

Eberhard Karls Universität Tübingen

Mathematisch-Naturwissenschaftliche Fakultät

Wilhelm-Schickard-Institut für Informatik

Database Systems Research Group

Master of Science Informatik

FROM POSTGRESQL TO DUCKDB: UNUSUAL QUERIES, THEIR PERFORMANCE, AND READABILITY

ANN-KATHRIN CLAESSENS

March 03, 2025

Examiner

PROF. DR. TORSTEN GRUST

Co-Examiner

JUN.-PROF. DR. JONATHAN IMMANUEL BRACHTHÄUSER

Supervisor

DENIS HIRN

Ann-Kathrin Claessens:

From PostgreSQL to DuckDB:

Unusual Queries, their Performance, and Readability

Master of Science Informatik

Eberhard Karls Universität Tübingen

01.09.2024 – 03.03.2025

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe. Des Weiteren erkläre ich, dass die Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist und dass das elektronische Exemplar der Abschlussarbeit exakt mit dem gedruckten und gebundenen Exemplar / den gedruckten und gebundenen Exemplaren übereinstimmt.

Tübingen, 03.03.2025

Ort, Datum



Unterschrift

Abstract

Quassnoi publishes each year a PostgreSQL query on his blog¹ that implements the solution for a challenging problem that requires a deep understanding of SQL. These problems do not represent use cases that are usually realized using SQL. Some of them even subvert common expectations regarding the type of problems that can be solved using SQL. This thesis aims to port some of the queries proposed by Quassnoi to DuckDB, optimize their readability for educational purposes and investigate their performance. This process requires considering the differences between both database management systems and exploring options supported by DuckDB to improve performance and readability.

¹<https://explainextended.com/>

Contents

Selbständigkeitserklärung	iii
Abstract	iv
1. Introduction	8
1.1. Motivation	8
1.2. Porting PostgreSQL Queries to DuckDB	8
1.3. Optimizing Readability	8
1.3.1. Readability for SQL Queries	9
1.3.2. Strategies to Improve Readability	10
1.3.3. Readability Metrics	11
1.4. Query Performance Evaluation	12
2. Topic 1: GIF Decoder	14
2.1. Motivation	14
2.2. Problem Specification	14
2.3. From PostgreSQL to DuckDB	15
2.3.1. Processing Hex-encoded Data	15
2.3.2. Storing the LZW Code Table	18
2.3.3. Shortcut for Conditional Branches	19
2.3.4. Additional Changes	21
2.4. Readability Optimizations	22
2.4.1. Discussion of the Porting Effects on Readability	22
2.4.2. Documentation of Changes for Improved Readability	24
2.4.3. Conclusion	25
2.5. DuckDB GIF Decoder Query Explained	28
2.5.1. Preparing the Input Data	28
2.5.2. Calculating the Offset to the Compressed Image Data	29
2.5.3. Extracting the Compressed Image Data	34
2.5.4. LZW Decompression Algorithm	36
2.5.5. Rendering the Image	42
2.6. Performance Evaluation	44
2.6.1. Effects of Readability Optimizations and CTE Materialization	44
2.6.2. Performance Optimizations	45
2.6.3. Conclusion	49
3. Topic 2: GPT inference	50
3.1. Motivation	50
3.2. Problem Specification	50
3.3. From PostgreSQL to DuckDB	52
3.3.1. Unnesting With Ordinality	52
3.3.2. Adapting the Tokenizer for DuckDB	52
3.3.3. Vector and Matrix Calculations	55
3.3.4. Other Necessary Changes	58
3.4. Readability Optimization	60
3.4.1. Discussion of the Porting Effects on Readability	60
3.4.2. Further Improving Readability	60
3.4.3. Conclusion	64
3.5. DuckDB GPT Inference Query Explained	65

3.5.1. Setting the Input Parameters	65
3.5.2. Tokenizing the Prompt	66
3.5.3. Recursively Inferring the Transformer	71
3.5.4. Recursively Calculating the Block Outputs	73
3.5.5. Multi-Head Self-Attention	75
3.5.6. Feed Forward	81
3.5.7. Choosing the Next Token	83
3.5.8. Query Output	86
3.6. Performance Evaluation	88
3.6.1. Effects of Readability Optimizations and CTE Materialization	88
3.6.2. Performance Optimizations	88
3.6.3. Conclusion	91
4. Conclusion	92
Bibliography	94

Introduction

1.1. Motivation

The Structured Query Language (SQL) is a programming language originally created for managing data stored in databases [1]. However, it has been expanded over time to a complete language that can be used to solve a variety of problems. In order to showcase the possibilities that this programming language provides, Quassnoi publishes a blog² post around each new year that presents an unusual and challenging project to be solved using a PostgreSQL query. In this thesis, a selection of these queries is ported from PostgreSQL to DuckDB (v1.1.1). The motivation for this is that PostgreSQL and DuckDB are database management systems that specialize on different use cases. Other than PostgreSQL, DuckDB is an embeddable database management system that specializes on analytical query workloads, where complex calculations are performed on stored datasets [2]. This makes it a promising candidate to support the type of complex and unusual queries presented by Quassnoi. Porting these queries to DuckDB allows to compare their performance in either database management system and thus explore how their differences affect query performance. After porting the queries, their DuckDB implementation is optimized in this thesis with regards to their readability. This optimized version of the query is then presented and explained in a dedicated subsection of each chapter, so that they can be utilized for educational purposes. Lastly, the performance of the queries is evaluated and further possibilities to improve their performance in DuckDB are explored.

1.2. Porting PostgreSQL Queries to DuckDB

Since DuckDB uses the PostgreSQL parser [3], there is a lot of overlap between the syntax supported by both database management systems. Nevertheless, a number of differences exist between the two as to which data types are supported, how type castings between them behave, as well as to the names and return types of certain functions. Furthermore, there are functions and even some SQL clauses that exist in PostgreSQL, but are not supported by DuckDB altogether. In many such cases, alternatives that provide the same functionality can be found in the DuckDB documentation [3]. But whenever a clause, a function or a query structure is encountered for which no direct equivalent is supported by DuckDB, successfully porting the query also requires looking at their context. By determining which goal or intermediate result they help to achieve, alternative ways to implement them can be explored. Sometimes, this may require representing certain data in a different manner than in the original query. This will then also affect the implementation of any operations performed on this data. Some queries discussed in this thesis are designed on the premise that certain types of data can be interpreted or transformed using a set of operations that forms the basis of the query's implementation. In this case, a fitting data representation that allows to realize the same operations in DuckDB has to be identified.

1.3. Optimizing Readability

Porting queries from one DBMS to another can have large effects on their readability. One aspect that greatly impacts readability is the availability of functions with descriptive names that fit the requirements for the query. Whenever a function that provides exactly the functionality needed in the query has to be substituted with an expression, the result may be the same, but the program's readability

²<https://explainextended.com/>

decreased. There may also be functions supported by DuckDB that provide functionality that requires longer implementations in PostgreSQL. These differences will be explored for every query discussed in this thesis.

1.3.1. Readability for SQL Queries

SQL originates from the Structured English Query Language (SEQUEL) presented by Chamberlin and Boyce in 1947 [4]. The syntax of this query language is composed of English keywords and is meant to “reflect how people use tables to obtain information”. As a result, SEQUEL, which was later renamed to SQL, was designed to be declarative. Its keywords should allow the programmer to describe what should be computed, rather than having to specify how it is computed [5]. The hope for this design approach was to make SQL easier to read and understand than other languages that make use of a more “mathematical notation” [4]. This is reflected by the components of a basic SQL query block. The **SELECT** clause lists the columns whose values should be retrieved from the tables listed in the **FROM** clause, where only rows that fulfill the condition specified in the optional **WHERE** clause are considered. For simple queries consisting of a single query block like in Listing 1, SQL thus almost reads like an English sentence describing the data that should be retrieved.

```
1  -- Get the names of all customers that are pilots.
2  SELECT  name
3  FROM    customers
4  WHERE   occupation = 'pilot';
```

SQL

Listing 1: A simple SQL query selecting data from a table named “customers” with columns named “name” and “occupation”

However, maintaining readability for much more complex queries like the ones that will be discussed in this thesis is a lot more challenging. One common reason for that is nesting. **SELECT** statements nested within another query block often serve to define intermediate results, but as their number increases, it also means that there are more parts to the query that have to be understood both individually and in relation to each other. This can quickly lead to a high level complexity, as SQL query blocks can be nested to an arbitrary depth. As a consequence, it can become nearly impossible to read queries from top to bottom. This has to do with one particularity of SQL syntax concerning the order in which the clauses of query blocks are written. Although the **SELECT** clause is usually written above the **FROM** and **WHERE** clauses, the execution starts with the **FROM** clause. It determines which table rows may be selected from, as well as the aliases that can be used to access their columns. Consequently, reading the **FROM** clause first is often a requirement for being able to understand the contents of the **SELECT** clause. This is especially true for larger queries where the **FROM** clause contains nested subqueries, or queries that make use of the *****-notation to select all columns from a table or subquery without explicitly listing them. Queries that are deeply nested may even have to be read starting from the bottom, at the innermost **FROM** clause, before parsing the **SELECT** clauses of the surrounding query blocks. Although DuckDB does support **FROM**-first syntax, it is not applied as a part of the readability optimization in this thesis. As the goal is to make the queries presentable for educational purposes, this syntax would conflict with the SQL syntax taught in lectures and courses, thus subverting any benefits it may or may not have.

As an alternative to nesting query blocks, they can be defined as auxiliary queries called Common Table Expressions (CTEs) using a **WITH** clause [3]. This clause begins with a list of CTE definitions that assign aliases to subqueries, which can then be referenced in the **FROM** clause of the main query block or in other CTE definitions like any other table. Defining subqueries that would otherwise be nested into other query blocks as CTEs at the beginning of the **WITH** clause allows to read them from top to bottom, as if they represent the implementation of a series of subgoals to solve the problem. This

approach is used in all PostgreSQL queries discussed in this thesis. For the purpose of optimizing their readability after porting them to DuckDB, the potential of further improving the query's readability by renaming and restructuring some of its CTE definitions is explored. This is also discussed further in Section 1.3.2.

Complexity that negatively impacts the readability of SQL queries not only manifests in their query structure, but also in the expressions used in their clauses. Expressions that are very long or include a multitude of function calls make column definitions harder to grasp. This is especially problematic when it affects expressions that are repeated several times throughout the query. One method that counteracts the negative impact of such expressions on the readability of the query as a whole that is applied in this thesis is to define them as functions. DuckDB supports the `CREATE MACRO` statement (alias `CREATE FUNCTION`), which can be used to assign descriptive names to scalar expressions or single `SELECT` statements. After a macro has been defined, its name can be used inside queries in place of its definition. Long and complex calculations that are repeated throughout a query can thus be hidden in macro definitions. Despite their official alias “function”, macros do not lead to function calls being executed in the same way as user-defined functions (UDFs) would. Instead, the macro names are internally replaced by their definition. [3] As a consequence, using them to improve the readability of a query does not lead to a major negative effect on its performance caused by overhead of function calls.

1.3.2. Strategies to Improve Readability

The methods used in this thesis to improve the readability of SQL queries are inspired by a selection of clean code principles proposed by Robert C. Martin [6]:

- **Don't Repeat Yourself (DRY):**
Instead of code duplication, create functions (or in this case: macros) with a descriptive name.
- **Single Responsibility Principle (SRP):**
Functions should serve one purpose and one purpose only. This can be applied to macros and CTEs.
- Names should be meaningful.
- Chosen formatting rules should be applied consistently.

In the following, it will be discussed in some more detail how these principles are applied to SQL. This section elaborates the goals that were committed to when optimizing the DuckDB queries for readability.

Common Table Expressions (CTEs): All PostgreSQL queries discussed in this thesis already make use of a `WITH` clause. For each of them, it will be examined in this thesis how its use of auxiliary queries can be further optimized to improve the readability of the program as a whole. This includes determining whether there still are deeply nested subqueries blocks present and if defining them as CTEs instead would be beneficial. Additionally, inspired by the clean code principles, CTEs are designed so that each of them serves only exactly one purpose. As a consequence, or as a means to ensure this, each CTE can be assigned a descriptive name that does not omit any other central tasks the CTE may have. It should be possible to describe its purpose in one simple sentence.

Defining macros: For queries that repeatedly perform a small set of operations, these operations are defined as macros. This way, the implementation of these operations is only written in one place, thus avoiding code duplication. Additionally, it provides the benefit of using the macro's descriptive name instead of its potentially long and complex definition.

Choosing meaningful names: Table names should be descriptive, accurately describing the result set they produce. Column names are chosen so that they describe the purpose or other relevant information about its values. Meaningful names can at times even help to remedy the issue of SQL clauses being written in the “wrong” order. Well-written `SELECT` clauses that use descriptive names have the potential to become “descriptive” themselves, providing a general idea of what the query

does even upon the first read. The exact definition of the values used in the `SELECT` clause can then be found further down in the `FROM` clause. With this approach, identifying the purpose of a query block becomes easier to identify, as it may not require reading and understanding the whole block. Aside from names being descriptive, it is also advantageous to have the names match the vocabulary used in the documentation that served as the basis for the original query. Being able to identify for every part of the program which part of the algorithm it implements is an important requirement for understanding the query. Matching the language used in the query to the one used in the problem specification can make this a lot easier.

Consistency: There are different preferences when it comes to indentation styles or when to use uppercase characters. For the readability optimizations performed in this thesis, the goal is to stick to widely favored style choices (e.g., uppercase for all SQL keywords and functions). Most importantly, these choices are applied consistently throughout the code. Other instances for upholding consistency include preserving the column order when selecting from tables or CTEs, as well as keeping the original column names unless they are used in new context that justifies a name change.

Other stylistic choices: The PostgreSQL queries discussed in this thesis use keywords `CROSS JOIN` to join multiple tables, rather than separating them by commas in the `FROM` clause. Conditional joins are written using the keyword `JOIN` in combination with `ON` or `USING` to specify the join condition. This syntax is also used in this thesis, as it lends itself well for describing the purpose of the `FROM` clause. Beyond that, the goal is avoid deviating from “textbook” syntax. For example, there are circumstances where SQL keywords are optional and can be omitted without causing a syntax error. In order to make the optimized queries more suitable for educational purposes, omitting optional keywords or other syntax is refrained from. This includes listing all column aliases of a CTE right after its name declaration. The `SELECT` clause of a CTE often does not make it obvious at a single glance which columns it defines. This may be because it includes long expressions that have to be skipped over in order to find the column alias, or because the `*` expression is used instead of explicitly listing all selected columns. Writing out the column names at the top of the CTE definition makes them easier to identify.

1.3.3. Readability Metrics

For lot of the strategies applied to improve readability, there is no means to measure their effectiveness. Instead, some quantifiable aspects related to query complexity are examined in this thesis. By choosing a number of metrics that give some amount of insight into the complexity of the query structure, different versions of the same query can be compared based on these criteria. The metrics chosen for this thesis are:

- the total number of Common Table Expressions
- the total number of query blocks
- the maximum depth of nesting
- the number of query blocks nested at a certain depth

A low value for these metrics implies a lower degree of complexity in that regard. Readability improvements like removing code duplication will be reflected by the metric as a lower query block count. Although the number of CTEs gives an impression of the size of the query, a higher number of CTEs can be the result of reducing the depth of nesting (as discussed in Section 1.3.2). As defining nested subqueries as CTEs increases the number of CTEs, but not the total number of query blocks that must be read, a reasonable increase of that metric is accepted if it serves to reduce nesting.

In addition to considering the query structure, the effects of defining macros on the complexity of expression in the query is examined. Whenever macros are used to reduce the number and length of complex expressions that appear in the query, their effect on the query is quantified by counting their occurrences as well as the number of operations they hide in their definition.

1.4. Query Performance Evaluation

The last aspect that will be discussed for every topic in this thesis is query performance. To that end, the first step is to examine how the DuckDB implementation obtained after porting the PostgreSQL query to DuckDB performs compared to the original. As both database management systems employ different approaches to query execution and are specialized on different types of workloads, large differences in the performance may be observed depending on the query. Furthermore, porting the queries often requires applying a number of changes to the implementation, which may perform differently than the original. Any differences that can be observed are discussed before exploring options to optimize the queries for performance.

In order to observe a query's performance, both PostgreSQL and DuckDB support the `EXPLAIN ANALYZE` statement, which can be used to obtain not only the query plan, but also the total execution time and performance numbers of the individual operators in this plan. In this thesis, this tool will be used to compare the total execution time of the DuckDB query to the original PostgreSQL query and the DuckDB query optimized for performance. In order to avoid outliers falsifying the comparison, the execution time is averaged over ten consecutive measurements. Additionally, the reason behind any performance differences will be discussed by looking at the performance numbers of the operators in the respective query plan. This query plan alongside an overview which operations contribute the most to the execution time is obtained from a query graph tool provided in the DuckDB repository³.

One aspect that is discussed for every query is CTE materialization. As mentioned previously, the SQL queries that will be considered in this thesis all make use of the `WITH` clause to define a row of Common Table Expressions. However, the rules behind how these clauses are evaluated differ in DuckDB compared to PostgreSQL. PostgreSQL's `WITH` clause has the property that the CTEs are evaluated only once per execution of the parent query, unless enforced otherwise. [7] This is not automatically the case for DuckDB. According to DuckDB's documentation [3], a CTE is only materialized if it includes a grouped aggregation and is referenced more than once. Otherwise, the CTE is inlined into the main query, which leads to code duplication if the CTE is referenced multiple times throughout the query. Adding the optional keyword `MATERIALIZED` enforces that DuckDB evaluates the CTE only once.

³<https://github.com/duckdb/duckdb>

Topic 1: GIF Decoder

2.1. Motivation

In Quassnoi's 10th New Year post [8] from December 2018, he presents an implementation of a GIF decoder in SQL. The task of this program is to process a string containing compressed image data in the Graphics Interchange Format (GIF), decompress the data and finally render the image in the terminal. The motivation for this exercise is to provide the challenge of realizing the decompression algorithm applied by GIF decoders as an SQL query. For the decompression, the post follows the tutorial "What's in a GIF?" [9] written by Eric S. Raymond and Mike Flickinger. The finished query has the ability to print a variety of greyscaled images in the terminal. This makes it attractive for teaching purposes, which is why this topic was chosen for this thesis.

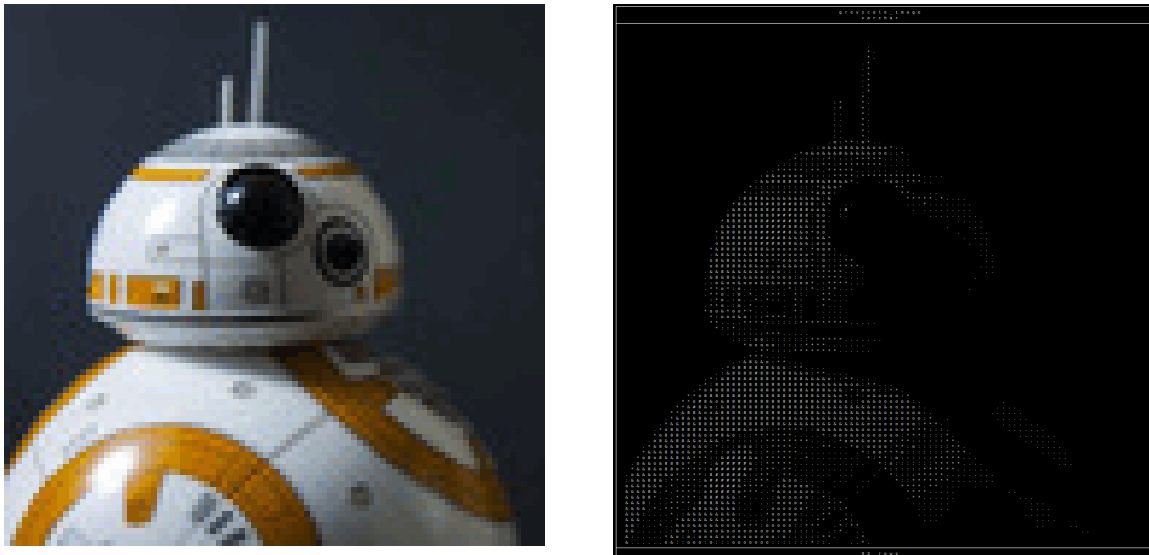


Figure 1: Example GIF image (left) and the GIF decoder query's terminal output (right), GIF image created and modified from [10]

Although GIF files can also contain data for multiple graphics or animations ever since the release of version GIF89a [11], this GIF decoder query is only designed for single GIF images.

2.2. Problem Specification

GIF was presented in 1987 by CompuServe Incorporated [12]. In this first specification of the format, GIF is described as a "standard for defining generalized color raster images". The use of raster data is explained to mean that the images in question can be represented as a sequence of pixel colors. The length of this sequence matches the number of pixels that make up the image. Assigning the colors in this sequence to the pixels, commonly in the order left to right and top to bottom, produces the image. The pixel colors are each represented by a number value, namely a pixel color index. These indices can be used to retrieve the color's RGB color values from a color table that is contained in the GIF file. With such a table at hand, raster images can thus be represented as a string of numbers, or pixel color indices, with one number for every pixel in the image.

This means that images that contain a large number of pixels require a lot of storage. GIF was one of the first image formats that resolves this issue by applying lossless image data compression [13]. To that end, it utilizes a variant of the Lempel–Ziv–Welch (LZW) compression algorithm described in an article by Terry A. Welch [14]. As stated in the GIF specification [12], this algorithm is used to transform the sequence of color indices representing the image into a sequence of variable length codes. Each of these codes is assigned to a color pattern, *i.e.*, a series of color indices. By representing the image data as a sequence of such codes instead of storing a color index for every single pixel, fewer values need to be stored.

The contents of a GIF file, which includes the code sequence representing the graphic, are also referred to as a “GIF Data Stream” by the GIF specification. A GIF Data Stream is defined as a sequence of blocks of data containing all relevant information required to render a graphic. This includes the color table of that graphic, the compressed image data in the form of a sequence of codes and further control data. A GIF decoder is a program that takes a GIF Data Stream as input and processes it. The decoder extracts relevant parameters from the data, locates the compressed image data in the data stream, decompresses it in order to obtain a sequence of pixel color indices and finally looks up the RGB color values for every pixel in the color table. At that point, the GIF decoder query discussed in this chapter can apply a formula to calculate the brightness of each pixel color and select a character to print in the terminal that matches its brightness. These characters are chosen from a fixed sequence of characters that serves as the color palette for the greyscale image, ordered by how bright they appear because of how much area they cover. Aggregating the characters for all pixels of the same row in the correct order into a string results in an output table whose row values appear like a greyscale image in the terminal.

2.3. From PostgreSQL to DuckDB

In this subchapter, the changes that had to be made to the SQL query in order to port it from PostgreSQL to DuckDB will be discussed. During that process, it also becomes clear how the differences between the two database management systems manifest in this particular program.

2.3.1. Processing Hex-encoded Data

As discussed in Section 2.2, the main objective of the SQL query is to process a GIF Data Stream. A central part of this problem is thus to implement a set of operations to access and interpret parts of such a data stream. These operations will then be used at many points throughout the GIF decoder query, forming the basis needed to realize the rest of the program.

Before defining these operations, the format of the input data has to be considered with the goal of determining how it is best accessed by the query. The input data is made available to the query by inserting the hex-encoded contents of a GIF file into a single-row, single-column table called “input”. The data stream is thus initially in the form of a single string of hexadecimal digits, stored as the only value in this input table. Then, a decision has to be made as to which data type will be the best fit for this hex-encoded input data in order to make processing it as convenient as possible. To that end, it is necessary to consider which operations are required for handling the data that this query will be working with. The “blocks” of data that make up a GIF data stream are defined as a series of fields that provide a variety of information. How different kinds of blocks can be located in the data stream and what information they hold is documented in the GIF specification. Every time the query has located a substring of hexadecimal digits from the input sequence that belongs to a block whose field values are of interest, it will have to extract and convert a subset of them. Converting the first few digits to a decimal number might for example tell us the width of the image in pixels. Others may be a hexadecimal representation of a sequence of binary digits that indicates, *e.g.*, the value of a code that

can be mapped to a pattern of color indices. This means that aside from applying knowledge about the structure of GIF files, the program will repeatedly have to perform the following operations:

- extract a substring of hexadecimal digits from the input sequence
- convert a sequence of hexadecimal digits to a decimal number
- convert hexadecimal digits to binary digits
- extract a substring of binary digits from a bitstring and convert them to a decimal number

These operations are the building blocks used in almost every part of the program. They will be realized using functions and typecasting supported by the database management system. Since PostgreSQL and DuckDB differ in regards to which data types and functions are supported, how these base operations are realized will be one of the major ways in which this DuckDB query differs from the original PostgreSQL query. As these differences occur in nearly every part of the program, it makes sense for them to be discussed separately before discussing the program itself.

PostgreSQL supports the data type `BYTEA` for variable-length binary strings. This type allows data to be encoded in “hex” format as two hexadecimal digits per byte [7]. This perfectly matches the original format of the GIF decoder query’s input. Furthermore, PostgreSQL supports several functions that extract a certain amount of data from a `BYTEA` value. The original GIF decoder query uses the binary string functions `GET_BYTE(string, offset)`, which extracts one byte of data, and `SUBSTRING(string, from_int, for_int)`, which extracts a substring from the data.

Since the return type of `GET_BYTE` is conveniently `INTEGER`, the function also automatically handles the conversion of the extracted hexadecimal digits to a decimal number. Although `SUBSTRING` returns a value of type `BYTEA` if the first parameter is a `BYTEA` value, this function is only used as an intermediate step. As soon as any of the bytes it returned need to be converted to decimal, the byte in question is simply extracted using the `GET_BYTE` function. An overview of the `BYTEA` functions that appear in the PostgreSQL query, including their type signatures, as well as the type conversions performed around binary data, is shown in Table 1.

In order to convert hexadecimal digits to binary digits, the `INTEGER` value returned by `GET_BYTE` can be explicitly cast to type `BIT(n)`. The resulting value will contain the n most significant bits of the number’s binary representation. That same method can also be used to extract a substring of binary digits from a bitstring. The m most significant bits of a value of type `BIT(n)` is obtained by casting the value to `BIT(m)`. If not the most significant bits need to be extracted, the bitwise shift left operator `<<` can be used to shift the relevant substring to the most significant bits of the value before performing the typecasting.

