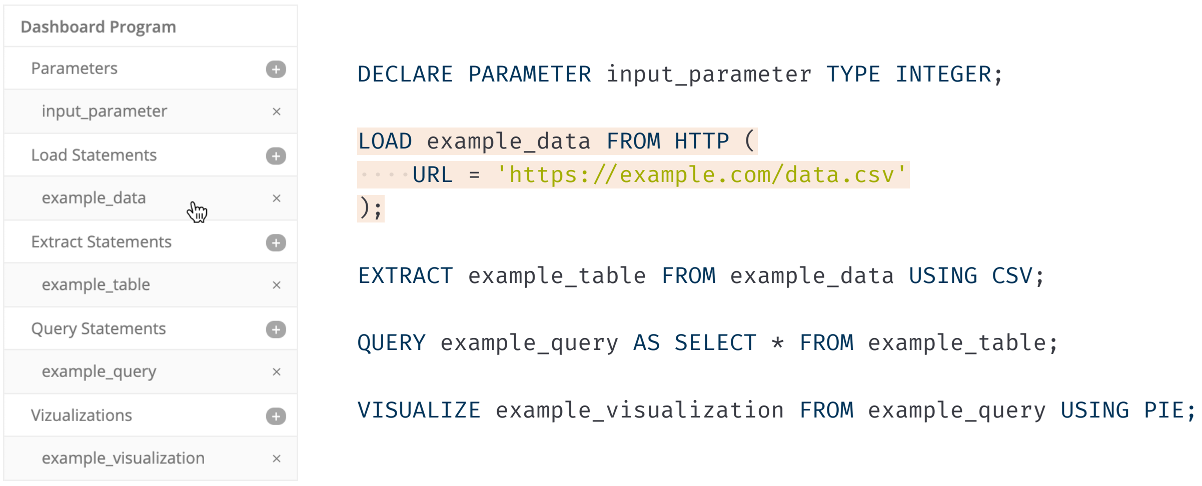
The qualitative evaluation assesses in how far our data analytics tool meets the goal of being interactive, and which functionality the dashboards provide. In regard to quantitative evaluation, we compare benchmarks of our implementation and another data analytics library that aims to solve a similar problem. That way, we can validate in how far our claims of efficiency are justified.

Lastly, we discuss the current limitations of our system.

6.1 Interaction with Dashboard Tool

In this section, we present fragments of the graphical user interface for our data analytics tool. The order in which we present these fragments corresponds to the order in which they are used in a typical workflow to create a dashboard.

Initially, a dashboard is a blank canvas. To add widgets for visualization and to create a data flow that can be provided to these widgets, the general structure of our programming language must be followed.



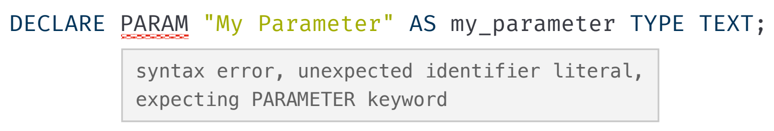
```
DECLARE PARAMETER input_parameter TYPE INTEGER;

LOAD example_data FROM HTTP (
    URL = 'https://example.com/data.csv'
);

EXTRACT example_table FROM example_data USING CSV;

QUERY example_query AS SELECT * FROM example_table;

VISUALIZE example_visualization FROM example_query USING PIE;
```

Figure 6.1: Interactive input mode to create a dashboard program. The graphical program representation on the *left* is synchronized with the textual representation on the *right*. When a statements is hovered in the graphical program structure, the corresponding section in the source code is highlighted. Various buttons are placed on the structure that allow removing statements, or adding statements via automatically inserted templates.

The user interface for dashboard creators provides tools to build a program, textually and graphically. The graphical outline of a dashboard program, as seen on the left in Figure 6.1, helps understand which program statements need to be provided. When a + or × sign is clicked on the outline, the program text is modified. In case of addition, a suitable template is inserted after the last statement of that group. When removing, the corresponding section is erased from the program text.