There are some cases in which a value spans two bytes instead of just one. In that case, the bytes need to be extracted separately and their order reversed, since these multi-byte fields use the little-endian format. This is achieved by casting the decimal values of the individual bytes to `BIT(8)` and then using the bit string concatenation operator `||` to concatenate the bitstrings in the desired order. The resulting value can be cast to type `BIT(16)` and then `INTEGER` to obtain the decimal value of the multi-byte field.

Description	Signature	Expression
Extract one byte (two hexadecimal digits) at byte index i from hex-encoded data “bytes”	<code>BYTEA -> INTEGER</code>	<code>GET_BYTE(bytes, i)</code>
Extract n bytes starting at byte index i from hex-encoded data “bytes”	<code>BYTEA -> BYTEA</code>	<code>SUBSTRING(bytes, i, n)</code>
Convert a byte of hex-encoded data to a decimal number	-	occurs automatically at extraction
Convert the value of a byte of hex-encoded data (“byte_val”) to binary digits	<code>INTEGER -> BIT(8)</code>	<code>byte_val::BIT(8)</code>
Extract one bit at index i from a bitstring “bits”	<code>BIT(n) -> BIT(1)</code>	<code>(bits << i)::BIT(1)</code>
Extract m bits from a bitstring “bits” of length n , starting from index “offset”	<code>BIT(n) -> BIT(m)</code>	<code>(bits << offset)::BIT(m)</code>
Convert a bitstring “bit_string” to a decimal number	<code>BIT(n) -> INTEGER</code>	<code>bit_string::INTEGER</code>
Concatenate “byte_1” and “byte_2” in reversed order to obtain the value of a multi-byte field	<code>(INT, INT) -> INT</code>	<code>(byte_2::BIT(8) byte_1::BIT(8))::BIT(16)::INT</code>

Table 1: Overview of base operations used in the original PostgreSQL query.

DuckDB also supports a variable-length binary data type named `BLOB` (Binary Large Object), for which the alias `BYTEA` exists, too. However, at the time of writing this thesis, DuckDB does not support a function such as `GET_BYTE` that allows the extraction of one byte of data. Furthermore, DuckDB’s `SUBSTRING` function only works as a text function, not for values of type `BLOB`. Typecasting from `BLOB` to `BITSTRING` (DuckDB’s type for strings of 1s and 0s, also alias `BIT`) does exist, but it just returns the binary encodings for every character in the `BLOB` value appended together, rather than performing a conversion from hexadecimal digits to binary digits. These points combined make DuckDB’s version of the `BYTEA` data type unfitting for the input data of this query.

The solution is to use the data type `VARCHAR` (alias `STRING` or `TEXT`) to store the hexadecimal digits, instead. This allows the use of the `SUBSTRING` function to extract an arbitrary number of digits from the string. A notable difference with this approach is that `SUBSTRING` applied to text types operates on single characters (one hexadecimal digit) instead of bytes (two hexadecimal digits at a time). Taking this into account, the function can be used to achieve the same results as the two functions supported by PostgreSQL for type `BYTEA`, as presented in Table 2, just with differing type signatures.

Because `SUBSTRING` returns a string of hexadecimal digits instead of automatically converting them to a decimal value like PostgreSQL’s `GET_BYTE` function, handling multi-byte values actually becomes a little more straightforward. Rather than having to convert the value of each byte to `BIT` in order to concatenate the bytes in reversed order, the string concatenation operator `||` can be used directly on the strings containing the hexadecimal digits for each byte. The full expression as well as all other operations discussed in the following can be found in Table 2.

The `VARCHAR` type also makes conversions to both integer or binary types possible, although they may not be quite as elegant. To obtain the decimal representation of a string of hexadecimal digits in DuckDB, the `VARCHAR` value is prefixed with the string literal `"0x"` and then cast the result to type `INTEGER`. For the binary representation, the value can then be cast further to type `BITSTRING`. Having to

cast to `INTEGER` first before being able to cast to `BITSTRING` is a not particularly intuitive but necessary intermediate step that is hidden in the function call of `GET_BYTE` in PostgreSQL, but becomes visible in DuckDB.

The `SUBSTRING` function can also be used to extract bits from a `BITSTRING` after casting it to `VARCHAR`. However, it should be noted that if the `BITSTRING` was obtained by casting an `INTEGER` value, it automatically has a length of 32, which includes a number of leading zeroes before the relevant bits. Unlike PostgreSQL’s `BIT(n)` type, DuckDB’s `BITSTRING` type does not support any modifiers dictating the length of the value. There exists a function that returns a bitstring of a determined length (`bitstring(VARCHAR, INTEGER) -> BIT`), but as of version v1.1.1., this function only works for input strings of 1s and 0s whose length is equal or smaller than the indicated length. In other words, this function can add leading zeroes to a string of binary digits, but it cannot discard any digits that were originally there. This has to be taken into consideration before extracting a substring. For example, the most significant relevant bit in such a bitstring would not be located at index 1, but at index $(33 - \text{\#relevant_bits})$.

Description	Signature	Expression
Extract two hexadecimal digits at byte index i from hex-encoded data “bytes”	<code>VARCHAR -> VARCHAR</code>	<code>SUBSTRING(bytes, i * 2 + 1, 2)</code>
Extract n bytes starting at byte index i from hex-encoded data “bytes”	<code>VARCHAR -> VARCHAR</code>	<code>SUBSTRING(bytes, i * 2 + 1, n * 2)</code>
Convert hexadecimal digits “hex_digits” to a decimal number	<code>VARCHAR -> INT</code>	<code>('0x' hex_digits)::INT</code>
Convert hexadecimal digits “hex_digits” to binary digits	<code>VARCHAR -> VARCHAR</code>	<code>('0x' hex_digits)::INT::BIT::VARCHAR</code>
Extract one bit at position index i from a string of bits	<code>VARCHAR -> VARCHAR</code>	<code>bits[i]</code>
Extract m bits (from a string of bits of length n) at position index i	<code>VARCHAR -> VARCHAR</code> <code>(INT -> BIT ->)</code> <code>VARCHAR -> VARCHAR</code>	<code>SUBSTRING(bits, i, m)</code> <code>SUBSTRING(bits, 33 - n + (i-1), m)</code>
Convert a string of bits “bits_str” to a decimal number	<code>BIT -> INT</code> <code>VARCHAR -> INT</code>	<code>bit_str::INT</code> <code>('0b' bit_str)::INT</code>
Get the value of a field spanning two bytes	<code>(VARCHAR, VARCHAR) -> INT</code>	<code>('0x' byte_2 byte_1)::INT</code>

Table 2: Overview of the base operations used in the DuckDB query.

2.3.2. Storing the LZW Code Table

LZW decompression uses a dictionary table that assigns codes (binary numbers) to a color pattern (a sequence of color indices). As the decoder processes the compressed image data, one code after the other, the decompression algorithm both looks up the color patterns for the codes in the table and also

builds the table at the same time. The compression and decompression algorithms are designed so that the code table that was used during the compression of the image does not have to be stored in the GIF file, as the decoder can create it during decompression. [11] This means the query has to keep track of the current state of the code table throughout all iterations of the algorithm and perform updates on it.

Initially, the code table has one entry for every trivial color pattern consisting of a single color, *i.e.*, one entry for every color in the image [15]. Then, as the decoder processes the sequence of codes extracted from the GIF Data Stream, it checks for every code whether there already exists an entry for it in the code table. In either case, the decoder also adds a new entry to the table in each iteration. As a consequence, the code table keeps growing until it reaches a maximum size of 4095 (0xFFFF) entries, unless the decoder encounters a “clear code” that prompts it to reset the table to its initial state. Clear codes can appear at any point in the code sequence, as the choice when to reset the table in order to make space for new entries lies with the encoder who generated it [16].

PostgreSQL’s type `HSTORE` for storing key/value pairs [7] is used in the original query to store the code table. This type is favored over arrays for its more efficient key lookups. It requires that all keys and values are of type `STRING`, which is achieved in this case via type casting. [8].

DuckDB supports the type `MAP` to store an ordered list of key/value pairs [3]. This type requires that all keys have the same type and all values have the same type. However, which type is chosen for either of them is arbitrary, making casting to `VARCHAR` (alias `STRING`) unnecessary. It can be used to perform the same operations on the code table as is done using by `HSTORE` in PostgreSQL, as shown in Table 3.

Operation	PostgreSQL	DuckDB
Create an empty dictionary	<code>NULL::HSTORE</code> <code>HSTORE (ARRAY [] ::TEXT [] [])</code>	<code>MAP () ::MAP (<key_type>, <value_type>)</code>
Get the value associated with a key from the table	<code>dictionary->key</code>	<code>dictionary[key] [1]</code>
Add a new key/value pair to the table	<code>dictionary </code> <code>HSTORE (key, value)</code>	<code>MAP_CONCAT (dictionary,</code> <code>MAP {key: value})</code>

Table 3: Comparing table operations for PostgreSQL and DuckDB

2.3.3. Shortcut for Conditional Branches

The original GIF decoder query distinguishes between three different cases in its implementation of the decompression algorithm. To realize this case distinction in an efficient manner, Quassnoi takes advantage of the way PostgreSQL’s query execution works to implement a shortcut similar to a return statement in imperative languages [8]. These three different cases are realized as three individual query blocks without a `FROM` clause, connected by `UNION ALL`. From this union, only one row is selected using the statement `LIMIT 1`. The query structure of this shortcut implementation is shown in Listing 2. The idea, as it is described in the blog post, is to only select the result of the first subquery that returns a row, *i.e.*, the first whose condition is not false. Once a result row has been produced, the remaining subqueries do not need to be executed.

```

1  SELECT *
2  FROM (
3      SELECT ...
4      WHERE <cond_1>
5      UNION ALL
6      SELECT ...
7      WHERE <cond_2>
8      UNION ALL
9      SELECT ...
10     FROM <nested sourceless subqueries>
11     ) b
12 LIMIT 1

```

Listing 2: `LIMIT` shortcut for three conditional branches implemented as sourceless query blocks

Although this code does not cause DuckDB to throw any exceptions, it performs very differently than originally intended. The cause of this is a fundamental difference between how query execution works in DuckDB compared to PostgreSQL. PostgreSQL performs pull-based query execution [17]. This means that the query execution does not wait for the full intermediate result to be ready. Instead, query plan operators request the parts of the result one at a time, by calling the next method of its source operator. Result data is generated lazily, only once the next method has been called. [18] If a `LIMIT` statement is present, this process can simply come to a halt once the intended number of result rows have been requested. This laziness also applies to conditional query blocks connected by `UNION ALL`. In PostgreSQL’s query plan, the subplans for each of these query blocks have a `RESULT` operator as a parent node that evaluates the filter condition in the `WHERE` clause before the query block even gets executed [17].

DuckDB’s query execution on the other hand is push-based, which means that the source operators are the ones that push their result in chunks to the operators depending on it [3]. Looking at the query execution plan for this implementation provided by DuckDB’s query graph tool, it is revealed that it does produce a huge intermediate result, which is then limited using a window function. This is not the intention here, and therefore this part of the query has to be written differently for DuckDB.

As an alternative to the original idea, `CASE` statements can be used to differentiate between the three branches. However, the output of a `CASE` expression can only be a single value, not a query with multiple columns. Consequently, the same case distinction is duplicated in each column definition. Because the column values depend on each other in some cases, their definitions could not be placed within a single `SELECT` clause, as aliases of sibling columns cannot be referenced in this particular type of correlated subquery. Instead, some of them are defined as CTEs of a nested `WITH` clause. As a result, the overall structure of this particular part of the GIF decoder query, presented in Listing 3, looks very different from the original.

```

1 WITH cte1 AS (
2     SELECT CASE WHEN <cond 1> THEN ...
3             WHEN <cond 2> THEN ...
4             ELSE ...
5     END AS column_1,
6 ),
7 ...
8 )
9 SELECT column_1,
10    CASE WHEN <cond 1> THEN ...
11         WHEN <cond 2> THEN ...
12         ELSE ...
13    END AS column_2,
14    ...
15 FROM cte1
16 ...

```

Listing 3: Using `CASE` to differentiate between the same branches as Listing 2 in every column

In practice, although a `CASE` statement is necessary for every column, there are some opportunities to reduce code duplication in this particular query. These will be discussed in some more detail in Section 2.4.

2.3.4. Additional Changes

Except for the subquery with the `LIMIT` shortcut, the overall structure of the original PostgreSQL query works just the same in DuckDB. However, there are some operations and functions that work differently or have different names in DuckDB compared to PostgreSQL. Therefore, several additional changes had to be applied that are elaborated on in this subchapter.

Converting ASCII character codes to a string: While it is not strictly necessary for obtaining the greyscale image, the subquery that decodes relevant control information from the beginning of the GIF data stream can be used to decode all the header fields. This includes the ASCII character codes for the GIF version that can be found in the first six bytes of any GIF data stream. PostgreSQL's string function `CONVERT_FROM` accepts a value of type `BYTEA` and converts it to a text using a specified encoding, like UTF-8 or in this case LATIN1. DuckDB supports the `BLOB` function `DECODE`, which converts a binary object to type `VARCHAR`. To convert the string of hexadecimal digits to `BLOB`, every pair of digits must be prefixed with the string literal `'\x'` before casting the `VARCHAR` to `BLOB`. This is handled by a small subquery which extracts the bytes one at a time, prefixes the string literal, and aggregates all result string in the correct order to a single `VARCHAR`.

Array-to-element concatenation: PostgreSQL's `||` operator not only concatenates arrays, but can also be used to concatenate a single new element to an array. DuckDB's `||` operator on the other hand can only concatenate arrays or lists. For array-to-element concatenation, the list function `LIST_APPEND(list, element)` is supported. Sometimes, it is required to create an array or list containing a single element. PostgreSQL's syntax for this is `ARRAY[element]`. This also works in DuckDB, but alternative functions, such as `ARRAY_VALUE(element)` or `LIST_VALUE(element)`, provide the same functionality.

Upper bound for array dimension: PostgreSQL supports the array function `ARRAY_UPPER(anyarray, int)`, which returns the maximum value of the array dimension indicated as an integer. Since the GIF decoder query only uses this function for array dimension 1, this is equivalent to DuckDB's `LEN` function, with the exception that `ARRAY_UPPER` returns `NULL` if the dimension is invalid. For dimension 1, this would be the case if the array was empty, *i.e.*, for an array of length 0. However, this function call is wrapped by another PostgreSQL function, `COALESCE`, which returns the first of its arguments that

is not `NULL`. This function is used to return the value 0 if `ARRAY_UPPER` should return `NULL`. Therefore, DuckDB's `LEN` function provides exactly the functionality that is intended for this query.

2.4. Readability Optimizations

In this chapter, the effects of the changes discussed in Section 2.3 to port the query to DuckDB are examined for how they affect the query's readability. Opportunities to optimize the query in regards to its readability are explored and their effectiveness for this particular problem quantified using methods discussed in Section 1.3.

2.4.1. Discussion of the Porting Effects on Readability

The absence of a function named “`get_byte`” or similar makes any access to the input data stream less intuitive to read. The DuckDB alternative is a call of function `SUBSTRING` that also includes the necessary calculations to convert the byte index to a string character index. This conversion is duplicated in every one of these function calls. In the original PostgreSQL query, the `GET_BYTE` function call appears 28 times in total. Since it is such a common function and a base operation for the query as a whole, it has a significant impact on the readability. Defining the `SUBSTRING` function calls as macros resolves this issue, and even provides a more descriptive name for extracting multiple bytes at once compared to the original PostgreSQL query.

PostgreSQL	DuckDB	DuckDB Macros
<code>GET_BYTE(bytes, i)</code>	<code>SUBSTRING(bytes, i * 2 + 1, 2)</code>	<code>get_byte(bytes, i)</code>
<code>SUBSTRING(bytes, i, n)</code>	<code>SUBSTRING(bytes, i * 2 + 1, n * 2)</code>	<code>get_n_bytes(bytes, n)</code>

Table 4: Comparison of extraction operations in PostgreSQL, DuckDB and using macros

Extracting a multi-byte field value remains a complex expression, as shown in Listing 4. Even though the use of a string function means that not as many casting operations are required, the expression lacks meaningful names.

```
1 SELECT (GET_BYTE(buffer, i + 1)::BIT(8) || GET_BYTE(buffer, i)::BIT(8))::BIT(16)::INT PostgreSQL
1 SELECT ('0x' || SUBSTR(bytes, (i + 1) * 2 + 1, 2) || SUBSTR(bytes, i * 2 + 1, 2))::INT DuckDB
```

Listing 4: Comparison of the implementations for obtaining the value of a field spanning over two bytes with indices i and $i + 1$

The macro `get_byte` helps to simplify this expression. However, the whole expression can also be defined as a new macro called `multi_byte_field`. This helps to clarify why two bytes are extracted and concatenated in reverse order. An overview over all defined macros and how often they are used in the query is shown in Section 2.4.3.

As mentioned in Section 2.3, converting a string of hexadecimal digits to a bitstring requires casting to `INTEGER` first (see Listing 5). This substep is hidden by PostgreSQL's `GET_BYTE` function, which already returns an integer value.

```
1 SELECT GET_BYTE(buffer, id)::BIT(8) PostgreSQL
1 SELECT ('0x' || SUBSTR(bytes, id * 2 + 1))::INT::BITSTRING::VARCHAR DuckDB
```

Listing 5: Extracting a byte and getting its binary representation

In order to extract any number of bits from the bitstring, the value is cast to `VARCHAR` to allow the use of the `SUBSTRING` function. This step includes stripping the bits of the leading zeroes that are a side effect of casting from `INT` to `BITSTRING`. In practice, the DuckDB expression would therefore rather look like in Listing 6.

```
1 SELECT SUBSTR(('0x' || SUBSTR(bytes, id * 2 + 1))::INT::BITSTRING::VARCHAR, 33 - n, n) DuckDB
```

Listing 6: Getting a binary representation of an extracted byte without leading zeroes

The advantage of doing this is that once the cropped string of bits is available, all following extraction operations become more intuitive. Rather than having to consider the leading zeroes in every bit index calculation, they are removed in a single expression. As a consequence, when it comes to handling strings of bits, it is the DuckDB implementation that is more straightforward. While PostgreSQL’s query performs shift computations and type castings, bits can be accessed simply using their position index (starting at index 1) or using the `SUBSTRING` function (see Listing 7).

<pre>1 SELECT (flags << i)::BIT(1) = b'1', PostgreSQL 2 (flags << j)::BIT(m)</pre>	<pre>1 SELECT flags[i+1] = 1, DuckDB 2 SUBSTR(flags, j+1, m)</pre>
---	---

Listing 7: Extracting bits from a string of bits

Since all these operations occur multiple times throughout the query, defining them as macros provides readability benefits for various parts of the GIF decoder query. Assigning meaningful names to complicated expressions also increases the benefits of value transformations that make accessing them more intuitive in the future. Stripping bitstrings of their leading zeroes as proposed in Listing 6 thus becomes worthwhile, as the implementation can be hidden inside a macro definition.

The DuckDB implementation of the code table as suggested in Section 2.3.2 offers the slight advantage that some type casting performed by the PostgreSQL query are no longer necessary. Instead of casting both the integer keys and the integer list values to `TEXT`, as well as back to their original type after extracting table entries, the `MAP` dictionary can be initialized so that it accepts keys of type `INTEGER` and values of type `INTEGER []`. These differences are shown in Listing 8.

```
1 SELECT (codes->(code::TEXT))::INT[] AS get_values, PostgreSQL
2 codes || HSTORE(current_table_key::TEXT, next_table_chunk::TEXT) AS add_entry
```

```
1 SELECT codes[code][1] AS get_values, DuckDB
2 MAP_CONCAT(codes, MAP {current_table_key: new_table_entry}) AS add_entry
```

Listing 8: Retrieving a value from dictionary “codes” and adding a new new entry

The implemented DuckDB alternative for the conditional branch shortcut discussed in Section 2.3.3 is longer and repeats `CASE` conditions in each of the four column definitions. The PostgreSQL implementation provides three separate versions of the subquery defining these columns, only one of which applies in any iteration of the algorithm. After considering the `WHERE` conditions of each query block, this implementation allows viewing all the parameter updates performed in this case in a single query block, separate from other cases. However, this approach includes some repetitions. Under certain circumstances, some column definitions are the same even across different branches. This is because the `WHERE` conditions that distinguish between the three versions of the subquery do not perfectly match the conditions that need to be considered for each individual column. By making a `CASE` distinction within each column definition, the branches can be tailored to the actual behavior of each parameter according to the algorithm. It also means that the code for updating any of the parameters, e.g., the state of the code table, is all in one place instead of being spread across three query blocks.

Because there are several dependencies between the columns, their definitions were put in a nested `WITH` clause. For the readability optimization, the `CASE` statements of the four columns are instead each moved into their own subquery, which are then all connected by `LATERAL` joins. This approach was favored over nesting them, because it makes reading the code more straightforward.

```

1  CROSS JOIN LATERAL
2      (
3      SELECT CASE WHEN <col1_cond 1> THEN ...
4              WHEN <col1_cond 2> THEN ...
5              ELSE ...
6      END AS column_1
7      ) c1
8  CROSS JOIN LATERAL
9      (
10     SELECT CASE WHEN <col2_cond 1> THEN ...
11             WHEN <col2_cond 2> THEN ...
12             ELSE ...
13     END AS column_2
14     ) c2
15 ...

```

Listing 9: Duplication-free differentiation between all branches of the algorithm

Even column definitions that do not depend on each other had to be moved in a separate subquery, because they could not be put into a the same `SELECT` clause without DuckDB reporting an error that the column was not found in the `FROM` clause.

2.4.2. Documentation of Changes for Improved Readability

In this chapter, it will be explored how the methods described in Section 1.3 can be applied to the GIF decoder query to improve its readability.

Defining macros: As shown in Section 2.4.1, many of the base operations for extracting or converting data discussed in Section 2.3.1 lack meaningful names and reveal conversion steps that are repeated every time these operations are used. The readability of expressions that use these operations greatly benefits from defining them as macros. Their definitions are shown in section Section 2.5.1 and reflect the operations presented in Table 2 of Section 2.3.1. The effects of defining these macros on the query, alongside the effects of the other readability optimizations, are discusses in Section 2.4.3.

Adjusting the Common Table Expressions: Like in the previous chapter, a `WITH` clause is used to break this problem into multiple subqueries, also referred to as Common Table Expressions (CTEs). The advantages this holds for the readability of the query have been discussed in Section 1.3. The number of CTEs and some of their names have been adjusted to better reflect the steps required to solve the problem as they are listed in Section 2.5. This includes moving some parts of the code to different CTEs, so that each one only serves one greater prupose and related information is located in the same CTE. For the GIF decoder query, this includes:

- Moving the calculation of the offset to the image data from CTE `image_header` into the (existing, but renamed) CTE `image_data_offset`
- Moving the extraction of the initial LZW code size from CTE `image_data` to the (existing, but renamed) CTE `lzw_params`

Furthermore, the names of the CTEs and their columns are chosen so that they match the terminology used in the GIF specification [12] and the tutorial [9] the query is based on. This is meant to make it easier to associate parts of the code with the concepts described there which they implement. A table with all the CTE and column name changes from the original PostgreSQL query can be found at the end of this subchapter in Table 7.

Query simplifications:

Some simplifications that were applied to the query in order to improve its readability include:

- Removing any joins that are not strictly necessary.
- If possible, move the code of single-purpose **LATERAL** subqueries into the same subquery that defines similar or related values.
- Distribute the column definitions in **LATERAL** subqueries so that all values that are required for the same purpose or subgoal are in the same subquery.
- Instead of a subquery nested into a **FROM** clause that selects all values of a CTE in order to rename some of its columns, assign the proper column names in the CTE's definition and make it a part of the **FROM** clause using a **CROSS JOIN**. This does not apply to columns whose purpose or significance changes within a recursive CTE from one iteration to the next, as reflecting these changes by a name change is justified.

2.4.3. Conclusion

Table 5 compares the different versions of the GIF decoder query with regards to the readability metrics discussed in Section 1.3.

Metric	PostgreSQL	DuckDB (initial)	DuckDB (readability)
Number of CTEs	12	15	12
Total number of query blocks	47	47	43
Max. depth of nesting	4	2	2
Number of query blocks at depth > 1	9	5	2

Table 5: Metric values for different versions of the GIF decoder query

The number of deeply nested query blocks could be decreased, without increasing the total number of CTEs. There are only two exceptions where query blocks are nested deeper than depth 1, and any nesting deeper than depth 2 could be avoided altogether. Furthermore, the query is a little more compact, as the total number of query blocks is slightly decreased.

The readability aspects discussed so far relate to the size and structure of the query. Another important consideration is the length and complexity of the code within the query's clauses. This is greatly impacted by the defined macros. The extend to which the defined macros impact the code is presented in Table 6.

Macro	Operations	Occurences
get_byte(bytes, id)	3	17
get_byte_val(bytes, id)	5	9
save_get_byte(bytes, id)	8	2
get_n_bytes(bytes, start, n)	4	2
get_n_bits(hexdigits, n)	6	3
multi_byte_field(hexdigits, byte_offset)	10	6

Table 6: Macros that appear in the GIF decoder query, how many operators, function calls or SQL clauses they hide in their definition and how often they appear in the code.

As shown in Table 6, the defined macros are used for a total of 39 times throughout the GIF decoder query. Each one hides up to 10 operations in their definition, which would otherwise be repeated in the code of the query. The total number of hidden operations through the use of macros adds up to 198. Of these hidden operations, 162 would be a mere repetition of similar expressions that already

appear elsewhere in the query. This is the number of repetitions that could be avoided by using macros to assign meaningful names to certain expression. Meanwhile, all the macro definitions combined only read as 27 operations or references to other macros. Overall, this shows that macros can contribute a great deal to simplifying the query, as several complicated expressions do not have to be repeated over and over again.