The information for which exact location in the text needs to be modified is transferred with the Protocol Buffer message for each statement. A statement stores the line and column number of where its first token starts, and where the last token of the statement ends.

Having access to location information for statements has additional practical applications in our program editing tool. The location information allows to associate the program outline and source text in a very direct visual way. Figure 6.1 demonstrates how hovering a statements results in highlighting the corresponding section in the source code. That way, it is easy to keep track of program statements, even when the source code grows in complexity over time.



```
DECLARE PARAM "My Parameter" AS my_parameter TYPE TEXT;
```

```
syntax error, unexpected identifier literal,
expecting PARAMETER keyword
```

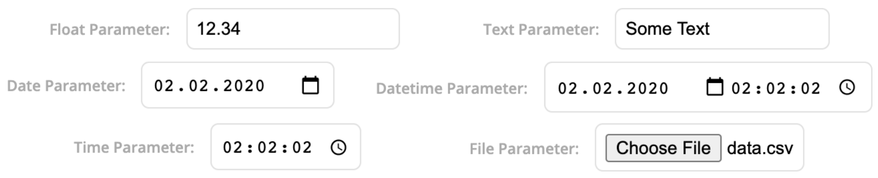
Figure 6.2: An inline error message appearing while editing program text. The `PARAMETER` keyword has been incorrectly abbreviated, leading to a syntax error. A contextual error message is provided, hinting at how the problem can be resolved.

When errors happen during development, contextualized diagnostic information is invaluable. Reading an error message directly where the error happened removes guess work for what produced the fault and hence speeds up the debugging process. Being able to associate an error visually with the source code is another good example for how the token locations are used.

A concrete instance of this feature can be seen in Figure 6.2, where misspelling a keyword leads to a syntax error. The contextualized error message provides help to fix the mistake.

The following examples demonstrate how a dashboard program affects the user-facing output. The user-facing canvas contains elements that visualize data and input controls to modify parameters.

A collection of these input elements can be seen in Figure 6.3. The appearance and behavior of the controls is automatically adjusted, depending on if the expected input type is numerical, textual, temporal or a file. The values that have been provided by a user can then affect which data should be sourced or if analytical queries should be reevaluated.



```
DECLARE PARAMETER "Float Parameter" AS float_parameter TYPE FLOAT;
DECLARE PARAMETER "Text Parameter" AS text_parameter TYPE TEXT;
DECLARE PARAMETER "Date Parameter" AS date_parameter TYPE DATE;
DECLARE PARAMETER "Datetime Parameter" AS datetime_parameter TYPE DATETIME;
DECLARE PARAMETER "Time Parameter" AS time_parameter TYPE TIME;
DECLARE PARAMETER "File Parameter" AS file_parameter TYPE FILE;
```

Figure 6.3: Dashboard input parameters. Declaring these specific parameters (*bottom*) in a dashboard program results in rendering the input controls (*top*) on a user-facing canvas. The first identifier in a parameter declaration appears as a label on the left of an input, while the second identifier can be referenced as a variable in a dashboard program.

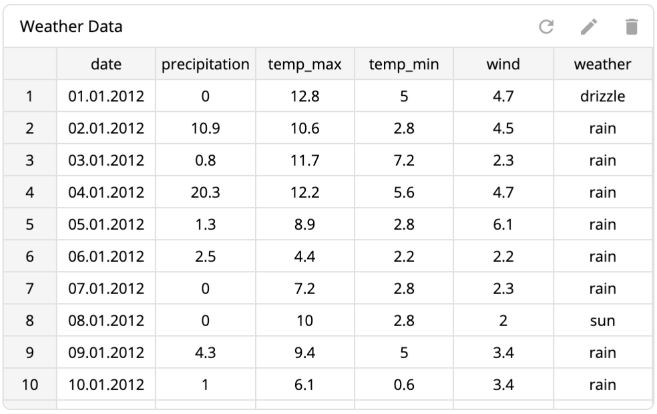
A full example for how all parts of the dashboard programming language work together is presented in Figure 6.4. The declarative nature of our programming language should already give a rough idea of what the program accomplishes, even when unfamiliar with the exact grammar and semantics.