Type	original name	renamed to	Comment
CTE	bin	input	
column	buffer	hexdigits	
CTE	header	screen_descriptor	
column	version	gif_version	
subquery	q	packed_field	
subquery	h	descriptor_fields	
column	gct	gct_flag	
column	depth	color_res	
CTE	blocks		
column	current	block_offset	
column	intro	introducer	
subquery	b	prev	
field	previous	previous_offset	
field	previous_intro	previous_introducer	
CTE	image_header	image_descriptor	
column	-	descr_size	name for magic number "10"
column	left	left_pos	
column	top	top_pos	
subquery	q	descriptor_fields	
subquery	l	ls	
subquery	l2	-	→ CTE image_data_offset
CTE	image_blocks		
subquery	l	s	
CTE	lzw_data	lzw_params	
column	code_size	initial_code_size	
column	compressed	code_stream	
subquery	i	img_data	
CTE	lzw_bits	decompression_steps	
column	current_code_size	code_size	
column	output_chunk	color_pattern	
column	codes	code_table	
column	next_index	pixel_index	
field	previous_chunk	previous_colors	
field	next_table_chunk	new_table_entry	
field	next_code_size	new_code_size	
CTE	indices	color_indices	
column	idx	color_idx	
column	rn	pixel_idx	
CTE	picture	grayscale_image	

Table 7: Documentation of renamed CTEs, subqueries and columns

2.5. DuckDB GIF Decoder Query Explained

In this section, the DuckDB code for the GIF decoder query is presented. The structure of this query largely corresponds to the SQL implementation proposed by Quassnoi [8], with the changes presented in Section 2.3 to port it to DuckDB and Section 2.4 to optimize its readability. The goal in this section is to break down the program into subproblems and explain the subqueries that implement them. To this end, Section 2.5.1 shows how the input data is made available to the GIF decoder query, as well as which operations it uses to process the data. The other subsections document the 4 steps that make up the GIF decoder query:

1. Calculating the offset to the compressed image data (Section 2.5.2)
2. Extracting the compressed image data from the input data stream (Section 2.5.3)
3. Performing the LZW decompression algorithm on the image data (Section 2.5.4)
4. Rendering the grayscale image (Section 2.5.5)

2.5.1. Preparing the Input Data

The GIF decoder query operates on the hex-encoded file contents of a GIF file. Assuming there is a GIF file named “input_image.gif”, the hex-encoded content of this file can be obtained using the following shell command [8]:

```
1 cat input_file.gif | od -A n -vt x1 | tr -d '\n '
```

shell

This command returns a string of hexadecimal digits, which will be the input data for the query. In order to make this data available to the GIF decoder query, it is inserted into a table named “input”, which consists of a single column of type `TEXT`.

```
1 CREATE TABLE input (  
2     data TEXT  
3 );
```

DuckDB

In order to process the string of hexadecimal digits, the query has to be able to extract bytes and interpret them as decimal values or convert them to binary digits. To this end, a handful of operations on hex-encoded data are defined as scalar macros. Doing so assigns names to common expressions in the query, which can then be used in the code in place of their implementation. This helps with readability, as these operations will appear over and over again throughout the query. To distinguish them from actual functions supported by DuckDB, their names are written in lowercase.

The digits are stored in a field of type `VARCHAR`. Extraction operations are therefore realized by using the `SUBSTRING` function. Since two hexadecimal digits make up one byte of data, two characters need to be extracted for every byte that should be extracted from the string. We also define macros for less intuitive type conversion operations and a macro that obtains the decimal value of a field that spans two bytes. This is a special case, since GIF’s multi-byte fields are stored in the little endian format [9]. As a consequence, the order of the two bytes has to be reversed before concatenating their digits.

```

1  -- Convert hexadecimal digits to a decimal number:
2  CREATE MACRO to_decimal(hexdigits) AS ('0x' || hexdigits)::INT;
3  -- Get one byte (two hex-digits) at byte index "id":
4  CREATE MACRO get_byte(hexdigits, id) AS SUBSTRING(hexdigits, id * 2 + 1, 2);
5  CREATE MACRO get_byte_val(hexdigits, id) AS to_decimal(get_byte(hexdigits, id));
6  CREATE MACRO save_get_byte(hexdigits, id) AS
7      CASE WHEN id * 2 + 1 < LEN(hexdigits) THEN get_byte(hexdigits, id) ELSE '00' END;
8  -- Get n bytes starting from byte index "id":
9  CREATE MACRO get_n_bytes(hexdigits, id, n) AS SUBSTRING(hexdigits, id * 2 + 1, n * 2);
10 -- Get the n most significant bits from a string of hexadecimal digits:
11 CREATE MACRO get_n_bits(hexdigits, n) AS
12     SUBSTRING(to_decimal(hexdigits)::BITSTRING::VARCHAR, 33 - n, n);
13 -- Get the value of a field spanning two bytes (little endian format):
14 CREATE MACRO multi_byte_field(hexdigits, id) AS
15     to_decimal(get_byte(hexdigits, id + 1) || get_byte(hexdigits, id));

```

Listing 11: Macro definitions for operations on hex-encoded data

The GIF decoder query uses a `WITH` clause to define a row of Common Table Expressions (CTEs) that serve to solve subproblems of the task. Because some of these CTEs will be recursive, the query starts with the `WITH` keyword followed by the optional modifier `RECURSIVE`. The first CTE is only used to transfer the input data from table “input” into a temporary table. This showcases where the input data is coming from, and provides an opportunity to assign a new name and data type to the column that is used to store it. In this case, the text data type `VARCHAR` should remain unchanged, because that is the type the macros were designed to work with. Other than renaming the column to “hexdigits” to describe the format of the data better, this CTE is therefore mostly redundant.

```

1  WITH RECURSIVE
2      input (hexdigits) AS (
3      SELECT data AS hexdigits
4      FROM   input
5      ),

```

Listing 12: Beginning of the GIF decoder query

2.5.2. Calculating the Offset to the Compressed Image Data

The goal of the GIF decoder query is to decompress the image data that is encoded in the GIF file contents in order to be able to print it as a grayscale image in the terminal. As the processing of the input string begins, the first objective is therefore to determine the byte offset at which the compressed image data can be found. This requires applying knowledge about the structure GIF files, which is documented in [12].

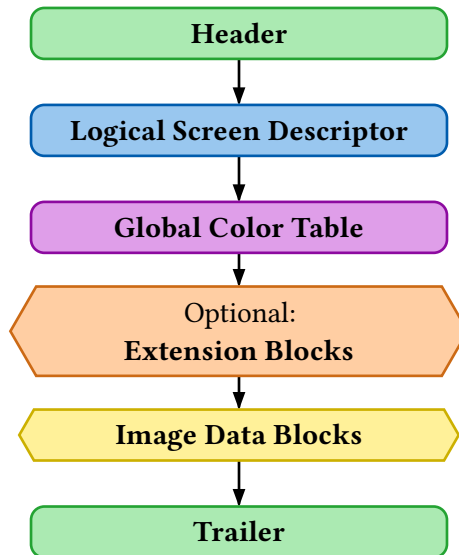


Figure 2: Structure of a GIF data stream for a single image

The image data is located after the Global Color Table, which stores the RGB values for all the colors in the image [12]. Its length thus depends on the image, while the length of the Header and the Screen Descriptor are fixed as 6 and 7 bytes, respectively. Extracting the fields of the Screen Descriptor provides some control data that is used to calculate the length of the Global Color Table. The descriptor immediately follows the Header Block, which takes up the first 6 bytes of data and encodes the GIF version as ASCII character codes. Decoding them returns either the string “GIF87a” or “GIF89a”.

Bytes	Description
0-5	GIF signature and version
6-7	Screen Width in pixels
8-9	Screen Height in pixels
10	Packed Field: GCT flag, Color Resolution, Sort Flag, GCT size
11	Background Color Index (BCI)
12	Pixel Aspect Ratio

Table 8: Byte indices of the Header and Screen Descriptor fields according to [16]

```

6  screen_descriptor (gif_version, width, height, gct_flag, res, gct_size, bci, ratio) AS
7  (
8  -- Get the values of the Header and Logical Screen Descriptor fields.
9  SELECT  DECODE(header.char_codes) AS gif_version,
10         descriptor_fields.*
11 FROM    input AS i
12 CROSS JOIN LATERAL
13     (
14     -- (A) Extract the ASCII character codes for the GIF signature and version (first 6 bytes)
15     SELECT  STRING_AGG('\x' || get_byte(i.hexdigits, id), '' ORDER BY id)::BLOB AS char_codes
16     FROM    GENERATE_SERIES(0, 5) AS bytes(id)
17     ) header
18 CROSS JOIN LATERAL
19     (
20     -- (B) Convert byte 10 (the Packed Field) to a string of bits of length 8.
21     SELECT  get_n_bits(get_byte(i.hexdigits, 10), 8) AS flags
22     ) packed_field
23 CROSS JOIN LATERAL
24     (
25     -- (C) Extract the values of the Logical Screen Descriptor fields.
26     SELECT  multi_byte_field(i.hexdigits, 6) AS width,
27            multi_byte_field(i.hexdigits, 8) AS height,
28            flags[1] = 1 AS gct_flag,
29            SUBSTRING(flags, 2, 3)::BITSTRING::INT + 1 AS res,
30            2 << SUBSTRING(flags, 6, 3)::BITSTRING::INT AS gct_size,
31            get_byte_val(i.hexdigits, 11) AS bci,
32            get_byte_val(i.hexdigits, 12) AS ratio
33     ) descriptor_fields
34 ),

```

DuckDB

Listing 13: Extracting the fields of the Header and Logical Screen Descriptor:

- (A) Before using the `DECODE` function in the `SELECT` clause to convert the ASCII character codes to a string, they have to be converted to type `BLOB`. This requires prefixing each byte, aka each pair of hexadecimal digits, with the string literal `'\x'` before performing the typecasting. Then, aggregate function `STRING_AGG` concatenates them in their original order to a single string of character codes.
- (B) The “Packed Field” of the Logical Screen Descriptor contains multiple fields, which, in this case, are only 1 or 3 bits long, packed into a single byte. In order to access them, the byte is first converted to a string of 8 bits.
- (C) The last three bits of the Packed Field serve to calculate the size of the Global Color Table using the formula $2^{\text{value} + 1}$. This calculation is realized as a left shift operation.

To locate the image data in the GIF data stream, only the size of the Global Color Table is relevant. The value `gct_size` indicates the number of colors in the Global Color Table. To calculate its length in bytes, this number has to be multiplied by 3, since the table stores a red, green and blue value for each color. Adding this length onto the combined length of the Header and Logical Image Descriptor returns the index of the first byte after the Global Color Table.

```

34 first_block_offset AS
35 (
36 SELECT 13 + 3 * gct_size AS first_block_offset
37 ),

```

DuckDB

Listing 14: Calculating the offset to the blocks past the Global Color Table

The compressed image data always begins with an Image Descriptor. Right before the Image Descriptor, there may be a row of Extension Blocks that the query has to skip in order to reach the actual image data. [12] This is achieved via a recursive Common Table Expression, which consists of a base case and a recursive step connected by `UNION ALL` [3]. It selects the byte offset of every block after the Global Color Table until it reaches a block that is not an extension block. Extension blocks can be recognized by their first byte, which is also called the “Extension Introducer” and has the value 0x21 [16]. The last row added to the result set contains the byte offset of the Image Descriptor.

```

37 blocks (block_offset, introducer) AS
38 (
39 -- Base Case: The first block after the Global Color Table
40 SELECT f.first_block_offset AS block_offset,
41        get_byte(i.hexdigits, f.first_block_offset) AS introducer
42 FROM   input AS i
43 CROSS JOIN
44        first_block_offset AS f
45 UNION ALL
46 -- Recursive Step: All following blocks that are a successor of a Graphics Control Extension block
47 SELECT next.block_offset,
48        get_byte(i.hexdigits, next.block_offset) AS introducer
49 FROM   input AS i
50 CROSS JOIN LATERAL
51        (
52 -- (A) Get the column values and block size of the previous iteration.
53 SELECT block_offset AS previous_offset,
54        introducer AS previous_introducer,
55        get_byte_val(i.hexdigits, previous_offset + 2) AS block_size
56 FROM   blocks
57 ) prev
58 CROSS JOIN LATERAL
59        (
60 -- (B) Calculate the offset of the next block.
61 SELECT previous_offset + block_size + 4 AS block_offset
62 ) next
63 -- Continue the recursion as long as the previous block begins with the extension introducer '21'.
64 WHERE previous_introducer = '21'
65 ),

```

DuckDB

Listing 15: Byte offsets of all blocks after the Global Color Table up until the first image data block

- (A) The recursive call is inside a nested subquery named “prev”, which renames the columns of the previous iteration and gets the size of the previous extension block. The size is indicated by the second byte after the Extension Introducer [16].
- (B) The indicated block size excludes the first three and the last byte of the Extension Block [16]. Hence, the block offset has to be incremented by 4.

The beginning of the Image Descriptor is marked by the Image Separator, a byte of value 0x2c [16]. To obtain the byte offset of the Image Descriptor, the result row of the previous recursive CTE where the introducer is equal to these two digits needs to be selected. This is equivalent to the last offset that was added to the result.

```

65 image_offset AS
66 (
67 SELECT block_offset AS image_offset
68 FROM blocks
69 WHERE introducer = '2c'
70 ),

```

Listing 16: Byte offset of the Image Descriptor, the beginning of the image data

The bytes that come immediately after the Image Separator contain the values of the other Image Descriptor fields. They provide information that will be needed to work with the table based image data [16]. They are extracted next, with multi-byte and Packed Fields being handled just like before.

```

70 image_descriptor (descr_size, image_offset, left_pos, top_pos, width, height, has_lct,
71                  interlace, sort, lct_size) AS (
72 SELECT 10 AS descr_size,
73        io.image_offset,
74        descriptor_fields.*,
75        ls.lct_size
76 FROM   image_offset AS io
77 CROSS JOIN LATERAL
78        input AS i
79 CROSS JOIN LATERAL
80        (
81          -- Convert the Packed Field to a string of bits of length 8.
82          SELECT get_n_bits(get_byte(i.hexdigits, io.image_offset + 9), 8) AS flags
83        ) packed_field
84 CROSS JOIN LATERAL
85        (
86          -- Extract the values of the Logical Screen Descriptor fields.
87          SELECT multi_byte_field(i.hexdigits, io.image_offset + 1) AS left_pos,
88                 multi_byte_field(i.hexdigits, io.image_offset + 3) AS top_pos,
89                 multi_byte_field(i.hexdigits, io.image_offset + 5) AS width,
90                 multi_byte_field(i.hexdigits, io.image_offset + 7) AS height,
91                 flags[1] = 1 AS has_lct,
92                 flags[2] = 1 AS interlace,
93                 flags[3] = 1 AS sort
94        ) descriptor_fields
95 CROSS JOIN LATERAL
96        (
97          -- (A) Calculate the size of the Local Color Table, if there is one.
98          SELECT CASE WHEN has_lct THEN 2 << SUBSTRING(flags, 6, 3)::BITSTRING::INT
99                 ELSE 0
100                END AS lct_size
101        ) ls
102 ),

```

Listing 17: Extracting the fields of the Image Descriptor

- (A) An optional Local Color Table may follow the Image Descriptor. If that is the case, *i.e.*, if flag `has_lct` is set, its size is calculated using formula $2^{\text{value}+1}$. This is once again realized as a left shift operation. Otherwise, the length of the Local Color Table is set to 0.

The byte offset to the image data is obtained by adding the size of the Image Descriptor, which is fixed to 10, and the size of the Local Color Table onto the byte offset to the Image Separator. Because the first byte of the image data contains the initial code size for the LZW algorithm, which will be discussed later, the value is incremented by 1. The result is the byte offset to the first block of compressed image data.

```
103 image_data_offset (first_block_offset) AS
104 (
105 SELECT image_offset + descr_size + lct_size + 1 AS first_block_offset
106 FROM image_descriptor
107 ),
```

Listing 18: Byte offset to the compressed image data

2.5.3. Extracting the Compressed Image Data

Another recursive CTE is used to iterate over all image data blocks and extract their data values. The first byte of each sub-block indicates how many bytes of data the block contains. The recursion stops once it encounters the Block Terminator, indicated by a block size field with value 0 [9].

```

107 image_blocks (block_offset, block_size, block_data) AS DuckDB
108 (
109 -- Base Case: Extract the data of the first image data sub-block.
110 SELECT img.first_block_offset AS block_offset,
111        s.block_size,
112        get_n_bytes(i.hexdigits, img.first_block_offset + 1, s.block_size) AS block_data
113 FROM   input AS i
114 CROSS JOIN
115        image_data_offset AS img
116 CROSS JOIN LATERAL
117        (
118        SELECT get_byte_val(i.hexdigits, img.first_block_offset) AS block_size
119        ) s
120 UNION ALL
121 -- Recursive Step: Extract the data of all following image data sub-blocks.
122 SELECT no.new_offset AS block_offset,
123        nbs.new_block_size AS block_size,
124        get_n_bytes(i.hexdigits, no.new_offset + 1, nbs.new_block_size) AS block_data
125 FROM   image_blocks -- recursive call
126 CROSS JOIN LATERAL
127        input AS i
128 CROSS JOIN LATERAL
129        (
130        SELECT block_offset + block_size + 1 AS new_offset
131        ) no
132 CROSS JOIN LATERAL
133        (
134        SELECT get_byte_val(i.hexdigits, no.new_offset) AS new_block_size
135        ) nbs
136 -- Stop the recursion when the size of next sub-block is 0.
137 WHERE  nbs.new_block_size > 0
138 ),

```

Listing 19: Extracting the data from all image blocks

As a second step, the data from all the sub-blocks is aggregated into a single string using the aggregate function `STRING_AGG`. The contents of this string are also referred to as the “code stream” [9], since it is a sequence of LZW codes that will be decoded in the decompression algorithm. Before that, the values of a few parameters for that algorithm are defined. This includes the code size, which is encoded in the byte before the first block of image data, as well as the values of the “clear code” and the “end of information code”, which are $2^{\text{code_size}}$ and $2^{\text{code_size}} + 1$. The actual first value for the code size that is used in the algorithm later is the value of the code size field increased by one [9].

```

138 lzw_params (initial_code_size, clear_code, eof_code, code_stream) AS DuckDB
139 (
140 SELECT cs.code_size + 1 AS initial_code_size,
141        (1 << cs.code_size) AS clear_code,
142        (1 << cs.code_size) + 1 AS eof_code,
143        img_data.code_stream
144 FROM (
145     -- Aggregate the data from all image blocks into a single string.
146     SELECT STRING_AGG(block_data, '' ORDER BY block_offset) AS code_stream
147     FROM image_blocks
148    ) img_data
149 CROSS JOIN
150 (
151     -- Get the initial code size from the first byte of image data.
152     SELECT to_decimal(get_byte(i.hexdigits, img.first_block_offset - 1)) AS code_size
153     FROM input AS i
154     CROSS JOIN
155          image_data_offset AS img
156    ) cs
157 ),

```

Listing 20: Setting the values of the parameters for the LZW decompression algorithm

2.5.4. LZW Decompression Algorithm

The purpose of the next CTE is to perform the LZW decompression algorithm adapted for GIF. Its job is to iterate over the previously extracted image data and decompress it. The compressed image data consists of a sequence of LZW codes, each representing a list of color indices, *i.e.*, a color pattern. Once all these patterns are decoded from the image data, assigning them to the pixels of the graphic in the order in which their codes appear will produce the image.

As a documentation of the most complex part of the GIF decoder query, this subchapter presents:

- an overview of the algorithm’s goals and parameters
- the base case of the recursive CTE, which initializes the parameters
- how LZW codes are extracted from the sequence
- how the query determines the color patterns represented by the LZW codes
- the SQL code for updating the algorithm parameters for the next iteration

The main objective of the CTE is to recursively iterate over the sequence of codes and determine for each one what color pattern it represents. This is achieved using a code table that maps LZW codes to a list of color indices. However, this code table is not included in the GIF file. Instead, the decoder builds the table while performing the decompression algorithm. Initially, the code table only holds the trivial entries for color patterns of length 1, plus a “clear code” to reset the table and an “end of information code”. Therefore, for a graphic with n different colors and thus color indices ranking from 0 to $(n - 1)$, the code table initially has $(n + 2)$ entries (see Table 9). Each code from 0 to $(n - 1)$ is mapped to a single-element list containing the color index that is equal to the value of code. The code with value n is the “clear code”, the code with value $(n + 1)$ is the “end of information code”.

Code value	List of color indices
0	[0]
1	[1]
...	...
n-1	[n-1]
n	<clear code>
n+1	<end of information code>

Table 9: Initial state of the code table

The purpose of the code table is to store the mapping between LZW codes and color patterns. However, determining the trivial color patterns of the initial code table entries is rather simple, as their values are identical to their code. Therefore, it is not necessary to look them up in the table. This also means that the initial entries do not necessarily have to be stored in the table in the first place (an approach suggested by [19]). Instead, the query can directly return the single-element list or reset the table if the “clear code” is encountered. Only the other codes, which are added to the code table during the execution of the decompression algorithm, are actually inserted into a *MAP* dictionary. This dictionary can therefore be initialized as empty at the beginning of the algorithm and resetting the code table to its initial state is as simple as replacing it by an empty *MAP*.

In addition to the current state of the code table, a few more parameters have to be passed on from each iteration to the next. In order to ensure that the colors of each decoded pattern eventually get assigned to the right pixels, the CTE stores a `pixel_index` value. This value indicates the index of the pixel to which the first color in the pattern should be assigned. An overview of all the columns and what they represent is shown in Table 10.

Column Name	Description
<code>code</code>	next LZW code that was extracted from the sequence
<code>color_pattern</code>	list of color indices represented by code
<code>pixel_index</code>	index of the pixel that will be assigned the first color in <code>color_pattern</code>
<code>code_table</code>	current state of the code table
<code>code_size</code>	current number of bits that make up a LZW code
<code>next_bit_offset</code>	bit offset from the beginning of the code sequence to the code that should be extracted in the next iteration
<code>next_table_key</code>	value of the key that will be added next to the code table

Table 10: Overview of the columns in the recursive CTE `decompression_steps`, based off the SQL implementation presented by Quassnoi [19]

The base case of the recursive CTE assigns each parameter for the algorithm its initial value. The initial code size has already been determined previously and can therefore be selected from CTE `lzw_params`. Any LZW code stream from a GIF file always begins with the clear code, which serves to reset the table. No color pattern is associated with this code, so the first value of the column `color_pattern` is just an empty array. The code table is initialized as empty, all remaining parameters are initialized as 0.

```

157 decompression_steps (current_code_size, code, color_pattern, next_bit_offset,
158                       code_table, next_table_key, pixel_index) AS (
159   -- Base Case: Initialize algorithm parameters.
160   SELECT  initial_code_size AS current_code_size,
161           clear_code AS code,
162           ARRAY[] AS color_pattern,
163           0 AS next_bit_offset,
164           MAP()::MAP(INTEGER, INTEGER[]) AS code_table,
165           0 AS next_table_key,
166           0 AS pixel_index
167   FROM    lzw_params
168   UNION ALL
169   ...

```

Listing 21: Base case of the recursive decompression CTE

The recursive step implements how these parameters are updated in every iteration of the algorithm. The main goals of each iteration of the algorithm are:

- Extracting the next code from the code sequence
- Determining the color pattern that the code represents
- Adding one entry to the code table

The implementation of the first part of the recursive step is shown in Listing 22. Its `SELECT` clause reflects how the parameters are updated. As this is a recursive query, it has to reference itself in the `FROM` clause. This recursive reference occurs in a subquery that assigns new names to the columns of the previous iteration, helping to distinguish their values from those of the current iteration.

```

169   ...
170   -- (A) Recursive Step: Perform one iteration of the decompression algorithm.
171   SELECT  cs.new_code_size AS code_size,
172           next.code AS code,
173           decode.color_pattern AS color_pattern,
174           p.bit_offset + p.current_code_size AS next_bit_offset,
175           ct.new_code_table AS code_table,
176           nk.next_table_key,
177           p.previous_idx + p.output_length AS pixel_index
178   FROM    (
179       -- Rename columns from previous iteration.
180       SELECT  current_code_size,
181               code AS previous_code,
182               color_pattern AS previous_colors,
183               next_bit_offset AS bit_offset,
184               next_bit_offset // 8 AS byte_offset,
185               code_table,
186               next_table_key AS current_table_key,
187               pixel_index AS previous_idx
188           FROM    decompression_steps
189           ) AS p
190   UNION ALL
191   ...

```

Listing 22: Recursive step of the decompression algorithm (Part 1)

- (A) Because this CTE is split across multiple Listings, the `SELECT` clause references columns whose definitions are shown later.

The next step is to extract the next code from the code stream, which is obtained from CTE `lzw_data`. Its location in the code stream is calculated using the byte and bit offsets that are provided by the values from the previous iteration. Before the bits of the code can be obtained, a three bytes long excerpt of the code stream that is guaranteed to contain them is extracted. They then have to be concatenated in reversed order [8] before the `SUBSTRING` function can be used to extract the bits of the code. This is relevant for codes whose bits are spread across more than one byte. Figure 3 illustrates this process of extracting codes from the bytes of the code sequence.

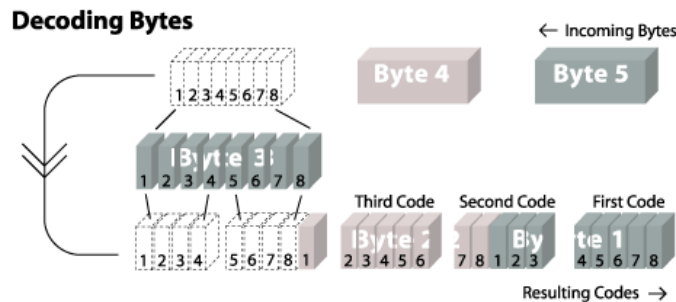


Figure 3: Extracting codes from the sequence for an exemplary codes size of 5, as illustrated by Mike Flickinger in [9]

```

190 ...
191 CROSS JOIN
192     lzw_params AS lp
193 CROSS JOIN LATERAL
194     (
195         -- (A) Get the next 3 bytes of the code stream (add leading zeroes if necessary).
196         SELECT get_n_bits(
197             save_get_byte(lp.code_stream, p.byte_offset + 2) ||
198             save_get_byte(lp.code_stream, p.byte_offset + 1) ||
199             get_byte(lp.code_stream, p.byte_offset)
200             , 24
201         ) AS cut
202     ) AS c
203 CROSS JOIN LATERAL
204     (
205         -- (B) Determine the value of the next code.
206         SELECT ('0b' ||
207             SUBSTRING(c.cut, 25 - (p.current_code_size + p.bit_offset % 8), p.current_code_size)
208             )::INT AS code
209     ) AS next
210 ...

```

Listing 23: Recursive step of the decompression algorithm (Part 2)

- (B) Since the maximum code size is 12 (the number of entries in the code table is limited to $0xFFFF = 4095$) and `bit_offset % 8` is at most 7, code stream excerpt “cut” of length 24 (A) is guaranteed to be long enough to hold the next code.

With the next code at hand, the goal is to determine which color pattern it represents. The first step is to check whether there already exists an entry for this code in the code table. If that is the case, said entry contains the list of color indices. Otherwise, the list can be built from the color pattern decoded in the previous iteration. This new pattern will later be added as a new entry to the code table, so that future iterations will only have to look it up.

```

209 ...
210 CROSS JOIN LATERAL
211     (
212     -- (A) Predicate: Does a code entry already exist in the code table?
213     SELECT (next.code < lp.clear_code) OR (next.code IN p.code_table) AS code_in_table
214     ) AS pr
215 CROSS JOIN LATERAL
216     (
217     -- (B) Decompression: Determine the color pattern represented by the code.
218     SELECT CASE WHEN next.code = lp.clear_code THEN []::INT[]
219             WHEN p.previous_code = lp.clear_code OR pr.code_in_table
220             THEN COALESCE(p.code_table[next.code][1], LIST_VALUE(next.code))
221             ELSE (
222             -- Derive color pattern from the table entry for the previous code
223             SELECT LIST_APPEND(p.previous_colors, p.previous_colors[1])
224             WHERE next.code <> lp.eof_code
225             )
226     END AS color_pattern
227     ) AS decoded
228 ...

```

Listing 24: Recursive step of the decompression algorithm (Part 3)

- (A) Since `code_table` does only store the non-trivial entries, deciding whether an entry exists in the code table requires an additional predicate that is also true for codes that are smaller than the clear code. For clarity and since this check is required more than once, the value of this predicate is determined in a subquery that can be referenced by the following subquery via a lateral join.
- (B) The list of color indices is either empty, can be found in the code table or is derived from the list for the previous code. When a color pattern should be retrieved from the code table, it can either be accessed in the `MAP` “code_table” or, if it a trivial code pattern of length 1, it is set to a list of length 1 containing the code as its single element. Function `COALESCE` returns its second argument if its first is `NULL`, which occurs when an entry is not actually stored in our table representation.

The next step is to add a new entry to the code table, unless it has already reached its maximum size of 4095 entries. If the table did not already have an entry for the code considered in the current iteration, its color pattern is now added to the table. Otherwise, an entry for a new color pattern created from the current and previous color pattern is added.

Lastly, the value of the next table key is increased by 1 if there was not a table reset, and the code size is increased if the current number of bits is not enough for the next key value. After the final subquery of the CTE’s `FROM` clause, the `WHERE` clause defines the stop condition of the recursion. Once an iteration extracts the end of information code or the bit offset to the code surpasses the length of the code stream, the recursion ends.

```

227 ...
228 CROSS JOIN LATERAL
229     (
230     -- (A) Update the code table.
231     SELECT CASE WHEN next.code = lp.clear_code THEN MAP()::MAP(INTEGER, INTEGER[])
232             WHEN p.previous_code = lp.clear_code OR p.current_table_key = 4095
233             THEN p.code_table
234             ELSE MAP_CONCAT(p.code_table, MAP {p.current_table_key: colors.pattern})
235     END AS new_code_table
236 FROM     (
237     -- (B) Determine the color pattern that will be added to the table.
238     SELECT CASE WHEN pr.code_in_table
239             THEN LIST_APPEND(p.previous_pattern, p.color_pattern[1])
240             ELSE decoded.color_pattern
241     END AS pattern
242     ) AS colors
243     ) ct
244 CROSS JOIN LATERAL
245     (
246     -- (C) Determine the value of the next key that will be added to the table.
247     SELECT CASE WHEN next.code = lp.clear_code THEN lp.eof_code + 1
248             WHEN p.previous_code = lp.clear_code THEN p.current_table_key
249             ELSE LEAST(p.current_table_key + 1, 4095)
250     END AS next_table_key
251     ) nk
252 CROSS JOIN LATERAL
253     (
254     -- (D) Increase the code size if the next table key is too large.
255     SELECT CASE WHEN next.code = lp.clear_code THEN lp.initial_code_size
256             WHEN nk.next_table_key = (1 << p.current_code_size) THEN p.current_code_size + 1
257             ELSE p.current_code_size
258     END AS new_code_size
259     ) cs
260 WHERE p.previous_code IS DISTINCT FROM lp.eof_code AND
261        p.bit_offset < LEN(lp.code_stream) * 4
262 ),

```

Listing 25: Recursive step of the decompression algorithm (Part 4)

- (A) Unless there has been a code table reset or the maximum number of entries has been reached, a new entry is added to the table. The key value for the new entry is already available as `current_table_key`, renamed from column `next_table_key` of the previous iteration.
- (B) If the code of the current iteration already has a table entry, the color pattern for the new table entry is created from the current and previous color pattern. Otherwise, an entry with the current pattern is made.
- (C) Unless there has been a code table reset or the maximum number of entries has been reached, the value of the next key is increased by 1.
- (D) The code size remains unchanged, unless the next table key becomes too large for the current number of bits available for each code, which is when it is increased by 1.

2.5.5. Rendering the Image

Each row of the previous CTE `decompression_steps` contains a list of color indices that represents one fragment of the sequence of color indices that defines the raster image. The goal of the next CTE is to unnest these lists to obtain a result set where each row contains exactly one color index alongside the index of the pixel that should be assigned this color. The value in column `pixel_index` already indicates the pixel index for the first color in the `color_pattern` list of the same row. DuckDB's list function `GENERATE_SUBSCRIPTS` can be used to generate the position indices (starting at index 1) of all list elements in `color_pattern`. With those at hand, the pixel index for each of the color indices in the pattern can be calculated the pixel index of the pattern's first color.

```
260 color_indices (color_idx, pixel_idx) AS DuckDB
261 (
262   SELECT  g.color_idx,
263          d.pixel_index + (g.position - 1) AS pixel_idx
264   FROM    decompression_steps AS d
265   CROSS JOIN LATERAL
266          (
267            SELECT  GENERATE_SUBSCRIPTS(d.color_pattern, 1) AS position,
268                   UNNEST(d.color_pattern) AS color_idx
269          ) g
270 ),
```

Listing 26: Unnesting the color patterns

At this point, the image data has been successfully decompressed. The final goal is now to print it as a grayscale image. To that end, the actual colors behind each color index are retrieved from the Global Color Table, which always starts at the 13th byte in the GIF Data Stream. These color values are then fed into a formula to calculate the brightness, or “luma”, of the color:

$$\text{luma} = 0.2126 * \left(\frac{r}{255}\right)^\gamma + 0.7152 * \left(\frac{g}{255}\right)^\gamma + 0.0722 * \left(\frac{b}{255}\right)^\gamma$$

Based of the brightness of the pixel color, a character or symbol that matches the brightness is chosen to be printed in the terminal. They are selected from a palette ordered from darkest (a whitespace) to brightest (#) by multiplying the luma value to the length of the palette. At the same time, the width of the image that was extracted from the Logical Screen Descriptor is used to convert the pixel indices for each value to a row and column index.

```

270 pixels (column_idx, row_idx, pixel_symbol) AS
271 (
272 -- Choose a symbol to print in the terminal for every pixel.
273 SELECT c.pixel_idx % screen.width AS column_idx,
274        c.pixel_idx // screen.width AS row_idx,
275        g.palette[LEAST(FLOOR(g.luma * g.palette_len) + 1, g.palette_len)::INT] AS pixel_symbol
276 FROM   color_indices AS c
277 CROSS JOIN
278        input AS i
279 CROSS JOIN
280        screen_descriptor AS screen
281 CROSS JOIN LATERAL
282        (
283 -- (A) Get the RGB values for each color index.
284 SELECT get_byte_val(i.hexdigits, 13 + c.color_idx * 3 + 0) AS r,
285        get_byte_val(i.hexdigits, 13 + c.color_idx * 3 + 1) AS g,
286        get_byte_val(i.hexdigits, 13 + c.color_idx * 3 + 2) AS b
287        ) AS val
288 CROSS JOIN LATERAL
289        (
290 -- Calculate the pixel brightness (luma).
291 SELECT '.*:o&8#' AS palette,
292        8 AS palette_len,
293        255 AS max_value,
294        2.2 AS gamma,
295        .2126 AS rw, .7152 AS gw, .0722 AS bw,
296        ((val.r / max_value) ^ gamma) * rw +
297        ((val.g / max_value) ^ gamma) * gw +
298        ((val.b / max_value) ^ gamma) * bw AS luma
299        ) AS g
300 ),

```

DuckDB

Listing 27: Choosing a symbol to print for every pixel based of its brightness

(A) The color table contains three values for each color index: the red, green and blue value. These values can be found in the three bytes starting at index $13 + \text{index} * 3$.

The final CTE aggregates the characters for pixels in the same row to a string. Selecting all rows of this CTE will print the grayscale image in the terminal.

```

300 grayscale_image (image_row) AS
301 (
302 SELECT STRING_AGG(pixel_symbol, '' ORDER BY column_idx) AS image_row
303 FROM   pixels
304 GROUP BY
305        row_idx
306 ORDER BY
307        row_idx
308 )
309 SELECT *
310 FROM   grayscale_image;

```

DuckDB

Listing 28: Aggregating the output strings

2.6. Performance Evaluation

Running the DuckDB code for the GIF decoder query presented in Section 2.5 quickly reveals that it is quite slow. Even with CTE materialization, which will be discussed in a moment, it is significantly slower than the original PostgreSQL query. In this subchapter, the total time obtained from running `EXPLAIN ANALYZE` (see Section 1.4) for different versions of the query is compared. By examining the query graph, possible causes for these performance differences are identified in order to create a new implementation that is optimized for performance.

2.6.1. Effects of Readability Optimizations and CTE Materialization

As discussed in Section 1.4, DuckDB only materializes CTEs under special circumstances [3], while PostgreSQL in most cases automatically evaluates a CTE only once, even if it is referenced multiple times by a sibling CTE [7]. Explicitly materializing CTEs to nullify this difference has an impact on the overall execution time when applied to CTE `lzw_params` and its sources. That materializing this particular CTE is so impactful can be explained by the circumstance that it is referenced in the recursive step of the decompression algorithm (CTE `decompression_steps`). Other than the recursive call to itself, `lzw_params` is the only CTE referenced by it. As it performs a large number of iterations, that explains why CTE materialization is only that effective for the CTE `lzw_params` and its source `image_blocks`, since it means they are only executed a single time instead of once for every iteration.

The initial DuckDB query obtained from porting the PostgreSQL inference query to DuckDB has an additional `WITH` clause nested in its recursive step of the decompression algorithm (see Section 2.3.3). However, materializing its CTEs does not provide any benefits. All CTEs of the outer `WITH` clause, except for the last one that orders the rows of the output, can be materialized without any negative side effects on the performance. With these materializations in place for both the initial DuckDB version and the version optimized for readability, `EXPLAIN ANALYZE` is used to compare their performance to that of the original PostgreSQL query (see Table 11).

	PostgreSQL	DuckDB (initial)	DuckDB (materialized)	DuckDB (readability)
Time	1.68s	281.46s	150.30s	119.79s

Table 11: Comparison of the averaged runtime of different versions of the GIF decoder query (ported to DuckDB, materialized, plus optimized for readability) on the GIF data of a 92x92 test image

Without any CTE materialization, the time of initial adaptation to DuckDB is approximately 281 seconds. With the materialization in place, the total execution time is on average almost halved. However, this is still more than 89 times slower than the original query. That the performance of the materialized version optimized for readability does not perform notably worse than the materialized initial query adaptation was to be expected. Aside from several query simplifications and a different structure of the joins in the CTE for the decompression algorithm, the major difference between them is the use of macros, which should not cause a significant overhead (see Section 1.4). In fact, the restructured joins appear to lead to an improvement of the query's performance, according to the analysis provided by the query graph tool (see Table 12). This analysis includes the runtime of the recursive CTEs in the query, which includes the time of several of the other operators shown in the table.

Phase	Time	Percentage	Phase	Time	Percentage
TOTAL TIME	154.36s	100.00%	TOTAL TIME	123.14s	100.00%
RECURSIVE CTE	153.44s	99.40%	RECURSIVE CTE	122.34s	99.36%
PROJECTION	86.04s	55.74%	PROJECTION	77.05s	62.57%
HASH JOIN	43.89s	28.43%	HASH GROUP BY	6.11s	4.96%
HASH GROUP BY	6.03s	3.90%	RIGHT DELIM JOIN	5.98s	4.86%
RIGHT DELIM JOIN	3.11s	2.02%	HASH JOIN	3.65s	2.96%

Table 12: Runtime analysis provided by DuckDB’s query graph tool for the materialized initial query (left) and the version optimized for readability (right)

The main difference between the two query versions is that the evaluation of the materialized initial DuckDB query needs a lot of time for `HASH JOIN` operations. By looking at the query graph, a single costly hash join in the recursive CTE `decompression_steps` appears to be the main contributor to this issue. The readability optimizations, which included restructuring the `FROM` clause of this CTE’s recursive step from nested `WITH` clauses to `LATERAL` joins, happened to drastically decrease these costs. In exchange, the time for `RIGHT DELIM JOIN` operators has slightly increased. Since the query optimized for readability performs a lot better, it is used as a starting point to discuss further performance optimizations.

2.6.2. Performance Optimizations

The performance aspects discussed so far do not address the issue why the original PostgreSQL version of this program was so much faster. The analysis provided by the query graph tool (Table 12) reveals that more than 99% of the runtime of the query version optimized for readability is caused by recursive CTEs. There are three recursive CTEs in this query: `blocks`, `image_data`, and `decompression_steps`. Looking at the query graph obtained from DuckDB’s query graph tool, the time for CTE `blocks` is indicated as less than 0.0016 seconds, `image_data` with 0.04 seconds and `decompression_steps` takes approximately 122 seconds. Therefore, the performance optimizations discussed in the following focus on CTE `decompression_steps`.

Examining the subgraph for recursive CTE `decompression_steps` quickly reveals that a single `PROJECTION` is responsible for the vast majority of its runtime. With approximately 76 seconds, it also represents the main cause for the 77.05 seconds needed for projections according to the analysis shown in Table 12. This expensive `PROJECTION` relates to the subquery with column `new_code_table`, where the code table representation is updated. Since the other parts of this subquery do not stand out as potentially costly operations, it is likely that it is the method of adding a new code/pattern pair to the dictionary that is so costly in the context of this CTE. As discussed in Section 2.3.2, the method used in the original query to represent the code table was not available for the DuckDB implementation. It is thinkable that the chosen alternative does not perform as well as the original. Therefore, an alternative approach to implementing code table updates is explored in the following, alongside an experiment to see if it has the potential to improve the performance.

Examining the values of the keys stored in the code table reveals that they are integer values. As shown in Listing 25, the value of each key that is added to the table is the previous key value incremented by 1. Under these circumstances, the key values can easily be converted to position indices of an ordered data structure:

Let k be the integer value of the first key that is added to the code table during the decompression algorithm. The next key will then have value $k + 1$, the one after that $k + 2$, and so on. For a data structure where entries are stored in the order in which they were added, subtracting each key value

by $k - 1$ returns their position index (where the first entry added to the structure has index 1). As discussed in Section 2.5.4, the initial code table entries with key values ranking from 0 to the “end of information code” value (`eof_code`) are not actually stored in a dictionary. Instead, the first key value that is added after initialization or a code table reset is the value of the “end of information code” incremented by 1 (see Listing 25). Thus, $k - 1$ corresponds to the value of the “end of information code”. Subtracting it from any key value returns the entry’s positional index. With an easy way to calculate this index at hand, storing key/value pairs is no longer necessary. Instead, the color patterns can be stored in a list. As long as new patterns are appended to the end of the list, so that they appear in the order in which they were added, they can be accessed using the index calculated from the value of the LZW code.

Whether this would bring any benefits was tested by comparing the performance of two recursive queries (Listing 29), one that adds key/value pairs to a dictionary of type `MAP<INT, INT[]>` and one that adds values to a list of type `INTEGER[]`. To model the queries after how the GIF decoder query performs code table updates, they are designed to start with an empty `MAP` or list and add one entry in every recursive step, so that their size increases with every iteration. For a more accurate representation of the code table updates, the length of the lists that are added should increase as well. For the sake of simplification, every iteration of the queries will only add a list of length 1.

```

1 WITH RECURSIVE map_update AS DuckDB
2 (
3   SELECT 0 AS count,
4         MAP()::MAP<INT, INT[]> AS dict
5   UNION ALL
6   SELECT count + 1,
7         MAP_CONCAT(dict, MAP {count: [42]})
8   FROM   map_update
9   WHERE  count < <N>
10  )
11 SELECT dict, CARDINALITY(dict) AS "# entries"
12 FROM   map_update
13 WHERE  count = <N>;

```

```

1 WITH RECURSIVE map_update AS DuckDB
2 (
3   SELECT 0 AS count,
4         []::INT[] AS dict
5   UNION ALL
6   SELECT count + 1,
7         dict || [[42]]
8   FROM   map_update
9   WHERE  count < <N>
10  )
11 SELECT dict, LEN(dict) AS "# entries"
12 FROM   map_update
13 WHERE  count = <N>;

```

Listing 29: Recursive sample queries that adds N elements to a `MAP` (left) or a list (right).

The code table grows to a maximum size of 4095 entries, unless a clear code leads to an early reset. This corresponds to a dictionary representation (which does not include the initial table entries) of size $4095 - eof_code$. However, performance difference between the two methods of storing the table can already be observed long before 4000 entries are reached. Table 13 shows the runtimes obtained from running `EXPLAIN ANALYZE` for the two queries in Listing 29 and for different numbers of iterations N . The values were averaged over 5 consecutive measurements.

	N = 500	N = 1000	N = 1500	N = 2000	N = 2500	N = 3000	N = 3500	N = 4000
Map	0.32s	2.52s	6.65s	15.40s	28.62s	44.69s	61.26s	89.06s
List	0.03s	0.05s	0.07s	0.09s	0.12s	0.15s	0.19s	0.24s

Table 13: Averaged runtimes for recursively adding N entries to a `MAP` compared to an `INTEGER[]` list

The results presented in Table 13 differ vastly for the two queries. For the recursive query that adds entries to a `MAP`, the runtime drastically increases as the number of iterations approaches 4000, almost reaching the mark of one and a half minutes. Meanwhile, the runtime for the recursive query that adds entries to a list only approaches 0.25s. Because of the large discrepancy between these runtimes,

which is also illustrated in Graph 1, it is clear that changing the code table representation to a list has a big potential to improve the performance of the GIF decoder query.

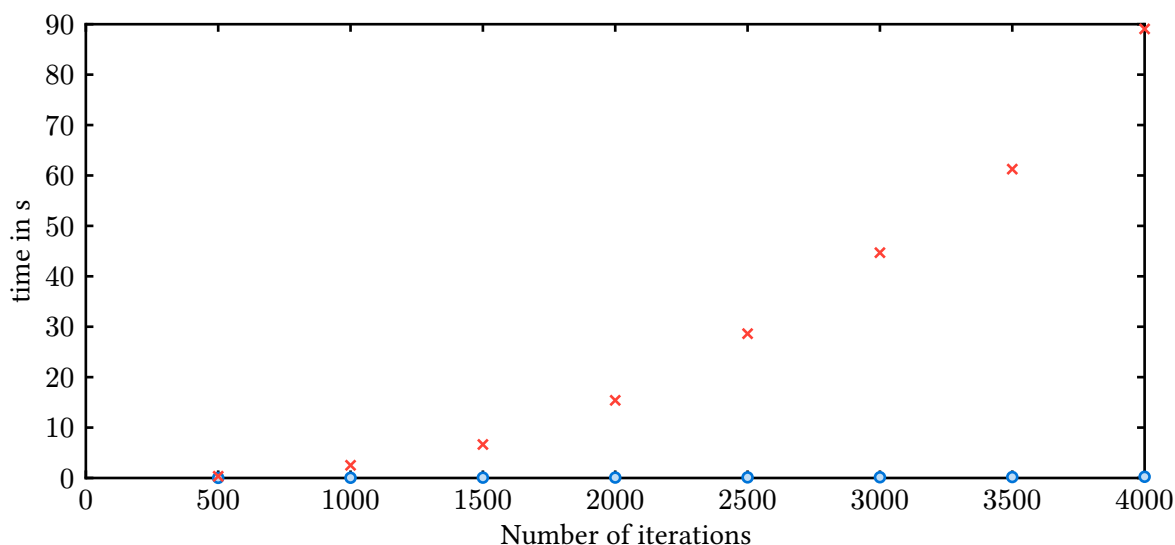


Figure 4: Scatter plot of the values in Table 9 (x for using a MAP, o for using a list)

Changing the code table representation from a MAP to type INTEGER[] is relatively easy, as not many lines of code are affected. Table 14 shows how the expressions involving the code table compare for the two data types.

Operation	MAP code table	INTEGER[] code table
Initialize the table	MAP() :: MAP(INTEGER, INTEGER[])	[] :: INTEGER[]
Add a new pattern	MAP_CONCAT(code_table, MAP {code: pattern})	code_table [pattern]
Retrieve a pattern	code_table[code][1]	code_table[table_index]

Table 14: SQL code for operations involving the code table using a MAP compared to a list

The most notable difference is that retrieving a value from the MAP uses the value of the extracted LZW code as key, while entries of the list representation are accessed using an index, which first has to be calculated from the LZW code. As described before, this is achieved by subtracting the value of the “end of information code” (eof_code). In order to fully match the functionality of retrieving data from a MAP as shown in Table 14, retrieving entries from the list has to return NULL when there is no entry stored in the code table representation for a given code. This occurs in the GIF decoder query for all initial code table entries with color patterns of length 1. As they are not stored in the dictionary, function COALESCE is used to return the list of length 1 when trying to retrieve an entry from the MAP returns NULL.

```
1 COALESCE(code_table[code][1], LIST_VALUE(code))
```

DuckDB

Listing 30: Excerpt from Listing 24: return a color pattern that is either stored in the MAP or of length 1

However, if the code is smaller than the value of the “end of information code”, the value that is calculated for the table index becomes negative. Indices smaller than 0 do not cause list accesses to return NULL, but indices being counted from the end of the list [3]. Setting the value of negative table indices to 0, instead, leads to the desired behavior of returning NULL. This is realized using function GREATEST, as shown in Listing 31.

```
1 SELECT GREATEST(0, code - eof_code) AS table_index
```

DuckDB

Listing 31: Converting an LZW code to a positional list index that does not become negative

With these changes in place, `EXPLAIN ANALYZE` is used once again to observe if the query’s runtime has changed. Indeed, the time has dropped to the order of magnitude of approximately 40 seconds for the same test image (see Table 17 for the effects of all performance optimizations). Although this is significantly faster, it is still a lot slower than the original PostgreSQL query. According to the new profiling result (Table 15) from DuckDB’s query graph tool, the majority of this remaining time is still in the recursive decompression CTE. The time for the projection operations has dropped immensely and is now less than 1 second, but the costs for `RIGHT DELIM JOINS` and `HASH GROUP BY` are still more than twice the total runtime of the whole PostgreSQL GIF decoder query.

Phase	Time	Percentage
TOTAL TIME	38.61s	100%
RECURSIVE CTE	38.01s	98.45%
HASH GROUP BY	5.62s	14.55%
RIGHT DELIM JOIN	5.43s	14.07%
HASH JOIN	2.76s	7.14%
PROJECTION	0.73s	1.88%

Table 15: Runtime analysis for `INT` representation of the code table

The most costly of these joins can indeed be located in the recursive step of the decompression CTE, whose `FROM` clause was realized as a row of `LATERAL` joins (see Section 2.4.1). That this does not lead to an optimal performance is supported by the fact that DuckDB to performs better for decorrelated joins [20].

As an effort to reduce these costs, its `FROM` clause is restructured to require less `LATERAL` joins. The code in subqueries that are only referenced once is moved in place of that reference. Column definitions that are not required for any other columns are moved up into the `SELECT` clause. With these changes, the `FROM` clause can be reduced to a join of three tables: the recursive call, the CTE with the algorithm parameters and a nested `WITH` clause (which performs better if materialization is not enforced). The new analysis provided by the query graph tool Table 16 shows that these measurements do in fact bring down both the `HASH GROUP BY` and the `RIGHT DELIM JOIN` time.

Phase	Time	Percentage
TOTAL TIME	8.49s	100%
RECURSIVE CTE	7.97s	93.91%
HASH GROUP BY	1.32s	15.52%
RIGHT DELIM JOIN	0.88s	10.35%
HASH JOIN	0.55s	6.53%
PROJECTION	0.47s	5.50%

Table 16: Runtime analysis for reduced `LATERAL` joins in `decompression_steps`

These changes lead to a final run time of just below 9 seconds on average (see also the runtime overview in Table 17). According to the analysis shown in Table 16, more than 90% of that time still comes from the recursive CTE for the decompression algorithm. A little more than 1 second is still caused by `HASH GROUP BY` operators in `decompression_steps`. But even the costs of all operators in the subgraph of this recursive CTE only add up to less than half of the CTEs indicated runtime. The strong impact of reducing the use of `LATERAL` joins has shown that worse performing joins also seem to have an overproportional negative effect on the recursive CTEs overall runtime. In addition to the rather complex `FROM` clause in the recursive step, another reason why the DuckDB implementation still performs worse

than the PostgreSQL original might have to do with the fact that this query is not structured in a way that allows DuckDB to apply its strategies for a fast evaluation of recursive CTEs. Although DuckDB supports parallelization in recursive CTEs [17], the iterations of the decompression algorithm cannot be executed in parallel. Each iteration not only depends on the LZW code and the color pattern from the previous iteration, but also on the current state of the code table, which is a result of all previous iterations of the algorithm.

	optimized for readability	INT [] code table	INT [] code table & less LATERAL
Time	119.79s	38.39s	8.86s

Table 17: Averaged runtimes for different versions of the DuckDB GIF decoder query

2.6.3. Conclusion

Using DuckDB’s dictionary type `MAP` to make up for the unavailability of type `HSTORE`, as well as the differing rules around materialization has lead to a significantly worse performance. Depending on the implementation, `LATERAL` joins were also observed to hold back the query’s performance. This is due to the fact that they affect the recursive step of the query’s biggest CTE that performs many iterations. Because it is not possible to apply parallelization to the evaluation of the recursive CTE, DuckDB cannot apply its full potential of fast query execution for this GIF decoder query. However, choosing an efficient method to store the code table entries, modeling joins after DuckDB’s strengths and explicitly materializing CTEs can significantly improve the query’s performance.