We can start reading the program from the bottom to see what the stated goal is, and then move further up to understand how it is accomplished.

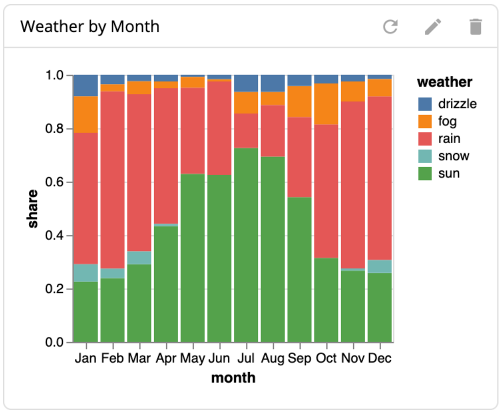
At the very bottom, two `VISUALIZE` statements are declared. Reading the title and visualization type, we can infer that the following widgets should to appear on the dashboard: One that provides an interface to scroll through raw tabular weather data, and another one that displays weather data in some aggregated form in a bar chart.

Next, we determine, where the `weather` relation comes from. Following the chain of `EXTRACT`, `LOAD` and `DECLARE` statements at the top of the program, we can understand that the weather relation is created by loading a local file in the CSV format.

In the next step, we resolve the more complex `weather_by_month` relation. It is defined in a `QUERY` statement and acts as an alias for a SQL statement. Again, we start reading the SQL statement from the bottom. It aggregates a relation that provides weather information along a month only instead of the full date. We count all occurrences of a weather condition that can be found for each month.


Weather Data

	date	precipitation	temp_max	temp_min	wind	weather
1	01.01.2012	0	12.8	5	4.7	drizzle
2	02.01.2012	10.9	10.6	2.8	4.5	rain
3	03.01.2012	0.8	11.7	7.2	2.3	rain
4	04.01.2012	20.3	12.2	5.6	4.7	rain
5	05.01.2012	1.3	8.9	2.8	6.1	rain
6	06.01.2012	2.5	4.4	2.2	2.2	rain
7	07.01.2012	0	7.2	2.8	2.3	rain
8	08.01.2012	0	10	2.8	2	sun
9	09.01.2012	4.3	9.4	5	3.4	rain
10	10.01.2012	1	6.1	0.6	3.4	rain



```
DECLARE PARAMETER weather_file TYPE FILE;

LOAD weather_csv FROM FILE $weather_file;

EXTRACT weather FROM weather_csv USING CSV;

QUERY weather_by_month AS WITH weather_monthly AS (
    SELECT
        MONTH(date) AS month,
        LEFT(MONTHNAME(date), 3) AS name,
        weather
    FROM weather
),
weather_measurements AS (
    SELECT
        month,
        COUNT(month) AS count
    FROM weather_monthly
    GROUP BY month
)

SELECT
    name AS month,
    weather,
    COUNT(month) * 1.0 / (
        SELECT count
        FROM weather_measurements m
        WHERE m.month = w.month
    ) AS share
FROM weather_monthly w
GROUP BY name, month, weather
ORDER BY w.month ASC, weather ASC;

VISUALIZE "Weather Data" FROM weather USING TABLE;

VISUALIZE "Weather by Month" FROM weather_by_month USING BAR;
```

Figure 6.4: Full example for a dashboard program. This dashboard is a recreation of the stacked bar chart example [32] of Vega Lite, using our dashboard programming language. Data is sourced from an example data set [33] of daily weather measurements in Seattle. Executing the program (*bottom*) with the weather data set results in the visualizations (*top*) to be drawn on the user-facing dashboard canvas.

To normalize the count, we need to divide the occurrences by the total amount of weather phenomena that have been observed, resulting in the `share` column.