Topic 2: GPT inference

3.1. Motivation

The final and perhaps most ambitious SQL query discussed in this thesis is Quassnoi's implementation of the inference of a generative language model. This query, described in his post "*Happy New Year: GPT in 500 lines of SQL*" [19], uses GPT-2 parameters obtained from OpenAI's website to generate English sentences that are grammatically and syntactically correct and that appear to be a logical continuation of the input text. This is achieved by implementing a simplified algorithm of the inference of a Generative Pre-trained Transformer (GPT), based off an article [15] by J. Mody. The goal of this simplified version, as described in this article, is to provide a general understanding of how this type of generative artificial intelligence creates texts, without facing the complexities of all existing optimizations. It was then used by Quassnoi as a basis to tackle the considerable challenge of realizing model inference using SQL. With the popularity of generative artificial intelligence on the rise, this topic lends itself well to showcase that even such complex tasks can be realized using SQL.

3.2. Problem Specification

The starting point of the inference query's computations is an input text, for example the beginning of a sentence, which is also referred to as the "prompt". In the end, the query returns a sensible continuation of this prompt. To achieve this, the first step is to find a representation of the prompt that involves numbers, which the program then can perform its calculations on. This is achieved using a token dictionary that maps character clusters to code points, or "tokens". The dictionary is prepared beforehand in a table named "tokenizer". With such a mapping at hand, the Byte Pair Encoding algorithm is applied to every word in the prompt to find the best sequence of tokens to represent it. From this point forward, the prompt is stored as a list of tokens, rather than a sequence of characters. For each of these tokens, a vector embedding is calculated. These vectors are meant to capture the positional and linguistic properties of the token and can be passed as input into the neural network architecture. [19]

The key to determining the continuation of the prompt is the implementation of the Generative Pre-trained Transformer, a neural network architecture that uses an attention function to allow the vector embeddings to influence each other [21]. This process transforms the set of vector embeddings of independent tokens into embeddings of the sequence of tokens, taking into account their context in the sentence. The final vector embedding returned by the neural network then captures the properties of a token that is deemed most likely to continue the input sentence. After adding a matching token to the prompt, the whole process is repeated until the desired number of tokens has been added (see also figure Figure 5).

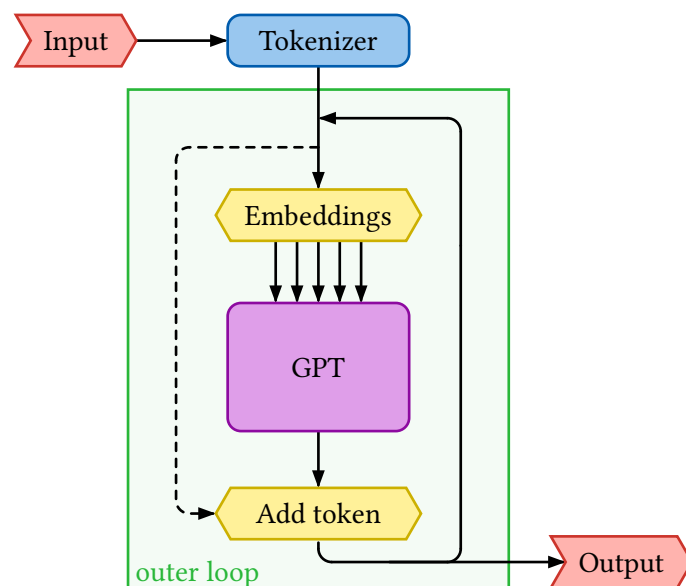


Figure 5: Overview of the structure of the GPT inference query

The Generative Pre-Trained Transformer consists of a series of layers, or “blocks”, each performing a multi-head self-attention step and a feedforward step. The implementation of these steps is essentially a series of calculations that involve linear transformations, *i.e.*, multiplying transformation matrices onto vectors. Although the formula for these calculations is the same for each block, the entries of the transformation matrices are different. They correspond to the parameters of the transformer blocks, which were determined using deep learning. This query uses the parameters of GPT-2, which are downloaded and inserted into tables where the query can access them beforehand. After the output of a block has been calculated, it is passed on as input for the next block. The last output vector of the final block is a representation of the predicted continuation of the prompt. Using a prepared table that maps all available tokens to their vector embedding, the query can then determine the token whose embedding most closely matches the transformer output and append it to the prompt. Alternatively, the parameters of the query can be adjusted so that it chooses one of the top n candidate tokens with a certain degree of randomness. This way, the generated outputs are not always the same for the same input and they appear more creative. [19] Once a token has been added to the prompt, the vector embeddings including that of the new token are once again passed as input into the transformer. With the new transformer output, yet another token can be added to the prompt. This process is repeated recursively until the prompt has reached the desired length.

3.3. From PostgreSQL to DuckDB

3.3.1. Unnesting With Ordinality

There are several instances in the inference query where an array must be expanded to a set of table rows, one for each of its elements. This is achieved using the array function `UNNEST`. In most cases, it is necessary to keep track of the order in which the values appeared in the array. Since table rows are unordered, the original order of the values would usually be lost in that process. PostgreSQL offers the option to specify a `WITH ORDINALITY` clause in combination with the `UNNEST` function, which adds an additional column to the result that enumerates the values.

As of writing this thesis, this clause is not yet supported by standard DuckDB. Instead, DuckDB's documentation [3] on unnesting suggests to use the function `GENERATE_SUBSCRIPTS` to generate the array indices as an additional column alongside the unnesting of the array. The function requires two arguments, an array and the dimension for which indices should be generated. For the purpose of the query discussed in this chapter, this will always be dimension 1. In the newest version of DuckDB available as of writing this thesis, `GENERATE_SUBSCRIPTS` cannot be used as a table function and therefore must be used in the `SELECT` clause. For the indices to appear in the same row as the corresponding array value, the function `UNNEST` also must be placed in the `SELECT` clause, as shown in Listing 32. This corresponds to how the documentation proposes using them.

```
1 SELECT * PostgreSQL
2 FROM UNNEST(array_val) WITH ORDINALITY AS t(v, i)
1 SELECT UNNEST(array_val) AS v, DuckDB
2 GENERATE_SUBSCRIPTS(array_val, 1) AS i
```

Listing 32: Ordinality as it is used in the PostgreSQL query compared to the DuckDB alternative.

In some cases, the PostgreSQL inference query only unnests a list `WITH ORDINALITY` in order to perform some calculation on each of its elements before aggregating them again. DuckDB supports a list function named `LIST_TRANSFORM` that applies a given lambda to each element of a list. It thus provides the same functionality without having to unnest and then aggregate the list elements.

```
1 SELECT ARRAY_AGG(<exp(x)> ORDER BY id) PostgreSQL
2 FROM UNNEST(1) WITH ORDINALITY ord (x, id)
1 SELECT LIST_TRANSFORM(1, x -> <exp(x)>) DuckDB
```

Listing 33: Applying expression `<exp(x)>` to all elements `x` of a list `1`

3.3.2. Adapting the Tokenizer for DuckDB

Before passing the prompt to the tokenizer, the string is split into several parts if it consists of multiple words (or other substrings separated by whitespace characters). In order to determine where to split the string, the PostgreSQL query uses a POSIX-style regular expression that consists of several branches separated by the alternation operator `|`. One of its branches matches a sequence of one or more whitespaces, followed by the negative look-ahead `(?!\S)`. According to the PostgreSQL documentation [7], this regular expression uses an extension that is not part of the POSIX standard. It checks in this case that no non-whitespace character follows the match. This means that the match must be followed by a whitespace character, but this whitespace character is not included as a part of the match.

DuckDB uses the RE2 library⁴ for regular expressions, where this negative lookahead is not supported. However, the branch of the regular expression that includes it is not strictly necessary for the query to work. The intended purpose of this branch was probably to exclude the last whitespace from matches for a series of whitespace characters. But in combination with the last branch of the regular

⁴<https://github.com/google/re2>

expression, `'\s+'`, which matches a sequence whitespace characters without the negative lookahead, this never comes to pass. As it is stated in the PostgreSQL documentation for pattern matching, regular expressions made up of branches connected by the alternation operator `|` are greedy, meaning the longest match is chosen whenever multiple possible matches start at the same point in the string. Since the last branch does not exclude the final whitespace character from the match, it will always be chosen over the match from the negative lookahead. As a consequence, the DuckDB query will work just as well as the original query without including the negative lookahead in the regular expression.

The function that is used in the PostgreSQL query to split the prompt using a regular expression is `REGEXP_MATCHES`. If called with the “g” flag, it returns one row for every match, each containing a text array with the matched substring [7]. It is used in the `FROM` clause in combination with a `WITH ORDINALITY` clause to generate a second column that contains the position index of the matches in the prompt.

For the DuckDB implementation, the function `REGEXP_EXTRACT_ALL` offers a similar functionality. However, instead of producing a set of table rows, this function returns a single array containing all the matches in the prompt. This array must be unnested to produce a table row for every match. Instead of a `WITH ORDINALITY` clause, the array indices are generated using function `GENERATE_SERIES`, a method discussed in Section 3.3.1.

```

1 -- Generate table rm (text[], bigint)
2 REGEXP_MATCHES(prompt, <reg_exp>) WITH ORDINALITY AS rm (part, part_position)

```

PostgreSQL

```

1 -- Define subquery rm (varchar, bigint)
2 SELECT UNNEST(parts) AS part,
3        GENERATE_SUBSCRIPTS(parts, 1) AS part_position
4 FROM   (
5     SELECT REGEXP_EXTRACT_ALL(prompt, <reg_exp>) AS parts
6 ) q

```

DuckDB

Listing 34: Excerpt from Quassnoi’s GPT inference query [19] to split the prompt into parts and a DuckDB implemenattion providing the same functionality

In the PostgreSQL query, column “part” has type `TEXT[]`, while its type in the DuckDB implementation is `VARCHAR`. Whenever this column is referenced in the PostgreSQL query, the `TEXT` value must be extracted from the array using the expression `part[1]`, while the DuckDB query can directly use it as a string value.

The code for splitting the prompt into parts is followed by the implmentation of the Byte Pair Encoding (BPE) algorithm. Its DuckDB implementation and the intention behind its code are presented in Section 3.5.2. For the sake of porting it to DuckDB, an alternative for the original query’s use of the `CONVERT_TO` function must be found. This function is not supported by DuckDB and is used to obtain the UTF-8 encoding of each part of the prompt. Afterwards, the individual bytes of this encoding are extracted one by one using the `GET_BYTE` function. These extracted bytes are equivalent to the byte encodings of the single characters in the part. Therefore, DuckDB’s `UNICODE` function can be used directly on the single characters in the prompt part to achieve the same result, as this function returns the unicode of the first character in its argument [3]. Both implementations are shown in Listing 35.

<pre> 1 SELECT GET_BYTE(bytes, n) 2 FROM CONVERT_TO(part[1], 'UTF-8') AS bytes 3 CROSS JOIN LATERAL 4 GENERATE_SERIES(0, LENGTH(bytes) - 1) AS n </pre> <p style="text-align: right;">PostgreSQL</p>	<pre> 1 SELECT UNICODE(part[n + 1]) 2 FROM GENERATE_SERIES(0, LEN(part) - 1) AS gs(n) </pre> <p style="text-align: right;">DuckDB</p>
---	---

Listing 35: Representation of how the PostgreSQL inference query [19] gets the byte encodings of all characters in part (left) and the DuckDB implementation that provides the same functionality (right)

With the changes discussed so far, the implementation of the BPE algorithm compiles in DuckDB. However, it does not return the correct result for prompts consisting of more than one word (or “part”). As described in the blog post [19], the BPE algorithm is supposed to be applied separately to each part of the prompt. It recursively merges adjacent characters into clusters whose token representations are deemed to be the best. The recursion stops when there is no pair of clusters left that when merged can be represented jointly as a single token. This process will be explained in more detail in Section 3.5.2, but for the sake of porting this implementation to DuckDB, it must be ensured that only characters of the same word are merged into clusters. In order to apply the algorithm to every part separately in the PostgreSQL GPT inference query, the recursive CTE implementing the BPE algorithm is placed in a subquery connected by a `LATERAL` join to the table containing the prompt parts.

```

1 ...
2 FROM <rm (part, part_position)>
3 CROSS JOIN LATERAL
4 (
5   <BPE implementation>
6 )

```

Listing 36: FROM clause structure proposed in [19] to apply BPE to every part (Listing 34)

However, looking at the query outputs of the compiling DuckDB version shown in Table 18, this does not seem to work as intended in DuckDB.

part_position	position	cluster
1	1	'Hello'

part_position	position	cluster
1	1	'HG'
1	2	'H'
1	3	'G'
1	4	'ew'
2	1	'HG'
2	2	'H'
2	3	'G'
2	4	'ewe'

Table 18: Output of the compiling DuckDB BPE implementation for prompt 'Hello' (left) and the faulty output for prompt 'Hello world' (right), where 'G' represents a space

For a prompt consisting of only the word 'Hello', the algorithm behaves just as it should and recursively merges neighbouring clusters of the single prompt part. If another word is added to the prompt, separated by a whitespace, the prompt is split into two parts. What can be observed in the query output (Table 18) is that now, characters from different parts of the prompt are merged into clusters (e.g., the 'e' in 'Hello' and the 'w' in ' world'). This is unwanted behavior and leads to incorrect outputs.

The window function `LEAD` is what is responsible for merging adjacent characters. It operates on the input rows of the previous iteration, ordered by their `position` value, which is from iteration 2 onward determined using the window function `ROW_NUMBER`. The definitive merge occurs later in the `SELECT` clause of the nested `WITH` clause. Explicitly separating the rows for different `part_position` values by combining all window functions with a `PARTITION BY part_position` clause and joining CTEs on their `part_position` value in the subquery that performs the merges leads to correct outputs (Table 19). This requires that `part_position` is included as an additional column in all the affected subqueries, which was not the case before.

part_position	position	cluster
1	1	'Hello'

part_position	position	cluster
1	1	'Hello'
2	1	'Ġworld'

Table 19: Correct outputs of the modified DuckDB BPE implementation for prompt 'Hello' (left) and for prompt 'Hello world' (right), where 'Ġ' represents a space

3.3.3. Vector and Matrix Calculations

Passing vector embeddings through the transformer requires performing calculations that involve vectors and matrices. These types of operations appear in many different parts of the query. This subchapter will present which operations have to be implemented, how the original PostgreSQL query realizes them and what needs to be changed to make it work for DuckDB.

Given two row vectors $v, w \in \mathbb{R}^{1 \times m}$, a column vector $u \in \mathbb{R}^{m \times 1}$ and matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$ the following operations need to be implemented for the GPT inference query:

Description	Notation
Elementwise addition	$v + w$
Elementwise multiplication	$v \odot w$
Dot product	$v \cdot u$
Matrix-vector multiplication	$A \cdot u$
Matrix-matrix multiplication	$A \cdot B$

Table 20: Overview of matrix and vector operations

Looking at how matrix-vector and matrix-matrix multiplications are calculated, it makes sense to think of the matrices as collections of row or column vectors. This makes it possible to calculate the results for both of these operations only using the dot product, which is already a required operation for this query, anyway. For an m -dimensional vector $u \in \mathbb{R}^{m \times 1}$ and a matrix $A \in \mathbb{R}^{n \times m}$ with m -dimensional row vectors a_1, \dots, a_n , the matrix-vector multiplication is calculated as follows:

$$A \cdot u = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \cdot u = \begin{pmatrix} a_1 \cdot u \\ a_2 \cdot u \\ \vdots \\ a_n \cdot u \end{pmatrix} \quad (1)$$

Let there two matrices $A \in \mathbb{R}^{n \times m}$ with row vectors a_1, \dots, a_n and $B \in \mathbb{R}^{m \times p}$ with column vectors b_1, \dots, b_p . The result of a matrix-matrix multiplication of $A \cdot B$ is a matrix with n rows and p columns. Its entries are obtained by calculating the dot product of every row vector in A with every column vector in B :

$$A \cdot B = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \cdot (b_1 \ b_2 \ \dots \ b_p) = \begin{pmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 & \dots & a_1 \cdot b_p \\ a_2 \cdot b_1 & a_2 \cdot b_2 & \dots & a_2 \cdot b_p \\ \vdots & \vdots & \ddots & \vdots \\ a_n \cdot b_1 & a_n \cdot b_2 & \dots & a_n \cdot b_p \end{pmatrix} \quad (2)$$

PostgreSQL: The original GPT inference query proposed by Quassnoi [19] uses the pgvector⁵ extension. This extension allows the definition of vectors and supports a number of operations on them. The ones that are used in the inference query are shown in Table 21.

⁵<https://github.com/pgvector/pgvector>

Description	PostgreSQL implementation
Elementwise addition	$v1 + v2$
Elementwise multiplication	$v1 * v2$
Dot product	<code>INNER_PRODUCT(v1, v2)</code>

Table 21: PostgreSQL code for operations using `VECTOR` values defined with extension `pgvector`

As the extension allows the query to efficiently perform these vector-vector operations, they are used to realize operations that involve matrices as well (see Equation 1 and Equation 2). This also affects how the entries of transformation matrices are stored as tables. Instead of each row storing a single value alongside its row and column index, most tables store row vectors (see also Table 22).

row_id	col_id	value
1	1	a_{11}
1	2	a_{12}
...
1	m	a_{1m}
2	1	a_{21}
...

row_id	values
1	$[a_{11}, a_{12}, \dots, a_{1m}]$
2	$[a_{21}, a_{22}, \dots, a_{2m}]$
...	...

Table 22: Table representations of a matrix A with m columns and entries a_{ij} storing single entries (left) and storing row vectors (right)

As a consequence, a `CROSS JOIN` of two tables representing a matrix A and a set of vectors v_i combines every row vector of A with every vector v_i in the set. This is exactly what is required to calculate the matrix-vector multiplications $A \cdot v_i$ using the dot product (Equation 1). Assuming that m is both the number of columns in A and the number of entries in the vectors v_i , a typical set of matrix-vector multiplications is implemented as shown in Listing 37.

```

1 -- For tables vectors(vector_id, values) and matrix_rows(row_index, values) representing
2 -- vectors v and a matrix A, calculate for each vector_id: A * v
3 SELECT v.vector_id,
4        ARRAY_AGG(INNER_PRODUCT(v.values, a.values) ORDER BY a.row_index)::VECTOR(m)
5 FROM   vectors AS v
6 CROSS JOIN
7        matrix_rows AS a
8 GROUP BY
9        v.vector_id;

```

Listing 37: Matrix-vector multiplications as they are realized in the GPT inference query [19]

Transposing a matrix represented as a set of row vectors is as simple as treating its row vectors as column vectors. For two tables representing matrices A and B with the same dimensions, a `CROSS JOIN` returns all combinations of row vectors in A and column vectors in B^T . These combinations are exactly what is required to calculate $A \cdot B^T$ using the dot product according to Equation 2. Listing 38 is an exemplary demonstration of how the implementation for this type of matrix-matrix multiplications is realized.

```

1 -- For matrices A and B of the same dimensions, calculate: A * B^T
2 SELECT a.row_index AS row_index,
3        b.row_index AS column_index,
4        INNER_PRODUCT(a.values, b.values) AS values
5 FROM   matrix_1 AS a
6 CROSS JOIN
7        matrix_2 AS b;

```

Listing 38: Matrix-matrix multiplication $A \cdot B^T$ for matrices A, B of the same dimensions, as it is realized in the GPT inference query [19]

The inference query’s implementation of a matrix-matrix multiplication where the second matrix is not transposed (and thus not represented as a set of column vectors) is more complicated. Let $A \in \mathbb{R}^{n \times m}$ and $C \in \mathbb{R}^{m \times n}$ be the matrices for which $A \cdot C$ is calculated. In the context of the GPT inference query, matrix A is represented as a table where each row holds a row index i , a column index j and a single element a_{ij} . Matrix C is represented as a set of row vectors. In order to realize the matrix-matrix multiplication, each value a_{ij} of matrix A is joined with the row vector c_j of matrix C , so that the row index of c_j is equal to the column index of value a_{ij} . Then, the value a_{ij} is multiplied onto every element of the row vector c_j . This is achieved by creating a vector of equal length whose values are all set as a_{ij} and then performing elementwise multiplication with row vector c_j using the $*$ operator. The resulting vectors are then grouped by the row index i of value a_{ij} and reduced to a single vector using aggregate function `SUM`, which performs elementwise addition thanks to the `pgvector` extension.

```

1 -- Matrix-matrix multiplication A * C, where C has n columns
2 SELECT a.row_index AS row_index,
3        SUM(ARRAY_FILL(a.value, ARRAY[n])::VECTOR(n) * c.values) AS values
4 FROM   matrix_1 AS a
5 JOIN   matrix_2 AS c
6 ON     a.column_index = c.row_index
7 GROUP BY
8        a.row_index;

```

Listing 39: Matrix-matrix multiplication without being able to join row and column vectors, as it is realized in the GPT inference query [19]

DuckDB: Just like in standard PostgreSQL, DuckDB also does not support a vector data type for SQL yet, although they are used in the internal representation of data. The alternative is to use either arrays or lists. DuckDB’s `ARRAY` type is for fixed-size arrays, which matches the properties of type `VECTOR`, whose length is also fixed. Therefore, the `ARRAY` type is used as a substitute for `VECTOR` in the definition of the prerequisite tables that store the entries of the linear transformation matrices. Concerning the vector operations performed in the original query, it depends on the type of operation which type is best suited. The `ARRAY` type supports a function to calculate the dot product. Values of the `LIST` type can be altered using function `LIST_TRANSFORM`, which applies a given lambda function to every list element. The dot product could also be implemented by nesting multiple list functions, but the dot product function for arrays is the more direct equivalent of what is used in the PostgreSQL query. Since type casting between the two types is easy, the DuckDB query can just convert the vector representations to whatever type is the most convenient in the given context.

Description	DuckDB implementation
Elementwise addition	<code>LIST_TRANSFORM(LIST_ZIP(11, 12), x -> x[1] + x[2])</code>
Elementwise multiplication	<code>LIST_TRANSFORM(LIST_ZIP(11, 12), x -> x[1] * x[2])</code>
Dot product	<code>ARRAY_DOT_PRODUCT(a1, a2)</code>

Table 23: DuckDB implementations of operations involving two arrays (a1, a2) or two lists (11, 12)

Realizing matrix-vector multiplications and the matrix-matrix multiplication $A \cdot B^T$ in DuckDB only requires replacing PostgreSQL's `INNER_PRODUCT` function with DuckDB's `ARRAY_DOT_PRODUCT` function and omitting the type casting to `VECTOR` in Listing 37 and Listing 38. For the other type of matrix-matrix multiplication, a workaround for the missing definition of a `SUM` aggregate on vectors is required. DuckDB's list function `LIST_REDUCE` reduces a list of values to a single value by applying a lambda to all elements. Assuming the set of vectors is available as a list of list representations, using `LIST_REDUCE` with a lambda that performs elementwise addition of two lists leads to the desired result. The implementation of elementwise addition of two lists was already discussed (see also Table 23) and serves as the body of this lambda. In order to aggregate the set of vector representations as a list to which `LIST_REDUCE` can be applied, the aggregate function `ARRAY_AGG` is used. The final change that must be applied to the PostgreSQL implementation is to find an alternative for the `ARRAY_FILL` function, which is not supported by DuckDB. Instead, the function `LIST_RESIZE` is used to define a list of an indicated length where all entries have the same value. Putting everything together, the DuckDB implementation for this type of matrix-matrix multiplications is shown in Listing 40.

```

1  -- Matrix-matrix multiplication A * C, where C has n columns
2  SELECT a.row_index AS row_index,
3         LIST_REDUCE(
4           ARRAY_AGG(
5             LIST_TRANSFORM(
6               LIST_ZIP(LIST_RESIZE([]::REAL [], n, a.value)::REAL [], c.values),
7               x -> x[1] * x[2]
8             )
9           ),
10        (x, y) -> LIST_TRANSFORM(LIST_ZIP(x, y), z -> z[1] + z[2])
11    ) AS values
12 FROM   matrix_elems AS a
13 JOIN   matrix_rows AS c
14 ON     a.column_index = c.row_index
15 GROUP BY
16        a.row_index;

```

Listing 40: DuckDB implementation of Listing 39

3.3.4. Other Necessary Changes

This subchapter documents smaller changes that had to be applied to the GPT inference query in order to port it to DuckDB.

Population variance:

The PostgreSQL inference query uses the aggregate function `VAR_POP` to calculate the population variance of the elements of a vector. This is equal to the square of the population standard deviation [7] defined by the following formula, where x_i is the i th of N input values and μ their mean value:

$$\sigma^2 = \frac{\sum_{i=1}^N (x_i - \mu)^2}{N} \quad (3)$$

In order to apply `VAR_POP` to vector elements, they are once again unnested after casting the vector to a list. In DuckDB, this can instead be realized by combining several list functions, making unnesting unnecessary. This approach uses the following functions:

- `LIST_AVG` to calculate the mean μ
- `LIST_TRANSFORM` to transform every element x_i to $(x_i - \mu)^2$
- `LIST_SUM` to add up the transformed list elements
- `LEN` to obtain the total number of elements N

Listing 41 shows the implementations of the approach used by the PostgreSQL inference query and of the proposed DuckDB implementation. Since the calculations that follow in the GPT inference query require both the variance and the mean, both values are defined as columns in the `SELECT` clause.

<pre> 1 SELECT AVG(value) AS mean, 2 VAR_POP(value) AS variance 3 FROM UNNEST(vector::REAL[]) value </pre>	PostgreSQL	<pre> 1 SELECT mean, 2 LIST_SUM(3 LIST_TRANSFORM(list_rep, x -> (x - mean)^2) 4) / LEN(list_rep) AS variance 5 FROM (SELECT LIST_AVG(list_rep) AS mean) avg_val </pre>	DuckDB
---	------------	--	--------

Listing 41: Calculation of the mean population variance in the PostgreSQL query [19] (left) and its implementation using list functions supported by DuckDB (right)

Resolving circular references:

In the inner loop of the GPT inference query, a row of CTEs is used to get the linear transformation parameters of the current block from the prerequisite tables. The names of these CTEs are identical to the names of the tables they select from. DuckDB demands that this is resolved by either adding `RECURSIVE` to make the CTEs reference themselves or explicitly stating the schema, with “main” being the default schema. Alternatively, renaming the CTEs also leads to the query working properly.