Lastly, we investigate how the remaining relations `weather_monthly` and `weather_measurements` are formed. The `weather_monthly` relation represents a month both as a number (so that it can be sorted afterwards) and as the first three letters of the month name. Additionally, it keeps track of the weather condition. The `weather_measurements` relations counts all measurements for a month that can be found in the `weather_monthly` relation regardless of weather condition, so that we can use that value for normalization afterwards.

6.2 Benchmarks

In the last part of Section 6.1, we recreated a dashboard based on a Vega Lite example. This is a great opportunity to make a faithful comparison between our implementation and a competitor.

At the same time, this analytical problem is a good real-world example and demonstrates how data analytics tools would perform in a comparable scenario. It is moderately complex and requires multiple aggregations to compute the solution.

The data input and query output for the stacked bar chart visualization in Figure 6.4 are well defined. The declarative format of both our dashboard programming language and the Vega Lite specification grammar allows us to precisely formulate the analytical problem. We then measure how much time each implementation takes to produce the desired result.

```
{
    "$schema": "https://vega.github.io/schema/vega-lite/v4.json",
    "data": {
        "values": [ ... ]
    },
    "mark": {
        "type": "bar"
    },
    "encoding": {
        "x": { "title": "month", "timeUnit": "month", "field": "date", "type": "ordinal" },
        "y": { "title": "share", "aggregate": "count", "stack": "normalize" },
        "color": { "field": "weather" }
    }
}
```

Figure 6.5: Vega Lite specification that produces a fully equivalent result to the “Weather by Month” visualization in Figure 6.4 when the same input data is provided. This specification has been used when benchmarking Vega Lite in Figure 6.6.

Results of these measurements can be observed in Figure 6.6. The graphs measure how many milliseconds each system took to compute the result, when provided with weather data that contains as many weather measurements as indicated by the x-axis.

The same performance measurements have been plotted into two separate graphs with different axis ranges, so that effects for both small and large data sets can be observed.

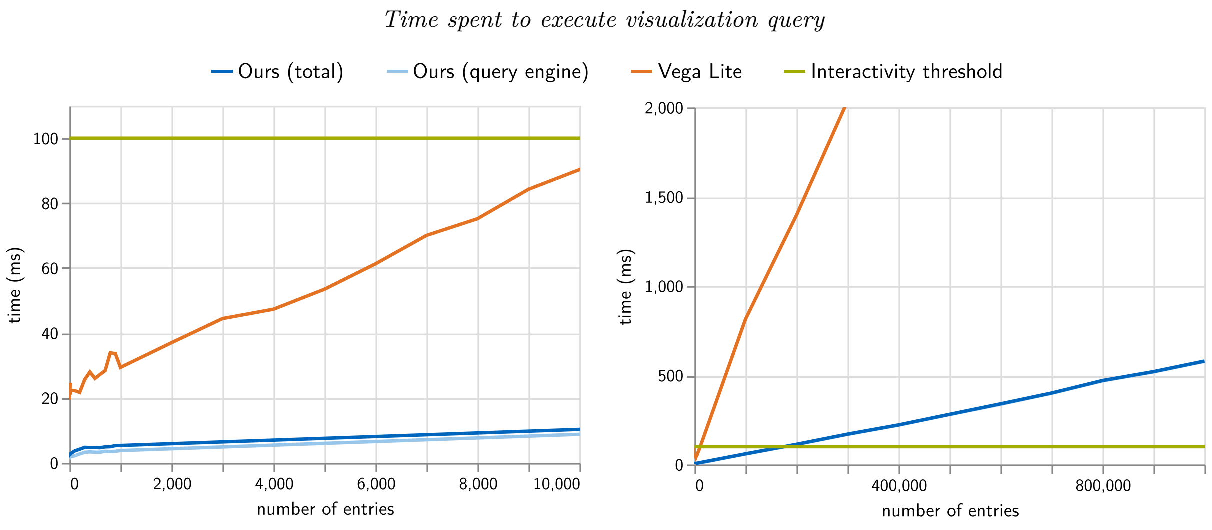


Figure 6.6: Benchmarks to return a query result for the bar chart visualization in Figure 6.4 before drawing to screen.

The *blue* line indicates the overall performance of our analytical query engine, including serializing and deserializing query results between WebAssembly and JavaScript. The *light blue* line indicates the performance of our query execution in WebAssembly before serializing the result. The *orange* line indicates query performance of Vega Lite. The *green* line indicates a common heuristic for latency requirements of systems that should be perceived as interactive.

The x-axis states the number of weather entries that have been loaded into the input buffer before query execution.

Benchmarks have been executed in the same environment, using the Chromium web browser (version 85.0.4183.102) on commodity hardware (Intel® Core™ i7-4770K CPU@3.5GHz).

The graph on the left lets us conclude that both Vega Lite and our system deliver low latency when processing up to 10,000 weather entries. Both systems return results faster than the threshold of 100ms, which is usually perceived as fluid interaction.

However, it is apparent that Vega Lite delivers way less consistent latencies than our WebAssembly-based system. This can be explained by just-in-time compilation and garbage collection of JavaScript engines, which are unpredictable from a programmer's perspective. Our query engine in WebAssembly is compiled ahead of time and therefore demonstrates very stable performance.

For our system, a discrepancy between pure query performance in WebAssembly and total time spent processing the request exists, as indicated by the gap between the blue and light blue lines. This is attributes to the costs of serializing and deserializing query results between JavaScript and WebAssembly via commonly shared memory. However, our query returns an upper bound of results when all monthly buckets of the aggregation have been filled. The cost of serialization is therefore quickly amortized as the number of input entries grows.

The graph on the right shows that our query engine also eventually exceeds the latency threshold. However, we are able to comfortably process up to roughly 175,000 weather entries while maintaining interactivity.

This is made possible by the many optimizations for analytical query workloads, as we have discussed extensively in Section 2.3. The workload of the bar chart query in Figure 6.4 scans sequentially over large amounts of data, which is precisely what our database engine is optimized for.

In this scenario, our systems performs roughly an order of magnitude better than our competitor.

6.3 Limitations

Regarding technical limitations, our system architecture in Section 5.1 has been designed around the limitations for WebAssembly modules in browsers.

No web browser implements an API yet that would allow to manipulate the DOM and communicate with the host system directly. Therefore, data needs to be transferred between JavaScript and WebAssembly, either when it should be loaded from external sources or when query results should be visualized. This induces costly steps of copying buffers, serialization and deserialization.

Using WebAssembly modules imposes further limits: Primitives for multi-threading are not implemented by all browsers yet or have been temporarily disabled due to security vulnerabilities in CPUs when using shared buffers. Therefore, parallel processing of queries is not possible at the moment.

The most pressing limitation with WebAssembly is that only a maximum of 4GB memory can be assigned to modules. This is due to the fact that the specification for 64-bit WebAssembly is not finalized yet. In current browser implementations, pointers in WebAssembly are limited to 32-bit and can therefore not address any memory above 4GB. This makes handling files or intermediate results that are close to or larger than 4GB very inconvenient, as these would need to be buffered through JavaScript memory.

As for the general implementation of our analytics tool, some features have only been prototyped. Not all visualization types are implemented or can only be used without further customization options.

The dashboard language interpreter is implemented naively: It materializes intermediate results for each `QUERY` statement. It doesn't implement any dependency graph that could determine that a result is not used by another statement, which could save the materialization step.

Error handling by the dashboard language interpreter is currently limited to syntactical errors, no static semantic analysis is performed before executing a program. Therefore, most errors can only be caught at runtime.

7 Future Work

Interesting directions for our future work can be derived from the limitations described in Section 6.3.