<pre> 1 WITH ln_2_b AS 2 (3 SELECT * 4 FROM ln_2_b 5 WHERE block = q.block 6) </pre>	PostgreSQL	<pre> 1 WITH ln_2_b_params AS 2 (3 SELECT * 4 FROM ln_2_b 5 WHERE block = q.block 6) </pre>	DuckDB
--	------------	---	--------

Listing 42: Example for resolving issue caused by circular references in DuckDB

3.4. Readability Optimization

3.4.1. Discussion of the Porting Effects on Readability

The vector and matrix operations discussed in Section 3.3.3 have the biggest impact on the query's readability. Since the PostgreSQL query proposed by Quassnoi [19] relies on an extension to allow the use of vectors and DuckDB v1.1.1. does not support vector operations, alternatives had to be implemented. These are by default not as short and easy to read as when for example elementwise addition of vectors was implemented for the `+` operator. Section 3.3.4 also shows that the formula of the population variance has to be implemented to make up for the missing support of the `VAR_POP` function in DuckDB. With the exception of calculating the dot product, realizing all these operations requires nesting multiple list functions, which leads to complex expressions. This issue can partly be resolved by defining them as macros. Although their names are still longer than operators, they hide the nested list function calls in their definition and at the same time allow to avoid repeating these implementations throughout the query.

Another disadvantage macros have compared to the vector operations predefined by the `pgvector` extension is that realizing multiple such operations in a row would require nesting macro references, e.g., `vec_add(vec_mul(v1, v2), v3)` for `v1 * v2 + v3`. As an alternative, a macro `vec_mul_add(v1, v2, v3)` that performs both operations is defined, which makes reading the expression slightly more fluid. An overview of all defined macros and their effect on the query is shown in Section 3.4.3.

As already discussed in Section 3.3.1, the `LIST_TRANSFORM` function supported by DuckDB makes the implementation of operations performed on all entries of a list more straightforward. This becomes relevant in the GPT inference query once an operation must be performed on all entries of a vector. The PostgreSQL query casts the vector to a list in order to then unnest its entries and perform the operation on them. The result values are then aggregated to an array that can be casted to type `VECTOR`. Listing 43 shows how this compares to using `LIST_TRANSFORM` in the DuckDB query. Because the `LIST_TRANSFORM` function makes the implementation shorter and because its name accurately describes the expression, it improves the readability of all four subqueries where this type of operation occurs.

```
1 SELECT ARRAY_AGG(<exp(x)> ORDER BY id)::VECTOR(n)
2 FROM UNNEST(vector::REAL[]) WITH ORDINALITY ord(x, id)
```

PostgreSQL

```
1 SELECT LIST_TRANSFORM(values::REAL[], x -> <exp(x)>)
```

DuckDB

Listing 43: Applying expression `<exp(x)>` to all elements `x` of a vector (representation) with n elements

3.4.2. Further Improving Readability

Nesting is one of the major reasons why the GPT inference query is so complex and hard to grasp. A certain depth of nesting is unavoidable, because the query is implemented as a nested recursion (as proposed in [19]). After porting it to DuckDB, the query includes a total of six `WITH` clauses, four of which are recursive. The CTE named `gpt` of the outermost `WITH` clause is the most deeply nested CTE of all, as its `FROM` clause holds the nested recursion. Its structure is presented in Listing 44 and unchanged from the original PostgreSQL query.

```

1  gpt AS (
2  < gpt base case >
3  UNION ALL
4  -- Recursive step of the outer recursion:
5  SELECT ...
6  FROM gpt -- outer recursive call
7  CROSS JOIN LATERAL
8  (
9  WITH RECURSIVE ...
10     transformer AS
11     (
12     < transformer base case >
13     UNION ALL
14     (
15     -- Recursive step of the inner recursion:
16     WITH previous AS
17     (
18     SELECT *
19     FROM transformer -- inner recursive call
20     )
21     SELECT ...
22     FROM ...
23     CROSS JOIN LATERAL
24     < innermost WITH clause >
25     )
26     ),
27     ...
28     ) AS next_token
29  WHERE ...
30 )

```

DuckDB

Listing 44: Structure of CTE gpt as proposed in [19]

When the CTEs of the outermost `WITH` clause are considered to be at depth 0, then the depth of the `<innermost WITH clause>` is 2. Because the structure of this nested recursion itself is already quite complex, the goal is to keep any further nesting in the CTEs of the innermost `WITH` clause to a minimum. As of now, however, 6 of the CTEs are still nested and lead to a maximum depth of 6.

	mha_norm	attention	mha	ffn_norm	ffn_a	ffn
Depth	+4	+1	+1	+2	+2	+1

Table 24: Additional depths of the nested CTEs within the `<innermost WITH clause>` at depth 2

In order to decrease their depth and thus improve the readability, the code of some of these CTEs is spread across multiple CTEs with a maximum depth of 1. This allows to read these newly defined CTEs from top to bottom as a series of substeps with a descriptive name. It also reduces the number of tasks of each CTE. For example, CTE `mha_norm` both normalizes the input values of the GPT layer and applies a linear transformation to calculate the entries of matrices named Q , K , and V . Therefore, it can be split into two CTEs named `mha_norm` and `mha_qkv`. Similarly, CTE `ffn_a` is split into `ffn_fc` and `ffn_gelu`, with the former performing a linear transformation and the latter applying the GELU function to the resulting values.

The CTE attention has only a depth of 1, but it performs a complicated matrix-matrix multiplication for each set in addition to merging the results of all sets. It is split into the CTEs `attention_heads` and `merge_heads`, both of depth 0.

The depth of CTEs outside of the innermost `WITH` clause can be reduced as well. CTE `tokens`, a sibling of `transformer`, has a depth of 4. It applies the softmax function to the similarity scores of every token with the GPT prediction result and also defines a subrange of interval $[0, 1]$ for each of them (see Section 3.5.7). This CTE is separated into `softmax_logits` and `token_ranges` of depth 1. CTE `output`, which determines the characters that make up the output string and also aggregates them, has a depth of 2. It is split into the CTEs `output_chars`, which holds the code of the original CTEs `FROM` clause and has depth 1, and `output`, which now only performs the task of aggregating the characters. This is facilitated by the fact that unlike PostgreSQL, DuckDB allows column aliases to be referenced by sibling columns of the same `SELECT` clause [3]. As a consequence, the column definitions of both subqueries making up the original CTE's `FROM` clause can be moved into the `SELECT` clause of the new CTE `output_chars`, even though they are correlated.

Other than defining nested subqueries as CTEs, some subqueries can be simplified in ways that reduce their depth. Both the new `mha_norm` and CTE `ffn_norm` can be simplified to reduce their depth by 1 following the strategy presented in Listing 45. It shows how nested subqueries whose `SELECT` clause only makes the values provided by its `FROM` clause available to the surrounding subquery can be merged into the main query block.

<pre> 1 -- Original query structure 2 SELECT <expr(q)> 3 FROM (4 SELECT q.* 5 FROM ... 6 CROSS JOIN LATERAL 7 (...) AS q 8) </pre>	DuckDB	<pre> 1 -- Simplified query structure 2 SELECT <expr(q)> 3 FROM ... 4 CROSS JOIN LATERAL 5 (...) AS q </pre>	DuckDB
--	--------	--	--------

Listing 45: Subquery simplification that leads to a reduction of the depth of nesting

Similarly, some `LATERAL` queries only serve the purpose of defining the values that will then be selected in the query block's `SELECT` clause, without having a `FROM` clause of their own. This keeps complex expressions out of the `SELECT` clause, but increases the complexity of the query's structure. Moving the code of such `LATERAL` subqueries into the `SELECT` clause, as shown in Listing 46, makes the query more compact. This reduces the number of query blocks and leads to a more straightforward implementation. This type of simplification can be applied to a CTE named `logits` and the new CTE `ffn_gelu`.

<pre> 1 -- Original query structure 2 SELECT q2.* 3 FROM (...) AS q1 4 CROSS JOIN LATERAL 5 (6 SELECT <expr(q1)> 7) AS q2 </pre>	DuckDB	<pre> 1 -- Simplified query structure 2 SELECT <expr(q1)> 3 FROM (...) AS q1 </pre>	DuckDB
--	--------	---	--------

Listing 46: Simplifying query blocks to remove `LATERAL` joins

Furthermore, `LATERAL` subqueries with a `FROM` clause and a correlated `WHERE` clause can under some circumstances be merged into the query block that surrounds it. As shown in Listing 47, this simplification turns the `LATERAL` join into a simple `CROSS JOIN`. This can be applied to the CTE named `next_token`.

```

1 -- Original query structure                                DuckDB
2 SELECT *
3 FROM ( ... ) AS q1
4 CROSS JOIN LATERAL
5 (
6     SELECT *
7     FROM t
8     WHERE <cond(q1)>
9 ) AS q2

1 -- Simplified query structure                            DuckDB
2 SELECT *
3 FROM ( ... ) AS q1
4 CROSS JOIN
5 t
6 WHERE <cond(q1)>

```

Listing 47: Simplifying query blocks to transform **LATERAL** joins to normal joins

In addition to the readability considerations discussed so far, some CTEs or columns are given slightly more descriptive names. Sometimes, a changed name is also just a reflection of how a CTE's code was modified. The original CTE `block_output` does only select the output of the last GPT block, as its name suggests. But in the DuckDB query version optimized for readability, it only selects the relevant vector of said block output, which represents the predicted prompt continuation. This simplifies the CTE logits, which was originally responsible for this. In order to reflect this change, CTE `block_output` is renamed to `gpt_prediction`. Table 25 lists all the CTE splits and name changes that were applied.

Type	original name	renamed to	Comment
CTE	input	input_params	
CTE	tokens		
column	input	input_tokens	
CTE	gpt		
column	input	prompt_tokens	
field	n_seq	n_tokens	
field	block	block_num	
CTE	transform	transformer	
CTE	mha_norm	mha_norm, mha_qkv	split CTE
CTE	sm_input		
column	x	row_id	also in: sm_diff, sm_exp, softmax
column	y	col_id	also in: sm_diff, sm_exp, softmax
CTE	mha		
field	w	m	also in: ffn_fc, ffn
CTE	attention	attention_heads, merge_heads	split CTE
CTE	ffn_a	ffn_fc, ffn_gelu	split CTE
CTE	block_output	gpt_prediction	
CTE	tokens	softmax_logits, token_ranges	split CTE
CTE	output	output_chars, output	split CTE
field	ordinality	token_id	becomes column of output_chars
field	position	char_position	becomes column of output_chars

Table 25: Documentation of renamed CTEs, subqueries and columns

3.4.3. Conclusion

Table 26 shows the scores of different versions of the GPT inference query as to the metrics discussed in Section 1.3.

Metric	PostgreSQL	DuckDB (initial)	DuckDB (readability)
Number of CTEs	40	40	47
Total number of query blocks	79	85	74
Max. depth of nesting	6	6	3
Number of query blocks at depth > 2	19	20	7

Table 26: Metric values for different versions of the GPT inference query

Through the subquery simplifications described in Section 3.4.2, the number of query blocks has slightly decreased compared to the original query, even though porting initially resulted in a higher number of query blocks. Furthermore, the measures to decrease the depth of nesting limited the perceived maximum depth to 3. Because the nested recursion required to realize the GPT inference automatically leads to a depth of 2, this means that CTEs at this depth appear to have a maximum depth of 1. The table macro `var_pop_means` hides another depth of nesting in its definition, but it does not impact the readability. Because a depth of 2 is the required minimum for the chosen implementation, the number of query blocks that are nested more deeply is listed in Table 26. Compared to the original PostgreSQL query, the readability optimizations more than halved this number. Although a lot of the query’s complexity caused by nesting remains, the worst extends of it could be somewhat reduced. In exchange for these improvements, the number of CTEs has increased. This increase was intentionally accepted to decrease nesting (as described in Section 1.4). Defining nested subqueries as CTEs increases the number of CTEs, but not the total number of query blocks that have to be read.

Lastly, the query’s readability is improved with the help of macros. Table 27 shows all macros that are used in the version optimized for readability, next to the original PostgreSQL expression they replace. Additionally, the number in column “operations” shows how many function calls, type castings or other operations are hidden within each macro’s definition.

Macro	Type	PostgreSQL code	Operations	Occurrences
<code>vec_add(v1, v2)</code>	scalar	<code>v1 + v2</code>	4	4
<code>vec_mul(v1, v2)</code>	scalar	<code>v1 * v2</code>	4	1
<code>vec_add_3(v1, v2, v3)</code>	scalar	<code>v1 + v2 + v3</code>	5	3
<code>vec_mul_add(v1, v2, v3)</code>	scalar	<code>v1 * v2 + v3</code>	5	3
<code>var_pop_mean(values)</code>	table	<pre>SELECT VAR_POP(unnest), AVG(unnest) FROM UNNEST(values::REAL[])</pre>	7	3

Table 27: Macros that appear in the GPT inference query, how many operators, function calls or type castings they hide in their definition and how often they appear in the code

Thanks to the use of macros, the length of the expressions for elementwise vector operations is similar to their length in the PostgreSQL query. Each macros hides a long expression of nested list functions. The macro names make operations on vector operations immediately identifiable and also indicate which type of operation is performed, which would otherwise require reading a long expression each of the 11 times they appear throughout the query. Avoiding to repeat these long expressions by only defining them once in a macro helps to focus on the rest of the query, once these operations and how they are implemented have been established at the beginning.

3.5. DuckDB GPT Inference Query Explained

In this section, the DuckDB implementation of the GPT inference query [19] optimized for readability is presented. Its goal is to feed vector representations of a given prompt through a Generative Pre-Trained Transformer and use its output to add words to the prompt. The necessary steps to achieve this and how they are implemented in DuckDB is discussed in this section.

The GPT inference query consists of a `WITH` clause that defines a row of CTEs that implement subgoals of the problem. An overview of these CTEs is shown in Listing 49. The first three CTEs serve the purpose of transforming the input into a form that the transformer can perform its calculations with. The implementation of the actual GPT inference is held in the CTE named `gpt`. This recursive GPT has a complex structure of its own, which is presented in Section 3.5.3.

```
1  WITH RECURSIVE
2      input_params AS (...),      -- Set the input parameters
3      prompt_parts AS (...),     -- Split the input prompt into words
4      clusters AS (...),        -- BPE algorithm: find best character clusters to represent as tokens
5      tokens AS (...),         -- Represent the prompt as a list of tokens
6      gpt AS (...),            -- Recursive GPT inference: add one token to the prompt per iteration
7      output_chars AS (...),    -- Map the list of tokens back to character clusters
8      output AS (...)          -- Aggregate the characters that make up the response to a string
9  SELECT *
10 FROM output;
```

DuckDB

Listing 49: Overview of the structure of the outer `WITH` clause

This section covers the following subproblems and their SQL implementation:

- Setting the input parameters (Section 3.5.1)
- Tokenizing the prompt (Section 3.5.2)
- Recursively inferring the transformer (Section 3.5.3)
- Recursively calculating block outputs (Section 3.5.4)
- Multi-head self-attention step (Section 3.5.5)
- Feed forward step (Section 3.5.6)
- Choosing the next token (Section 3.5.7)
- Query output (Section 3.5.8)

3.5.1. Setting the Input Parameters

The first CTE, `input_params`, is used to define several parameters for the program. For some of them, their significance becomes most apparent once their use in the query is discussed. Table 28 shows an overview of the purpose of each parameter. Listing 50 shows the CTE’s implementation with exemplary values for the input parameters.

Column Name	Description
<code>prompt</code>	the input string, which the query is supposed to continue
<code>threshold</code>	the number of tokens that should be added to the prompt (= number of GPT inferences)
<code>top_n</code>	the number of most likely continuations from which the query should choose
<code>temperature</code>	a constant that affects the likelihood of not choosing the most likely continuation (makes outputs appear more “creative”)

Table 28: Overview of the columns in the CTE `input_params`

```

1 WITH RECURSIVE
2     input_params (prompt, threshold, top_n, temperature) AS
3     (
4     SELECT 'Happy New Year! I wish you' AS prompt,
5           8 AS threshold,
6           5 AS top_n,
7           2 AS temperature
8     ),

```

DuckDB

Listing 50: CTE where the inference query’s input parameters can be set

3.5.2. Tokenizing the Prompt

The first subgoal of the GPT inference query is to find a representation of the prompt that consists of numbers, which can then be passed as inputs into the transformer. For that purpose, two tables that help to define a mapping between character clusters and vectors are utilized. The token dictionary stored in the `tokenizer` table maps clusters to code points, *i.e.*, integer values that are also referred to as “tokens”. The Word Token Embedding stored in table `wte` stores a vector embedding $v \in \mathbb{R}^{768}$ for every token in the token dictionary. The intention behind these vector embeddings is that their dimensions represent certain properties of the character clusters concerning their meaning or grammatical function in a sentence. As a consequence, vector embeddings that are similar in their values represent words or clusters that have similar meanings or properties. The task of assigning each token a vector embedding in a way that fulfills this property is solved using deep learning, which is not a responsibility of the inference query. Instead, as a part of the preparations before running the query, the Word Token Embeddings of GPT-2 are inserted into table `wte`. [19]

Before the vector embeddings that represent the prompt can eventually be determined (as discussed in Section 3.5.3), the prompt has to be represented as a list of tokens. This is achieved by applying the Byte Pair Encoding (BPE) algorithm separately on each word, or “part”, of the prompt. Therefore, the first step is to split the prompt. In order to decide where to split the string, the inference query uses the following POSIX-style regular expression:

```
''s|'t|'re|'ve|'m|'ll|'d| ?\w+| ?\d+| ?[^\s\w\d]+|\s+'
```

The branches of this regular expression, separated by the alternation operator `|`, match:

- various shortforms of the English language (“s” for “is” or “has”, “re” for “are”, *etc.*)
- `' ?\w+'`: at most one space followed by at least one word character
- `' ?\d+'`: at most one space followed by at least one digit
- `' ?[^\s\w\d]+'`: at most one space followed by neither whitespaces, digits or word characters
- `'\s+'`: one or more whitespace characters

Using this regular expression as an argument for function `REGEXP_EXTRACT_ALL` returns an array with all matches in the prompt. The CTE `prompt_parts` (Listing 51) then selects each match in the array as a part of the prompt, while keeping track of its position in the prompt. This is achieved by unnesting the array of matches and generating its position indices using function `GENERATE_SUBSCRIPTS`.

```

9  prompt_parts (part_position, part) AS
10 (
11  SELECT  GENERATE_SUBSCRIPTS(p.parts, 1) AS part_position,
12         UNNEST(p.parts) AS part
13  FROM    (
14         SELECT  REGEXP_EXTRACT_ALL(
15                 prompt,
16                 '''s|'t|'re|'ve|'m|'ll|'d| ?\w+| ?\d+| ?[\s\w\d]+|\s+'
17             ) AS parts
18         FROM    input_params
19     ) AS p
20 ),

```

DuckDB

Listing 51: Split the prompt into parts using a regular expression

The next CTE, named `clusters`, applies the BPE algorithm to each part of the prompt. To that end, the algorithm's implementation is placed in a `LATERAL` join with CTE `prompt_parts`. The algorithm itself is realized as a `WITH` clause, which defines a single recursive CTE named `bpe`. The base case of this CTE generates one table row for every character in the part, while also storing its position index. In the recursive step, of which the implementation will be discussed later, characters that are adjacent in the part are recursively merged to form character clusters that can be represented as tokens. To that end, the recursive step checks for all possible merges if the token dictionary `tokenizer` stores a token for the cluster that would result from this merge. However, since `tokenizer` represents some characters as symbols in its `cluster` column, the base case has to convert the characters it extracts from the part to their respective representation in the `tokenizer` table. This affects whitespaces, for example. The `encoder` table, which also must be prepared before executing the GPT inference query, stores a mapping from the byte encodings of characters to their representation in table `tokenizer`. After applying the `UNICODE` function to each character extracted from the part in order to obtain the value of its encoding, it can be used to perform a join with table `encoder` and select its character representation. Listing 52 shows the implementation of the base case and how the BPE implementation as a whole is embedded in the CTE `clusters`.

```

20 clusters (part_position, position, cluster) AS
21 (
22 SELECT bpe_result.*
23 FROM prompt_parts AS pp
24 CROSS JOIN LATERAL
25 (
26 WITH RECURSIVE
27     bpe (part_position, position, character, continue) AS
28     (
29         -- Base case: Get the representation of every character in the part
30         SELECT pp.part_position,
31                gs.n::BIGINT AS position,
32                enc.character,
33                TRUE AS continue
34         FROM   GENERATE_SERIES(1, LEN(pp.part)) AS gs(n)
35         JOIN   encoder AS enc
36         ON     enc.byte = UNICODE(pp.part[n])
37         UNION ALL
38         ( <recursive step> )
39     )
40     -- Get the result of the BPE algorithm (all clusters that could not be merged further)
41     SELECT part_position,
42            position,
43            character AS cluster
44     FROM   bpe
45     WHERE  NOT continue
46     ) bpe_result
47 ),

```

Listing 52: Base case of the recursive CTE implementing the BPE algorithm embedded in the CTE clusters

In the following, the `<recursive step>` of the CTE implementing the BPE algorithm is discussed. It is realized as yet another recursive `WITH` clause that defines four CTEs. The first one only selects the rows of the previous iteration of `bpe` where column `continue` is set to `TRUE`, *i.e.*, whose characters should continue to be merged. The second CTE, named `bn`, uses window function `LEAD()` to determine all possible merges of two neighbouring characters or clusters in the part. In order to ensure that only characters that belong to the same part are merged to clusters, all window functions are combined with a `PARTITION BY` clause. The third CTE, `top_rank`, joins the rows of CTE `bn` with table `tokenizer` to obtain the values of the tokens that represent the potential clusters, if they exist. For each `part_position`, only the cluster with the highest token value is selected. This cluster is deemed the “best” merge, which should be performed in this iteration of the algorithm. In order to find it, the aggregate function `MAX_BY` is used. By passing the name of the `tokenizer` table as its first argument and column `token` as its second, it returns the entire row of `tokenizer` in which the token value is maximal. The column values of this row are returned as a `STRUCT`, which means they have to be unnested using function `UNNEST` in order to separate them into their own columns. The full implementation of these steps is shown in Listing 53.

```

38 WITH RECURSIVE
39     base (part_position, position, character, continue) AS
40     (
41         -- Get the rows of the previous bpe iteration whose characters can continue to be merged.
42         SELECT *
43         FROM   bpe
44         WHERE  continue
45     ),
46     bn (part_position, position, continue, character, cluster) AS
47     (
48         -- Determine the clusters that result from merging characters with their neighbour in the part.
49         SELECT part_position,
50                ROW_NUMBER() OVER (PARTITION BY part_position ORDER BY position) AS position,
51                continue,
52                character,
53                character || LEAD(character) OVER (PARTITION BY part_position ORDER BY position) AS cluster
54         FROM   base
55     ),
56     -- (A) For each part_position, only select the clusters for which the token value is maximal.
57     top_rank (part_position, token, cluster) AS
58     (
59         SELECT bn.part_position,
60                UNNEST(MAX_BY(tokenizer, token))
61         FROM   bn
62         JOIN   tokenizer
63         USING (cluster)
64         GROUP BY
65                bn.part_position
66     ),

```

Listing 53: Choosing the best merge for every part in the prompt

(A) Table `tokenizer` is filled with rows before running the inference query. It has two columns named `token` and `cluster`. The latter is used to perform a join with CTE `bn`. The name of column `token` is used as the second argument of aggregate function `MAX_BY`. As a consequence, `MAX_BY` returns the entire row of table `tokenizer` that contains the highest token value of the grouped rows.

The fourth and final CTE of the BPE recursive step is a recursive CTE named `breaks`. Its goal is to select a subset of the position indices from CTE `bn`, *i.e.*, the position indices of the clusters that currently make up the prompt part, by taking into account which merges were chosen in the CTE `top_rank`. The `bn` rows whose character is the first part of a chosen merge are assigned a “length” column value of 2. All other rows, whose potential cluster is not in `top_rank`, get assigned a length of 1. This is realized by performing a `LEFT JOIN` with the CTE `top_rank` on the cluster column. As a consequence, accessing `top_rank.token` returns `NULL` for the rows of the CTE `bn` for which no matching row in `top_rank` exists. Using a `CASE` expression to check whether that is the case allows to set the length value accordingly. Furthermore, the position of `bn` rows whose character is the second part of a chosen merge does not appear in the result set of `breaks`. This is achieved by joining the rows of `bn` with the rows of the previous iteration of `breaks` on the condition that `bn.position` is equal to `breaks.position + breaks.length`. To give an example, let x be the position index of a `bn` row whose character was chosen to be merged with its neighbour, which is stored in the row with `bn.position = x + 1`. This means that CTE `breaks` will add a row with values `breaks.position = x` and `breaks.length = 2` to its result

set. The next iteration of breaks will then add another row for `bn.position = x + 2`, thus skipping over the position with value `x + 1` that belongs to the merged row. The implementation of CTE breaks is shown in Listing 54.

```
64 breaks (part_position, position, length) AS DuckDB
65 (
66 -- Inner base case:
67 SELECT 0::BIGINT AS part_position,
68         0::BIGINT AS position,
69         1 AS length
70 UNION ALL
71 -- Inner recursive member:
72 SELECT bn.part_position,
73        bn.position,
74        CASE WHEN top.token IS NULL THEN 1 ELSE 2 END
75 FROM   breaks
76 JOIN   bn
77 ON     bn.position = breaks.position + breaks.length
78 LEFT JOIN
79        top_rank AS top
80 ON     top.cluster = bn.cluster AND
81        top.part_position = bn.part_position
82 )
```

Listing 54: Choosing the best merge for every part in the prompt

Finally, the `SELECT` clause of the BPE recursive step's `WITH` clause performs the definitive merge. Its implementation is shown in Listing 55. Each row in the CTE `breaks` is joined with a correlated subquery that gets all rows of `bn` whose character values should be merged to a cluster. The condition for this join is that their `part_position` values match and that `bn.position` is within the bounds of the cluster, defined by `breaks.position` and `breaks.position + length`. The characters are then aggregated using function `STRING_AGG`. Additionally, the `SELECT` clause sets for every `part_position` a boolean value that indicates if a merge could be executed. This is determined by performing a `LEFT JOIN` with CTE `top_rank` on column `part_position`, so that `top_rank.token` is `NULL` if no `top_rank` row exists for that `part_position`. In that the case, the value of column `continue` is set to `FALSE`, otherwise to `TRUE`. The `bpe` rows where this value is set to `FALSE` are not selected for the next iteration. All in all, this means that the recursion stops when either the whole part has been merged to a single cluster that can be represented by a token, or when no token exists in the token dictionary for any of the merges that would still be possible.

```

82 -- Select clause of the recursive step
83 SELECT part_position,
84        breaks.position,
85        bn_merge.character,
86        top.token IS NOT NULL AS continue
87 FROM   breaks
88 LEFT JOIN
89        top_rank AS top
90 USING  (part_position)
91 CROSS JOIN LATERAL
92        (
93        -- Perform the chosen merges
94        SELECT STRING_AGG(character, '' ORDER BY position) AS character
95        FROM   bn
96        WHERE  bn.position >= breaks.position
97        AND    bn.position < breaks.position + length
98        AND    bn.part_position = breaks.part_position
99        ) bn_merge
100 WHERE  position > 0

```

Listing 55: `SELECT` clause of the `WITH` clause implementing the BPE recursive step

Listing 55 is the final piece of CTE clusters. Its result set indicates which substrings of the prompt should be represented as a single token. CTE tokens selects their token values from table `tokenizer` and aggregates them to a list (see Listing 56). The result set of this CTE is a single row containing the list of tokens representing the prompt.

```

109 tokens (input_tokens) AS
110 (
111 SELECT ARRAY_AGG(t.token ORDER BY c.part_position, c.position) AS input_tokens
112 FROM   clusters AS c
113 JOIN   tokenizer AS t
114 USING (cluster)
115 ),

```

Listing 56: Representation of the prompt as a list of tokens

3.5.3. Recursively Inferring the Transformer

The implementation of the transformer consists of a nested recursion in the CTE `gpt`. Every iteration of the outer recursion passes a list of tokens that represents the current state of the prompt to the inner recursion, which implements the GPT inference. The result of the inner recursion is a new token, which is added to the list of tokens that will be the input for the next GPT inference. The outer recursion stops if the token returned by the inner recursion is the “end of text” token of value 50256 or when the list of tokens has already reached the desired length, *i.e.*, the original length plus the value of input parameter `threshold`. Listing 57 shows the code for the outer recursion in the CTE `gpt`.

```

115 gpt (prompt_tokens, original_length) AS DuckDB
116 (
117 -- Base Case: select the list of tokens representing the input prompt
118 SELECT input_tokens AS prompt_tokens,
119        LEN(input_tokens) AS original_length
120 FROM   tokens
121 UNION ALL
122 -- Recursive Step: add a token to the prompt
123 SELECT LIST_APPEND(input_tokens, next_token.token),
124        original_length
125 FROM   gpt          -- recursive call
126 CROSS JOIN
127        input_params AS ip
128 CROSS JOIN LATERAL
129        ( <inner WITH clause> ) AS next_token
130 -- Stop the recursion when the prompt has reached the desired length
131 -- or when the "end of text" token was chosen:
132 WHERE  LEN(gpt.prompt_tokens) < gpt.original_length + ip.threshold
133        AND next_token.token <> 50256
134 ),

```

Listing 57: Implementation of the outer recursion in the CTE `gpt`

The `<inner WITH clause>` implements the GPT inference. It feeds the vector embeddings of the tokens into the transformer and calculates its output. This output includes a vector embedding that represents how the transformer predicts the prompt to continue. By calculating the similarity between all available tokens and this prediction, one of the tokens that match the prediction most closely is chosen to be added to the prompt. An overview of all the CTEs that make up this `WITH` clause is shown in Listing 58.

```

1  WITH RECURSIVE DuckDB
2    hparams AS (...),          -- Set the hyperparameters
3    embeddings AS (...),      -- Calculate the vector embedding of each prompt token
4    transformer AS (...),    -- Recursively calculate the output of the 12 blocks of the transformer
5    gpt_prediction AS (...),  -- Get the predicted vector embedding of the prompt continuation
6    ln_f AS (...),          -- Normalize the predicted vector embedding
7    logits AS (...),        -- Calculate the similarity of all available tokens with the prediction
8    softmax_logits AS (...), -- Convert the similarity scores to probabilities between 0 and 1
9    token_ranges AS (...),   -- Define subranges in [0,1] for each top candidate token
10   next_token AS (...)      -- Choose one of the top candidate tokens by drawing a random number
11 SELECT token
12 FROM   next_token

```

Listing 58: Overview of the structure of the `<inner WITH clause>` performing GPT inference to determine which token should be added to the prompt

The first two CTEs of the `<inner WITH clause>` prepare the hyperparameters and inputs for the inference of the Generative Pre-Trained Transformer, whose implementation is presented in Section 3.5.4. CTE `hparams` stores the current number of tokens in the prompt `n_token` and the number of blocks that make up the transformer `n_blocks`, which is set to 12. The second CTE calculates the vector embeddings of the tokens representing the current state of the prompt. As mentioned in Section 3.5.2, table `wte` stores a mapping from tokens to vector embeddings. Furthermore, another table named `wpe` stores the Word Position Embedding. This is a mapping from position indices to vector embeddings. Adding both of these vector embeddings together for every token in the prompt returns their vector embedding.

[19] Vectors are generally represented as arrays throughout this query. The elementwise addition of these vectors is performed by a macro named `vec_add`, which converts their array representations to lists and realizes the operation using list functions. The definitions of all macros used in this query are presented in Section 3.5.4.

```
129 -- Correlated subquery "next_token":
130 WITH RECURSIVE
131   hparams (n_tokens, n_blocks) AS
132   (
133     SELECT LEN(gpt.prompt_tokens) AS n_tokens,           -- correlation with "gpt"
134            12 AS n_blocks
135   ),
136   embeddings (place, values) AS
137   (
138     SELECT tid.place,
139            embedding.values
140   FROM   (
141     SELECT GENERATE_SUBSCRIPTS(gpt.prompt_tokens, 1) - 1 AS place -- correlation with "gpt"
142   ) tid
143   CROSS JOIN LATERAL
144   (
145     -- Calculate the vector embedding of each token (add their WTE and WPE vectors).
146     SELECT vec_add(wte.values, wpe.values) AS values
147   FROM   wte
148   CROSS JOIN
149         wpe
150   WHERE  wte.token = gpt.prompt_tokens[tid.place + 1] -- correlation with gpt
151         AND  wpe.place = tid.place
152   ) embedding
153   ),
```

Listing 59: Excerpt of the `<inner WITH clause>`: Calculating the vector embeddings of each token

3.5.4. Recursively Calculating the Block Outputs

The Generative Pre-Trained Transformer of this query consists of 12 layers, or “blocks”. A CTE named `transformer` recursively performs the calculations of every block, where each block’s output serves as input for the next. The implementation of these calculations will be discussed in Section 3.5.5 onwards. The rest of this CTE’s code is shown in Listing 60. Each row of its result set stores the index of the current block as well as the block’s output values. The base case represents a dummy block with index 0 and the values of the vector embeddings from CTE `embeddings`, which are the input values for the first block.

```

151 transformer (block_num, place, values) AS
152 (
153 -- Base Case: start with the vector embeddings of the current state of the prompt
154 SELECT 0 AS block_num,
155         place,
156         values
157 FROM embeddings
158 UNION ALL
159 (
160 WITH previous (block_num, place, values) AS
161     (
162     -- Select the output of the previous block
163     SELECT *
164     FROM transformer -- recursive call
165     )
166 -- Recursive Step: select the output of the current block
167 SELECT curr.block_id + 1 AS block_num,
168        block_output.*
169 FROM (
170     -- Get the id of the current block
171     SELECT block_num AS block_id
172     FROM previous
173     WHERE block_num < 12
174     LIMIT 1
175     ) curr
176 CROSS JOIN LATERAL
177     ( <transformer blocks> ) block_output
178 )
179 ),

```

DuckDB

Listing 60: Table macro to norm vector values

The implementation of `<transformer blocks>` is realized by yet another nested `WITH` clause that calculates the block outputs. The calculations performed by each block are divided into two steps:

- Multi-Head Self-Attention (Section 3.5.5)
- Feed Forward (Section 3.5.6)

The formulas for these calculations are the same for every block, but the values of their parameters are different. These parameters are stored in a total of 12 tables that have to be prepared before running the query, each holding the parameters for one specific calculation alongside the index of the block for which they are intended. The first 12 CTEs of the nested `WITH` clause only serve the purpose of selecting the parameters for the current block.

```

179 WITH ln_1_b_params (values) AS
180     (
181     SELECT ln_1_b.values
182     FROM ln_1_b
183     WHERE ln_1_b.block = curr.block_id -- correlation with subquery "curr"
184     ),

```

DuckDB

Listing 61: Selecting the parameters of the current block, shown at the example of table `ln_1_b`.

Each block of the transformer performs various calculations on the vector embeddings of the input tokens. For better readability and to avoid code duplication, some of these vector operations are defined

as macros. Their definitions have to be executed before running the GPT inference query and include a set of elementwise vector operations that are implemented using list functions (see Listing 62). `LIST_ZIP` groups elements with the same index together. `LIST_TRANSFORM` then applies a lambda that adds or multiplies the values in each group to the result of `LIST_ZIP`, thus achieving elementwise addition or multiplication of the lists.

```

1 CREATE MACRO vec_add(a1, a2) AS
2     LIST_TRANSFORM(LIST_ZIP(a1::REAL [], a2::REAL []), x -> x[1] + x[2]);
3 CREATE MACRO vec_add3(a1, a2, a3) AS
4     LIST_TRANSFORM(LIST_ZIP(a1::REAL [], a2::REAL [], a3::REAL []), x -> x[1] + x[2] + x[3]);
5 CREATE MACRO vec_mul(a1, a2) AS
6     LIST_TRANSFORM(LIST_ZIP(a1::REAL [], a2::REAL []), x -> x[1] * x[2]);
7 CREATE MACRO vec_mul_add(a1, a2, a3) AS
8     LIST_TRANSFORM(LIST_ZIP(a1::REAL [], a2::REAL [], a3::REAL []), x -> x[1] * x[2] + x[3]);

```

Listing 62: Scalar macro to for elementwise addition and multiplication of lists

Furthermore, a table macro is defined to create a table that holds the mean and the population variance of the values in an array representation of a vector (see Listing 63). The mean is calculated using list function `LIST_AVG`. The formula of the population variance (Equation 3) is realized using list functions `LIST_SUM`, `LIST_TRANSFORM` and `LEN`.

```

10 CREATE MACRO var_pop_mean(values) AS TABLE
11     SELECT a.mean AS mean,
12           LIST_SUM(
13               LIST_TRANSFORM(values::REAL [], x -> (x - a.mean)^2)
14           ) / LEN(values::REAL []) AS variance
15 FROM     (SELECT LIST_AVG(values::REAL []) AS a(mean);

```

Listing 63: Table macro to calculate the mean and population variance of an array of values

3.5.5. Multi-Head Self-Attention

The first step performed in each block of the GPT is called multi-head self-attention. It is a mechanism that allows the word embeddings of the tokens in the prompt to influence each other. This enables the transformer to consider relationships between them, rather than only considering their meaning isolated from any context [15]. The formula for the attention function, as it was originally proposed by Vaswani *et al.* [21], uses three matrices named “Query” (Q), “Key” (K) and “Value” (V):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (4)$$

Before discussing the implementation of this formula, the first step is to calculate the three input matrices Q , K and V . They are obtained by applying a linear transformation to the normalized input vector embeddings of the current block. The normalization occurs in the CTE `mha_norm` (Listing 64). It uses the table macro `var_pop_mean` to calculate the mean and population variance for each vector. Both values are then used in a lambda that is applied to each entry using function `LIST_TRANSFORM`. This is followed by an elementwise multiplication with the array of block parameters stored in table `ln_1_g` and an elementwise addition with the array of block parameters stored in `ln_1_b`.

```

284 mha_norm (place, values) AS DuckDB
285 (
286 SELECT previous.place,
287         vec_mul_add(normed.values, ln_1_g.values, ln_1_b.values) AS values
288 FROM   previous
289 CROSS JOIN LATERAL
290       (
291         SELECT LIST_TRANSFORM(previous.values, x -> (x - mean) / SQRT(variance + 1E-5)) AS values
292         FROM   var_pop_mean(previous.values)      -- table macro
293       ) AS normed
294 CROSS JOIN
295       ln_1_g_params AS ln_1_g
296 CROSS JOIN
297       ln_1_b_params AS ln_1_b
298 ),

```

Listing 64: Normalizing the input vectors of the current block. Both CTEs `ln_1_g_params` and `ln_1_b_params` only have one row with the values for the current block.

As for the linear transformation to obtain matrices Q , K and V , the first step is to multiply the normalized input vectors represented by CTE `mha_norm` with a transformation matrix $W \in \mathbb{R}^{768 \times 2304}$, whose weights are stored in table `c_attn_w`. This is considered a matrix-matrix multiplication, as the set of normalized vectors x_1' to x_n' are treated as row vectors of a matrix $X' \in \mathbb{R}^{n \times 768}$ that is multiplied with W , where n is the current number of tokens in the prompt. Table `c_attn_w` represents matrix W as a set of column vectors w_1 to w_{2304} , which are stored as arrays with 768 entries. Consequently, performing a `CROSS JOIN` of these table representations returns all combinations of row vectors in X' and column vectors in W . The entries of result matrix $M_{qkv} \in \mathbb{R}^{n \times 2304}$ of the matrix-matrix multiplication are obtained by calculating the dot product for each row/column pair (see also the discussion of matrix operations in Section 3.5.4).

$$M_{qkv} = X' \cdot W = \begin{pmatrix} x_1' \\ x_2' \\ \vdots \\ x_n' \end{pmatrix} \cdot (w_1 \ w_2 \ \dots \ w_{2304}) \quad (5)$$

In order to represent the result matrix as a set of row vectors, its entries are grouped by the row index of X' and aggregated into an array of length 2304. This is convenient for the next step of the transformation, which is realized by performing elementwise addition of each row vector with the parameters stored in table `c_attn_b` for the current block. The implementation of the full transformation is shown in Listing 65.

```

298 mha_qkv (place, values) AS DuckDB
299 (
300 SELECT m.place,
301        vec_add(m.values, c_attn_b.values) AS values
302 FROM (
303        -- Matrix-matrix multiplication of the normalized block inputs with transformation matrix W:
304        SELECT mha_norm.place,
305               ARRAY_AGG(
306                   ARRAY_DOT_PRODUCT(mha_norm.values, w.values) ORDER BY w.col_id
307               )::REAL[2304] AS values
308 FROM     mha_norm
309 CROSS JOIN
310         c_attn_w_params AS w
311 GROUP BY
312         mha_norm.place
313 ) m
314 CROSS JOIN
315         c_attn_b
316 ),

```

Listing 65: Linear transformation to obtain all entries for matrices Q , K , and V

The resulting matrix represented by `mha_qkv` holds the values for 12 sets, or “heads”, of matrices Q , K and V stacked next to each other, each with 64 columns. As a consequence, the first 768 entries of each row vector are the row values of the 12 Q matrices, the next 768 entries belong to the 12 K matrices and the remaining entries are that of the 12 V matrices. Multi-head attention means that the attention formula (Equation 4) is calculated for these 12 different sets of Q , K , and V matrices. This approach was proposed in the original transformer architecture [21].

The table representation of matrices Q , K and V is obtained by splitting `mha_norm.values` into 12 sets of 3 row vectors with 64 entries each. This is performed by a CTE named “`qkv_heads`”, which splits the row vectors of the matrix represented by CTE `mha_qkv` into chunks of length 64.

```

316 qkv_heads (head, q_values, k_values, v_values) AS DuckDB
317 (
318 SELECT place,
319        head,
320        values[(g.head * 64 + 1) : (g.head * 64 + 64)] AS q_values,
321        values[(g.head * 64 + 1 + 768) : (g.head * 64 + 64 + 768)] AS k_values,
322        values[(g.head * 64 + 1 + 1536) : (g.head * 64 + 64 + 1536)] AS v_values
323 FROM     mha_norm
324 CROSS JOIN
325         GENERATE_SERIES(0, 11) AS g(head)
326 ),

```

Listing 66: Splitting the matrix represented by `mha_qkv` into three matrices Q , K , and V .

With the table representation of these three matrices at hand, the attention formula can be implemented. The next CTE (Listing 67) calculates for each head the matrix-matrix multiplication QK^T divided by $\sqrt{d_k}$, which is the input for the softmax function in the attention formula (Equation 4). The value d_k is the dimension of Query and Key [21], which is 64 in this case. Just like before, the matrix-matrix multiplication is realized using a join of the matrices’ table representations. Here, `qkv_heads` represents both matrices and is therefore joined with itself. As the attention formula is supposed to be calculated separately for the different sets of matrices, only rows of the same head are combined.

Conveniently, the values of the row vectors of K represented by CTE `qkv_heads` are at the same time the values of the column vectors k_1, \dots, k_n for K^T . The entries of the result matrix can thus once again be obtained by calculating the dot product of each row/column pair.

$$Q \cdot K^T = \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_n \end{pmatrix} \cdot (k_1 \ k_2 \ \dots \ k_n) \quad (6)$$

This time, the entries of the result matrix are not aggregated to row vectors. Therefore, the table representation of the result stores a single entry per table row, alongside its row and column index. That way, each of these entries can simply be divided by $\sqrt{d_k} = \sqrt{64} = 8$.

As a last step before continuing with the calculation of the attention formula, a mask is applied to the result matrix. The intention behind this, as explained in the GPT tutorial [15], is that values obtained from the dot product of a row/column pair that would make a vector embedding depend on tokens that come later in the sequence to 0. Since the next step of the attention formula is to apply the softmax function to scale the values to the range between 0 and 1 with row entries adding up to 1, the masking cannot be applied afterwards without undoing this property. Instead, a very large number is subtracted from the affected entries where the column index of K^T is higher than the row index of Q . This will translate to a value of 0 after applying the softmax function.

```

326 sm_input (head, row_id, col_id, value) AS
327 (
328 SELECT head,
329         q.place AS row_id,
330         k.place AS col_id,
331         ARRAY_DOT_PRODUCT(q.q_values, k.k_values) / 8 + CASE WHEN k.place > q.place THEN -1E10 ELSE 0 END
332         AS value
333 FROM   qkv_heads AS q
334 JOIN   qkv_heads AS k
335 USING (head)

```

Listing 67: Calculating the input for softmax

Before applying the softmax function to each row of the masked result matrix represented by CTE `sm_input`, its maximum value is subtracted from each row entry. This is meant to help with numerical stability [15].

```

335 sm_diff (head, row_id, col_id, diff) AS
336 (
337 SELECT head, row_id, col_id,
338         value - MAX(value) OVER (PARTITION BY head, row_id) AS diff
339 FROM   sm_input
340 ),

```

Listing 68: Subtracting the maximum row value from each row entry

For each value x in a row with index i , the softmax result is calculated using the following formula [15]:

$$\text{softmax}(x)_i = \left(\frac{e^{(x_i)}}{\sum_j e^{(x_j)}} \right) \quad (7)$$

CTE `sm_exp` calculates all $e^{(x_i)}$ values. CTE `softmax` then divides the values by the sum, which is calculated using aggregate function `SUM` while partitioning the rows by their head and row index. The implementation of these CTEs is shown in Listing 69.

```

340 sm_exp (head, row_id, col_id, e) AS
341 (
342 SELECT head, row_id, col_id,
343        EXP(diff) AS e
344 FROM   sm_diff
345 ),
346 softmax (head, row_id, col_id, value) AS
347 (
348 SELECT head, row_id, col_id,
349        e / SUM(e) OVER (PARTITION BY head, row_id) AS value
350 FROM   sm_exp
351 ),

```

Listing 69: Calculating the softmax values

The final step for calculating the attention is multiplying the result of the softmax function with V , the third matrix represented by table `qkv_heads`. This matrix-matrix multiplication is more complicated than the ones discussed before, because V is not represented as a set of column vectors, but as a set of row vectors. Obtaining all row/column pairs using a straightforward cross join is therefore not possible. Instead, the matrix-matrix multiplication is viewed from the perspective of a single entry in `softmax`. To calculate the dot products of every row in `softmax` with every column in V , each entry in `softmax` is multiplied with all those entries in V whose row index is equal to the softmax entry's column index. This can be realized using a `JOIN`. Afterwards, some of these products have to be added up, with each sum returning one entry of the result matrix. This type of calculation can be realized using a `GROUP BY` clause. Combining this idea with the use of row vectors and elementwise list operations results in the implementation shown in Listing 70.

```

351 attention_heads (head, place, values) AS
352 (
353 SELECT s.head,
354        s.row_id AS place,
355        LIST_REDUCE(
356          ARRAY_AGG(vec_mul(LIST_RESIZE([], ::REAL [], 64), s.value), v.v_values)),
357          (x, y) -> vec_add(x, y)
358        ) AS values
359 FROM   softmax AS s
360 JOIN   qkv_heads AS v
361 ON     s.head = v.head AND
362        s.col_id = v.place
363 GROUP BY
364        s.head, s.row_id
365 ),

```

Listing 70: Calculate for every head the final result of the attention formula

The `JOIN` pairs each matrix entry stored in `softmax` with exactly one row vector of matrix V , namely with the one whose row index (“place”) is equal to the column index of the softmax value. In the `SELECT` clause, this softmax value is then multiplied onto every element in the array representation of said row vector. This is achieved by using function `LIST_RESIZE` to create a list of the same length (64 entries) whose entries are all set to the softmax value, before then performing elementwise multiplication

using macro `vec_mul`. The `GROUP BY` clause divides the result set into groups of table rows of the same head where the softmax values have the same `row_id`. Aggregate function `ARRAY_AGG` collects all the lists resulting from the elementwise multiplications within that group in an array. Then, the function `LIST_REDUCE` makes all these lists into one by performing elementwise addition. The single list returned by this function represents one row vector of the result matrix of the matrix-matrix multiplication. This means that each group (with `softmax.row_id = i`) returns the result of all dot products that involve row i of the matrix represented by CTE `softmax`, aggregated to an array. This is also illustrated in Table 29.

Group	softmax value	row vector	Aggregate input
row_id = 1	s_{11}	$[r_{11}, \dots, r_{1n}]$	$[s_{11} \cdot r_{11}, \dots, s_{11} \cdot r_{1n}]$
	s_{12}	$[r_{21}, \dots, r_{2n}]$	$[s_{12} \cdot r_{21}, \dots, s_{12} \cdot r_{2n}]$
	\vdots	\vdots	\vdots
	s_{1n}	$[r_{n1}, \dots, r_{nn}]$	$[s_{1n} \cdot r_{n1}, \dots, s_{1n} \cdot r_{nn}]$
row_id = 2	s_{21}	$[r_{11}, \dots, r_{1n}]$	$[s_{21} \cdot r_{11}, \dots, s_{21} \cdot r_{1n}]$
	s_{22}	$[r_{21}, \dots, r_{2n}]$	$[s_{22} \cdot r_{21}, \dots, s_{22} \cdot r_{2n}]$
	\vdots	\vdots	\vdots
	s_{2n}	$[r_{n1}, \dots, r_{nn}]$	$[s_{2n} \cdot r_{n1}, \dots, s_{2n} \cdot r_{nn}]$
\vdots	\vdots	\vdots	\vdots

Table 29: Illustration of how softmax values s_{ij} and row vectors of V are joined and grouped for each head, and thus which resulting lists are reduced to a row of the result matrix

Now that the attention is calculated for all 12 heads, all their results have to be merged. This is accomplished by stacking the result matrices next to each other to rows of length $12 * 64 = 768$, which is the original vector dimension. CTE `merge_heads` (Listing 71) groups the row vectors of all heads by their row index “place” and aggregates them to an array of lists. This array is then flattened by function `FLATTEN` to a simple array with 768 entries.

```

365 merge_heads (place, values) AS
366 (
367 -- Stack the 12 attention result matrices horizontally
368 SELECT place,
369         FLATTEN(ARRAY_AGG(values ORDER BY head))::REAL[768] AS values
370 FROM attention_heads
371 GROUP BY
372         place
373 ),

```

Listing 71: Merge the result of all heads

Let result set of CTE `merge_heads` (Listing 71) be the table representation of a matrix $A \in \mathbb{R}^{n \times 768}$ with row vectors a_1 to a_n . To obtain the final result of the multi-head self-attention step, one last linear transformation is applied to A . Like before, this includes a multiplication with a transformation matrix, in this case $W \in \mathbb{R}^{768 \times 768}$, followed by an elementwise addition of every row with a vector $b \in \mathbb{R}^{768}$. Additionally, the original inputs for the multi-head attention step, which were previously described as vectors x_1 to x_n , are added to the result. This step was also proposed in the original paper on transformers [21]. Equation 8 shows how the final result matrix R_{mha} of the multi-head self-attention step is calculated.

$$\begin{aligned}
 R_{\text{mha}} &= A \cdot W + B + X \\
 &= \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \cdot (w_1 \ w_2 \ \dots \ w_{768}) + \begin{pmatrix} b \\ b \\ \vdots \\ b \end{pmatrix} + \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}
 \end{aligned} \tag{8}$$

The implementation of this equation is nearly identical to Listing 65, other than the last part of adding the inputs and the different dimensions. The column vectors of the matrix W are stored in table `c_proj_w`, the parameters for the subsequent elementwise addition with every row vector are stored in table `c_proj_b`.

```

373 mha (place, values) AS (
374   SELECT m.place,
375          vec_add3(m.values, b.values, previous.values) AS values
376   FROM (
377     SELECT attn.place,
378            ARRAY_AGG(
379              ARRAY_DOT_PRODUCT(attn.values, w.values) ORDER BY w.place
380            )::REAL[768] AS values
381     FROM merge_heads AS attn
382     CROSS JOIN
383           c_proj_w AS w
384     GROUP BY
385           attn.place
386   ) m
387   CROSS JOIN
388     c_proj_b AS b
389   JOIN previous
390   USING (place)
391 ),

```

Listing 72: Final linear transformation to obtain the result of the multi-head self-attention step

3.5.6. Feed Forward

The feed forward step performs a row of calculations on the result of the multi-head self-attention step before the resulting vectors are passed to the next block of the transformer. These calculations can be summarized as the following formula [19], whose implementation will be explained step by step:

$$\text{FFN}(R) = R + \text{c_proj}(\text{GELU}(\text{c_fc}(\ln_2(R)))) \tag{9}$$