Once technical limitations around memory size and access to host APIs in WebAssembly get gradually lifted, it could become a viable option to reverse the architecture of our data analytics tool. Instead of orchestrating program flow from JavaScript and using WebAssembly modules for performance-critical parts of the system, the WebAssembly module could determine the entire control flow, while only using a limited amount of imported JavaScript APIs. That way, the predictable and performant code execution of WebAssembly would cover all code paths, and additionally remove the need to serialize data between the boundary of JavaScript and WebAssembly.

If memory limitations in WebAssembly should stay for the foreseeable future, Buffer management implementations could be explored to allow working on data sets larger than 4GB. However, before exploring that topic, it would make sense to ensure that data sets of that size can be queried with acceptable performance in the browser.

Considering different query engine architectures could also be part of our future work to make query execution in a browser even faster, especially for large data sets. We have only discussed operator-centric query engines in this Section 2.3. However, using a data-centric, compiling query engine could unlock performance improvements of another order of magnitude.

A broader area of future work would be extending the functionality of our data analytics tool. Providing options to arrange the layout of widgets on the dashboard, extending the number of supported visualization types or implementing more decoding formats could be ideas for future improvements.

Making dashboards more interactive from the perspective of a dashboard user could be an interesting topic. Implementing a system to handle user events and make granular, partial updates to visualization would be part of that.

8 Conclusion

In this thesis, we set the goal to make analytical query processing in a web browser efficient. We examined how the WebAssembly instruction set works on the lowest level and why it can be executed faster than JavaScript. To understand how query performance is optimized for analytical workloads, we looked at the design choices for in-memory databases, columnar storage, and vectorized query execution. By inspecting a binary message format, we learned how query results are transferred with low overhead between the boundaries of WebAssembly and JavaScript.

Simultaneously, we explored how the process of creating dashboards can be made more interactive. We looked at how the Document Object Model works to create graphical user elements in a browser. With the goal of creating a dashboard programming language, we examined how parsers are constructed. Based on an architectural system design, we combined a parser for the dashboard programming language, a query engine in C++, and an implementation for a simple interpreter. As a result, we created a web-based data analytics tool.

To evaluate our work, we took both a qualitative and quantitative approach. By reconstructing example dashboards of popular web-based data analytics libraries, we concluded that our dashboard programming language is similarly expressive. In benchmarks measuring query performance for the same analytical problem, we compared our implementation to a competitor. As a result, we could validate our considerations about efficient query execution and observed a tenfold relative performance improvement.

Bibliography

[1] Institute for Health Metrics and Evaluation. *COVID-19 Projections.* 2020. URL: https://covid19.healthdata.org (visited on 08/20/2020).

[2] *Tableau Desktop.* URL: https://www.tableau.com/products/desktop (visited on 08/20/2020).

[3] *What is Power BI Desktop? – Power BI | Microsoft Docs.* URL: https://docs.microsoft.com/power-bi/fundamentals/desktop-what-is-desktop (visited on 08/20/2020).

[4] *Tableau Online | SaaS Analytics For Everyone.* URL: https://www.tableau.com/products/cloud-bi (visited on 08/20/2020).

[5] *Gallery | Tableau Public.* URL: https://public.tableau.com/s/viz-of-the-day (visited on 08/20/2020).

[6] *HTML Standard.* URL: https://html.spec.whatwg.org (visited on 08/20/2020).

[7] *Scalable Vector Graphics (SVG) 2.* URL: https://www.w3.org/TR/SVG (visited on 08/20/2020).

[8] *CSS Snapshot 2018.* URL: https://www.w3.org/TR/CSS (visited on 08/20/2020).

[9] *DOM Standard.* URL: https://dom.spec.whatwg.org (visited on 08/20/2020).

[10] *ECMAScript® 2021 Language Specification.* URL: https://tc39.es/ecma262 (visited on 08/20/2020).

[11] *asm.js.* URL: http://asmjs.org/spec/latest (visited on 08/20/2020).

[12] *WebAssembly Core Specification.* URL: https://www.w3.org/TR/wasm-core (visited on 08/20/2020).

[13] André Eickler Alfons Kemper. *Datenbanksysteme. Eine Einführung.* 10th ed. Oldenbourg Verlag, 2015. ISBN: 978-3-11-044375-2.

[14] Cristian Diaconu et al. "Hekaton: SQL Server's Memory-Optimized OLTP Engine." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* SIGMOD '13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 1243–1254. ISBN: 9781450320375. DOI: 10.1145/2463676.2463710.

[15] Franz Färber et al. "SAP HANA Database: Data Management for Modern Business Applications." In: *SIGMOD Rec.* 40.4 (Jan. 2012), pp. 45–51. ISSN: 0163-5808. DOI: 10.1145/2094114.2094126.

[16] Alfons Kemper and Thomas Neumann. “HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots.” In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering.* ICDE ’11. USA: IEEE Computer Society, 2011, pp. 195–206. ISBN: 9781424489596. DOI: `10.1109/ICDE.2011.5767867`.

[17] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. “Breaking the Memory Wall in MonetDB.” In: *Commun. ACM* 51.12 (Dec. 2008), pp. 77–85. ISSN: 0001-0782. DOI: `10.1145/1409360.1409380`.

[18] Mark Raasveldt and Hannes Mühleisen. “DuckDB: An Embeddable Analytical Database.” In: *Proceedings of the 2019 International Conference on Management of Data.* SIGMOD ’19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1981–1984. ISBN: 9781450356435. DOI: `10.1145/3299869.3320212`.

[19] D. Richard Hipp, Dan Kennedy, and Joe Mistachkin. *About SQLite.* URL: `https://www.sqlite.org/about.html` (visited on 08/20/2020).

[20] Marcin Zukowski and Peter Boncz. “From X100 to Vectorwise: Opportunities, Challenges and Things Most Researchers Do Not Think About.” In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’12. Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 861–862. ISBN: 9781450312479. DOI: `10.1145/2213836.2213967`.

[21] Alon Zakai. “Emscripten: An LLVM-to-JavaScript Compiler.” In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion.* OOPSLA ’11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 301–312. ISBN: 9781450309424. DOI: `10.1145/2048147.2048224`.

[22] *SQLite compiled to JavaScript.* URL: `https://github.com/sql-js/sql.js` (visited on 08/20/2020).

[23] Kareem El Gebaly and Jimmy Lin. “In-Browser Interactive SQL Analytics with Afterburner.” In: *Proceedings of the 2017 ACM International Conference on Management of Data.* SIGMOD ’17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1623–1626. ISBN: 9781450341974. DOI: `10.1145/3035918.3058736`.

[24] M. Bostock, V. Ogievetsky, and J. Heer. “D^3 Data-Driven Documents.” In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2301–2309.

[25] A. Satyanarayan et al. “Vega-Lite: A Grammar of Interactive Graphics.” In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 341–350.

[26] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation.” In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[27] *Protocol Buffers | Google Developers.* URL: `https://developers.google.com/protocol-buffers` (visited on 08/20/2020).

[28] *The Fast Lexical Analyzer – scanner generator for lexing in C and C++.* URL: `https://github.com/westes/flex` (visited on 08/20/2020).

[29] *Bison – GNU Project – Free Software Foundation.* URL: `https://www.gnu.org/software/bison` (visited on 08/20/2020).

[30] *lex – generate programs for lexical tasks.* URL: `https://pubs.opengroup.org/onlinepubs/009695399/utilities/lex.html` (visited on 08/20/2020).

[31] *yacc – yet another compiler compiler.* URL: `https://pubs.opengroup.org/onlinepubs/009695399/utilities/yacc.html` (visited on 08/20/2020).

[32] *Stacked Bar Chart with Rounded Corners | Vega-Lite.* URL: `https://vega.github.io/vega-lite/examples/stacked_bar_count_corner_radius_mark` (visited on 08/20/2020).

[33] *Seattle Weather Example Data.* URL: `https://vega.github.io/editor/data/seattle-weather.csv` (visited on 08/20/2020).