Just like the multi-head attention step, the feed forward step begins with normalizing its input and then performing a linear transformation. The transformation projects the values into a higher dimension

that is 4 times the dimension of the vector embeddings. This is achieved by multiplying them with a matrix $W \in \mathbb{R}^{768 \times 3072}$ consisting of column vectors w_1 to w_{3072} .

$$X' \cdot W = \begin{pmatrix} x_1' \\ x_2' \\ \vdots \\ x_n' \end{pmatrix} \cdot (w_1 \ w_2 \ \dots \ w_{3072}) \quad (10)$$

The implementation of these steps is largely the same as in Listing 64 and Listing 65 of the multi-head self-attention step. It uses parameters stored in tables `ln_2_g` and `ln_2_b` for the normalization and transformation parameters from tables `mlp_c_fc_w` and `mlp_c_fc_b`.

```

392 ffn_norm (place, values) AS
393 (
394 SELECT mha.place,
395        vec_mul_add(normed.values, ln_2_g.values, ln_2_b.values) AS values
396 FROM   mha
397 CROSS JOIN LATERAL
398        (
399         SELECT LIST_TRANSFORM(mha.values, x -> (x - mean) / SQRT(variance + 1E-5)) AS values
400         FROM   var_pop_mean(mha.values)      -- table macro
401        ) AS normed
402 CROSS JOIN
403        ln_2_b_params AS ln_2_b
404 CROSS JOIN
405        ln_2_g_params AS ln_2_g
406 ),
407 ffn_fc AS (
408 SELECT m.place,
409        vec_add(m.values, b.values) AS values
410 FROM   (
411        SELECT ffn_norm.place,
412               ARRAY_AGG(
413                ARRAY_DOT_PRODUCT(ffn_norm.values, w.values) ORDER BY w.place
414                )::REAL[3072] AS values
415        FROM   ffn_norm
416        CROSS JOIN
417                mlp_c_fc_w AS w
418        GROUP BY
419                ffn_norm.place
420        ) m
421 CROSS JOIN
422        mlp_c_fc_b_params AS b
423 ),

```

Listing 73: Normalization and linear transformation in the feed forward step

The next step of implementing the feed forward formula (Equation 9) is the GELU function. It is approximated using the following formula [19]:

$$\text{GELU}(x) \approx 0.5x \cdot (1 + \tanh(0.797884 \cdot (x + 0.044715x^3))) \quad (11)$$

This formula is realized as a lambda, which is then applied to each element of the vectors using list function `LIST_TRANSFORM`. This is shown in Listing 74.

```

423 ffn_gelu (place, values) AS DuckDB
424 (
425 SELECT place,
426 LIST_TRANSFORM(
427 values::REAL [],
428 x -> 0.5 * x * (1 + tanh(0.797884560802 * (x + 0.044715 * x^3)))
429 )::REAL [3072] AS values
430 FROM ffn_fc
431 ),

```

Listing 74: Apply the GELU function approximation to each value

Lastly, the final linear transformation is applied to the GELU result. Just like at the end of the multi-head self-attention step, the input values of the feed forward step are added to its result. The implementation for this, as shown in Listing 75, mirrors that of Listing 72.

```

431 ffn (place, values) AS DuckDB
432 (
433 SELECT m.place,
434 vec_add3(m.values, b.values, mha.values) AS values
435 FROM (
436 SELECT ffn_gelu.place,
437 ARRAY_AGG(
438 ARRAY_DOT_PRODUCT(ffn_gelu.values, w.values) ORDER BY w.place
439 )::REAL [768] AS values
440 FROM ffn_gelu
441 CROSS JOIN
442 mlp_c_proj_w AS w
443 GROUP BY
444 ffn_gelu.place
445 ) m
446 CROSS JOIN
447 mlp_c_proj_b_params AS b
448 JOIN mha
449 USING (place)
450 )

```

Listing 75: Complete the feed forward step

Listing 75 shows the last CTE of the `WITH` clause implementing the transformer blocks first discussed in Section 3.5.4. Its `SELECT` clause is as simple as selecting all rows of CTE `ffn`, as they represent the block's output.

```

450 SELECT * DuckDB
451 FROM ffn

```

Listing 76: `SELECT` clause of the `WITH` clause implementing `<transformer blocks>`

3.5.7. Choosing the Next Token

After the inner recursion has finished calculating the output of each block, CTE `gpt_prediction` (Listing 77), which is a sibling of CTE `transformer`, selects those parts of the output that are required to choose the next token to append to the prompt. The result set of CTE `transformer` includes the outputs of every block of the transformer. However, in order to choose the next token, only the vector with the highest position index from the output of the final block is needed. This vector represents the vector embedding of the token predicted by GPT to continue the prompt. Its index is calculated

from the number of tokens stored in `hparams.n_tokens`, while the number of the last block is stored in `hparams.n_blocks`.

```
455 gpt_prediction (values) AS DuckDB
456 (
457   SELECT transformer.values
458   FROM   transformer
459   JOIN   hparams AS hp
460   ON     transformer.block_num = hp.n_blocks AND
461         transformer.place = hp.n_tokens - 1
462 ),
```

Listing 77: Get the vector embedding of the prompt continuation from the transformer output

The vector embedding representing the prediction is then normalized using parameters from tables `ln_f_g` and `ln_f_b`. Other than the different parameters, the method of normalization is the same as in Listing 64 or Listing 73.

```
462 ln_f (values) AS DuckDB
463 (
464   SELECT vec_mul_add(normed.values, ln_f_g.values, ln_f_b.values) AS values
465   FROM   gpt_prediction
466   CROSS JOIN LATERAL
467     (
468       SELECT LIST_TRANSFORM(gpt_prediction.values, x -> (x - mean) / SQRT(variance + 1E-5)) AS values
469       FROM   var_pop_mean(gpt_prediction.values) -- table macro
470     ) AS normed
471   CROSS JOIN
472     ln_f_b_params AS ln_f_b
473   CROSS JOIN
474     ln_f_g_params AS ln_f_g
475 ),
```

Listing 78: Normalizing the GPT prediction

The next goal is to find the `input_params.top_n` candidate tokens in the Word Token Embedding table `wte` whose vector embeddings most closely match the GPT prediction for the next token. Their similarities are determined by calculating the dot product of the prediction vector with each of the 50257 vector embeddings stored in `wte` (see Listing 79). The tokens with the `top_n` highest dot product results, or “logits”, are chosen as candidates for the prompt continuation. This is realized by ordering the logits in a descending order. The output modifier `LIMIT` is then used to only select the rows with the `top_n` scores.

Ideally, the value of `input_params.top_n` would be used in combination with the `LIMIT` modifier. Unfortunately, this causes an error in DuckDB, as correlated columns are not supported in `LIMIT`. Therefore, the number must be modified in the code if a different number of top candidates should be considered.

```

475 logits (token, product) AS
476 (
477 SELECT token,
478     ARRAY_DOT_PRODUCT(ln_f.values, wte.values) AS product
479 FROM ln_f
480 CROSS JOIN
481     wte
482 ORDER BY
483     value DESC
484 LIMIT 5 -- top_n
485 ),

```

Listing 79: Determine the top_n best token candidates for the prompt continuation

In order to choose between the token candidates, their similarity scores produced by the dot product are converted to probabilities between 0 and 1 using the softmax function. Additionally, the hyperparameter “temperature” is used in the calculation of these probabilities. It affects how likely the program is to choose a token that is not the top candidate [19].

Like in Section 3.5.5, the maximum input value is subtracted from each value of the input. The result of the function itself is once again calculated using Equation 7, which was already implemented in Listing 69.

```

485 softmax_logits (token, softmax) AS
486 (
487 SELECT si.token,
488     (e / SUM(e) OVER ()) AS softmax
489 FROM (
490     SELECT token,
491         (l.product - MAX(l.product) OVER ()) / ip.temperature AS x,
492         EXP(x) AS e
493     FROM logits AS l
494     CROSS JOIN
495         input_params AS ip
496     ) sm_exp
497 ),

```

Listing 80: Use the softmax function to convert the similarity scores to probabilities between 0 and 1

These softmax results are then used to define for each token a subrange in $[0, 1]$. The length of each subrange is equal to the token’s softmax value, which means all ranges have a combined length of 1. As shown in Listing 81, they are defined by setting the upper bound of the subrange (“high”) as the sum of the token’s own softmax result and all softmax values that are lower. The lower bound (“low”) is obtained by subtracting the softmax value from the upper bound.

```

503 token_ranges (token, high, low) AS
504 (
505 SELECT token,
506     SUM(softmax) OVER (ORDER BY softmax) AS high,
507     high - softmax AS low,
508 FROM softmax_logits
509 ),

```

Listing 81: Define the subranges of $[0, 1]$ for each candidate token

Finally, to choose one of the tokens, a random number is drawn. The token for which the random number is in between its low and high range limits is selected as the prompt continuation. The CTE realizing this, named `next_token` (see Listing 82), is the last CTE of subquery `next_token` in the CTE `gpt`. It is thus followed by the nested `WITH` clause's `SELECT` clause, which simply selects all rows from CTE `next_token`. This concludes the discussion of the code in the CTE `gpt`.

```
507 next_token (token) AS
508 (
509   SELECT range.token
510   FROM (
511     SELECT RANDOM AS rnd
512     ) r
513   CROSS JOIN
514     token_ranges AS range
515   WHERE rnd >= low AND
516         rnd < high
517 )
518 -- Select clause of the nested With clause in the CTE "gpt"
519 SELECT token
520 FROM next_token
```

DuckDB

Listing 82: Draw a random number in $[0, 1)$ and choose the token whose subrange includes that number

3.5.8. Query Output

After the recursive CTE `gpt` has finished adding the desired number of tokens to the prompt, the result of its final iteration is obtained by selecting the list of tokens whose length is equal to `gpt.original_length` plus the value of input parameter “`threshold`”. All that remains for the query to do is to map this list of tokens back to a string.

Each token in the list can be mapped to a cluster of characters using table `tokenizer`. As discussed in Section 3.5.2, the clusters stored in this token dictionary represent some characters as symbols. The mapping between the byte encoding of characters and their representation in the token dictionary is stored in table `encoder`. After selecting the byte encoding of each character or symbol in the clusters represented by the output tokens, these byte encodings can later be converted to characters. This first requires a table where every row stores a single character alongside its position index. This is realized in the CTE `output_clusters` (Listing 83) by using function `REGEXP_SPLIT_TO_ARRAY` to receive an array of all characters in the clusters, which can then be unnested. The `SELECT` clause of this CTE makes use of the fact that DuckDB supports columns referencing aliases of sibling columns [3].

```

525 output_clusters (token_id, chars, character, char_position) AS DuckDB
526 (
527 SELECT token_id,
528         tokenizer.cluster,
529         REGEXP_SPLIT_TO_ARRAY(tokenizer.cluster, '') AS chars,
530         UNNEST(chars) AS character,
531         GENERATE_SUBSCRIPTS(chars, 1) AS char_position
532 FROM   input_params AS ip
533 JOIN   gpt
534 ON     LEN(gpt.prompt_tokens) = gpt.original_length + ip.threshold
535 CROSS JOIN LATERAL
536       (
537         SELECT UNNEST(gpt.prompt_tokens) AS token,
538                GENERATE_SUBSCRIPTS(gpt.prompt_tokens, 1) AS token_id
539       ) n
540 JOIN   tokenizer
541 USING (token)
542 ),

```

Listing 83: Map each token of the result list to a cluster and unnest them

Finally, by joining CTE `output_clusters` with table `encoder` using their character value, the characters' byte encodings can be obtained. They are then converted to the corresponding character using text function `CHR`. Aggregating the resulting characters to a string, ordered by their `token_id` and position in the cluster, returns the final output string.

```

542 output (response) AS DuckDB
543 (
544 SELECT STRING_AGG(CHR(encoder.byte::INT), '' ORDER BY token_id, char_position) AS response
545 FROM   output_clusters
546 JOIN   encoder
547 USING (character)
548 )
549 SELECT *
550 FROM   output;

```

Listing 84: Convert the byte encodings provided by table `encoders` to characters and aggregate them to the final output

This marks the end of the DuckDB implementation of the GPT inference query, adapted from [19]. For the exemplary prompt `'Happy New Year! I wish you'` and a `top_n` value of 1, the query completes the sentence as illustrated in Table 30.

response
'Happy New Year! I wish you all the best for the new year!'

Table 30: Exemplary GPT inference query output

3.6. Performance Evaluation

3.6.1. Effects of Readability Optimizations and CTE Materialization

The result of running `EXPLAIN ANALYZE` on the original PostgreSQL query and the initial version that resulted from porting it to DuckDB are presented in Table 31. Furthermore, runtimes obtained after materializing the Common Table Expressions (as discussed in Section 1.4) in the outer `WITH` clause and after applying the changes to improve readability are examined. These times are impacted by the input parameter `threshold`, which indicates how many tokens are added to the prompt. Since the runtime of any version of the inference query to complete a full sentence is relatively long (over 1 minute), the performance is measured for `threshold = 1`. It should be noted that even though this setting leads to exactly one token being added to the prompt, the resulting measurements do not reflect the runtimes for inferring GPT exactly once, but for inferring it twice. This is because the `WHERE` condition that determines when the outer recursion of the CTE `gpt` stops depends on the value of the chosen token to continue the prompt. If it is the “end of text” token, the row is discarded and the recursion ends. As a consequence, the inner recursion, which calculates the GPT output and then chooses the next token based on its prediction, must be executed before the condition to halt the outer recursion can be evaluated.

	PostgreSQL	DuckDB (initial)	DuckDB (materialized)	DuckDB (readability)
Time	7.46s	33.34s	33.12s	32.68s

Table 31: Comparison of the averaged runtime of different versions of the GPT inference query (the original, ported to DuckDB, with materializing the CTEs of the outer `WITH` clause, plus optimized for readability) to add one token to the prompt `'Happy New Year! I wish you'`

Any DuckDB version of the GPT inference query considered thus far is considerably slower than the original PostgreSQL query. Materializing the CTEs of the outer `WITH` clause shows no significant effect, as the query’s performance remains very similar to the initial DuckDB version. An improved performance would be expected when materializing CTEs that are referenced multiple times throughout the query, especially if these references occur in the recursion (which is observed to impact the runtime in Section 2.6). However, the only outer CTE that is referenced in the recursive step of `gpt` is the one that stores a single row of input parameters. This does not represent an expensive evaluation where preventing its repeated computation provides significant benefits. It is also the only CTE of the outer `WITH` clause that is referenced multiple times. Meanwhile, some CTEs of the innermost `WITH` clause are promising candidates for CTE materialization. Especially the CTE `heads`, which is joined with itself, fulfills the criteria of a known scenario where DuckDB evaluates the CTE for each reference rather than materializing it automatically [3]. However, trying to materialize the CTEs of the inner `WITH` clause can lead to an internal error for the current query structure. This issue is resolved once the query optimization discussed in Section 3.6.2 is applied. Consequently, CTE materialization of the innermost CTEs is discussed in the subsequent section.

3.6.2. Performance Optimizations

A simple but effective measure to improve the performance is to remove the correlation between the innermost `WITH` clause in the CTE `transformer` (Listing 60) and the other subquery in the CTE’s `FROM` clause. This correlation exists so that the innermost CTEs can access the current `block_id`, which is required to get the parameters of the current transformer block. By defining the subquery that determines the `block_id` as an additional CTE in the innermost `WITH` clause, so that the other innermost CTEs can reference this new CTE instead of the correlated subquery, significantly improves the performance (Table 32). Listing 85 highlights the described changes that are applied to the recursive step of CTE `transformer` in order to decorrelate its subqueries.

```

1  -- Excerpt of the recursive step in CTE "transformer":
2  SELECT ...
3  FROM (
4      -- Get the id of the current block
5      SELECT block_num AS block_id
6      FROM previous
7      WHERE block_num < 12
8      LIMIT 1
9  ) curr
10 CROSS JOIN -- NEW: no lateral join
11 (
12     WITH q AS
13     (
14         SELECT block_num AS block_id -- NEW: Repeat subquery "curr" as "q"
15         FROM previous
16         WHERE block_num < 12
17         LIMIT 1
18     ),
19     ln_1_b_params (values) AS
20     (
21         SELECT ln_1_b.values
22         FROM ln_1_b, q -- NEW: join with "q"
23         WHERE ln_1_b.block = q.block_id -- NEW: reference "q", not "curr"
24     ),
25     ...

```

Listing 85: In the CTE transformer (Listing 60), remove the subquery correlation by repeating the subquery “curr” as a CTE in the innermost **WITH** clause to improve performance

The runtimes obtained from running **EXPLAIN ANALYZE** on the original PostgreSQL inference query and the decorrelated DuckDB version are presented in Table 32. They are compared for adding one token to the prompt and for adding eight tokens, which completes the sentence for the exemplary prompt.

	PostgreSQL	DuckDB (decorrelated)
threshold = 1	7.46s	6.87s
threshold = 8	87.05s	44.37s

Table 32: Comparison of the averaged runtime of different versions of the GPT inference query to add a number of tokens to prompt: 'Happy New Year! I wish you'

With this change alone, the DuckDB query suddenly slightly outperforms the PostgreSQL original for a threshold of 1. This slight advantage adds up when the number of outer iterations indicated by the threshold is increased. For a threshold of 8, the runtime of the DuckDB query is only approximately half of that of the original PostgreSQL query.

Furthermore, decorrelating the innermost **WITH** clause allows materializing its CTEs. In order to observe the effect on the evaluation of the query, a DuckDB Python API is set up to run the query. This allows to use user-defined functions (UDFs) created from Python functions in the query [3]. This approach is not used to realize, for example, the elementwise vector operations, because it leads to a worse performance, most likely due to overhead caused by context switching [17]. However, UDFs can be used to verify that CTEs are not evaluated more often than necessary. To that end, a simple Python function is defined that takes a value, adds it to a list that was previously defined as a global variable, and returns the unchanged input value. From this function, an UDF is created (see Listing 86). As a side

effect of calling this function in the query, the length of the list defined as a global variable increases by 1. After the query is executed, the length of this list is printed in the terminal. By applying the UDF to the index column place of the innermost CTEs, the list's length indicates how many rows were computed for this CTE during the query evaluation.

```

1 values = []
2 def showvals(value):
3     global values
4     values.append(value)
5     return value
6
7 # Create UDF from Python function
8 con = duckdb.connect()
9 con.create_function("showvals", showvals, [BIGINT], INTEGER)

```

Listing 86: Creating an UDF from a Python function that adds its argument to a global list

The number of computed rows is compared to the number that is expected if the CTE is only evaluated once per iteration (see Table 33). For a single inference, the expected number of computed CTE rows is equal to the number of blocks in the transformer ($n_blocks = 12$) multiplied by the CTE's number of rows. The formulas for calculating the number of rows are shown in Table 33. All of them depend on the current number of tokens in the prompt (n_tokens). Some also depend on the number of heads used in the multi-head attention step ($n_heads = 12$). Since $threshold = 1$ means that GPT is inferred twice and one token is added to the prompt after each inference, this calculation must be performed a second time with n_token increased by 1. After the tokenization is completed, the exemplary prompt is represented as a list of seven tokens. This means for the example of the CTE `mha_norm`, which has as many rows as there are tokens in the current state of the prompt, that the expected number of computed rows is $n_blocks * 7 + n_blocks * 8 = 12 * 7 + 12 * 8 = 180$. The expected number for the other CTEs, alongside the number obtained from using the UDF before and after materializing the CTE, are presented in Table 33.

CTEs	Number of Rows	Expected	Without Innermost CTE Materialization	After Materializing the CTE	Decr. by Factor
<code>mha_norm</code>	n_tokens	180	1080	180	6
<code>heads</code>	$n_heads * n_tokens$	2160	12960	2160	6
<code>sm_input, sm_diff, sm_exp, softmax</code>	$n_heads * n_tokens^2$	16272	32544	16272	2
<code>attention mha</code>	n_tokens	180	360	180	2
<code>ffn_norm, ffn_a, ffn</code>	n_tokens	180	180	180	1

Table 33: The expected and measured number of rows computed for every CTE in the innermost `WITH` clause (for $threshold = 1$ and an initial number of tokens of 7) before and after materializing the CTE. The expected number is calculated by multiplying the number of rows by $n_blocks = 12$, once with $n_tokens = 7$ and once with $n_tokens = 8$.

Table 33 shows that after materializing the CTEs of the innermost `WITH` clause, their number of computed rows is equal to the expected number. Without CTE materialization, the number is up to 6 times higher. This demonstrates that CTE materialization ensures that they are not computed more often than is necessary. Now that there is confirmation that the CTEs of the transformer implementation

are no longer evaluated more often than required, the runtime is measured once more using `EXPLAIN ANALYZE`. As illustrated in Table 34, the runtime for `threshold = 1` has once again slightly decreased, leading to an even bigger speedup for `threshold = 8` than before.

	DuckDB (decorrelated)	DuckDB (decorr. & innermost CTE mat.)
<code>threshold = 1</code>	6.87s	5.85s
<code>threshold = 8</code>	44.37s	37.79s

Table 34: Comparison of the averaged runtime of the decorrelated DuckDB inference query and the version that additionally applies CTE materialization of the innermost CTEs to add a number of tokens to prompt: 'Happy New Year! I wish you'

3.6.3. Conclusion

Although porting the query to DuckDB initially led to a worse performance, removing a single decorrelation sufficed to allow it to outperform the original PostgreSQL query. This effect is only increased by applying CTE materialization. While the difference between the runtimes is only less than 2 seconds for adding a single token to the exemplary prompt, this difference becomes increasingly large as the query adds more tokens to the prompt.

Aside from optimizations that relate to differences between the database management systems, there are changes that could be made to the logic or structure of the queries in order to improve the performance. As mentioned in Section 3.1, the GPT inference algorithm Quassnoi realized as a PostgreSQL query, which was adapted to DuckDB for this thesis, is a simplified one [15]. Some improvements that may lead to a better performance are not included for the sake of simplicity. For example, the queries recalculate the vector embeddings of all tokens in the prompt before every GPT inference. Instead, the original list of tokens representing the input prompt could be calculated only once, before the nested recursion. Changes of that kind are not specific to any database management system and can thus be applied to both the PostgreSQL and the DuckDB query.

Conclusion

The chosen PostgreSQL queries were successfully ported to DuckDB. This process not only revealed a number of differences between the two database management systems related to their supported functions and data types, but also to their differing approaches to query evaluation.

In addition to the necessary changes to adapt the query to the different behavior of some functions or operators, porting the GIF decoder query required re-examining the best suited data type of the input data. Consequently, new implementations of the set of base operations that lay the foundation of the query had to be determined for the new input data type. For the GPT inference query, list functions were used to realize a multitude of operations performed on vectors. The resulting SQL code of each DuckDB query was optimized for readability by reducing the number of nested code blocks, hiding complex expressions that are repeated throughout the query in macro definitions and other measures to improve readability. The optimized code was presented by explaining the subgoals that are realized in order to solve the problem and how each of them is implemented.

When evaluating the query performance, it was shown that optimizations to improve readability cannot be fully separated from optimizations that improve performance. Some readability optimizations also resulted in an improved performance. Both the GIF decoder and the GPT inference query, which contain a complex recursion that cannot be parallelized, initially performed worse than the original query after being ported to DuckDB. For the GIF decoder query, the different rules around CTE materialization were observed to have an impact on the performance of the recursion. Additionally, a single operation in the recursion was identified to contribute to the longer runtime. It was replaced by an alternative implementation that was observed to be much more efficient. These optimizations combined with an effort to reduce the use of correlated subqueries significantly improved the performance of the DuckDB query, although it still falls behind that of the original query. During the performance optimization of the GPT inference query, decorrelating subqueries in recursive CTEs made an even greater impact. This measure alone allowed the DuckDB query to outperform the original PostgreSQL query. Applying CTE materialization in the inner recursion further improved the query performance.

Bibliography

- [1] Y. Silva, I. Almeida, and M. Queiroz, “SQL: From Traditional Databases to Big Data,” 2016, pp. 413–418. doi: [10.1145/2839509.2844560](https://doi.org/10.1145/2839509.2844560).
- [2] M. Raasveldt and H. Mühleisen, “DuckDB: an Embeddable Analytical Database,” in *Proceedings of the 2019 International Conference on Management of Data*, in SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1981–1984. doi: [10.1145/3299869.3320212](https://doi.org/10.1145/3299869.3320212).
- [3] “DuckDB Documentation.” [Online]. Available: <https://duckdb.org/docs/>
- [4] D. D. Chamberlin and R. F. Boyce, “SEQUEL: A structured English query language,” in *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, in SIGFIDET '74. Ann Arbor, Michigan: Association for Computing Machinery, 1974, pp. 249–264. doi: [10.1145/800296.811515](https://doi.org/10.1145/800296.811515).
- [5] D. Fahland *et al.*, “Declarative versus Imperative Process Modeling Languages: The Issue of Understandability,” 2009, pp. 353–366. doi: [10.1007/978-3-642-01862-6_29](https://doi.org/10.1007/978-3-642-01862-6_29).
- [6] R. C. Martin and J. O. Coplien, *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ [etc.]: Prentice Hall, 2009. [Online]. Available: https://www.amazon.de/gp/product/0132350882/ref=oh_details_o00_s00_i00
- [7] “PostgreSQL Documentation.” [Online]. Available: <http://www.postgresql.org/docs/15/>
- [8] Quassnoi, “Happy New Year: GIF decoder in SQL,” Dec. 2018, [Online]. Available: <https://explainextended.com/2018/12/31/happy-new-year-10/>
- [9] E. S. Raymond and M. Flickinger, “What's In A GIF,” [Online]. Available: <https://giflib.sourceforge.net/whatsinagif/index.html>
- [10] Lee, [Online]. Available: https://pixabay.com/?utm_source=link-attribution&utm_medium=referral&utm_campaign=image&utm_content=2558877
- [11] B. Nema, “Effective Variations on Opened GIF Format Images,” *IJCSNS*, vol. 8, p. 70, 2008.
- [12] C. Incorporated, “GIF Graphics Interchange Format: A standard defining a mechanism for the storage and transmission of bitmap-based graphics information.” 1987.
- [13] R. Wiggins, H. Davidson, H. Harnsberger, J. Lauman, and P. Goede, “Image File Formats: Past, Present, and Future1,” *Radiographics : a review publication of the Radiological Society of North America, Inc*, vol. 21, pp. 789–798, 2000, doi: [10.1148/radiographics.21.3.g01ma25789](https://doi.org/10.1148/radiographics.21.3.g01ma25789).
- [14] Welch, “A Technique for High-Performance Data Compression,” *Computer*, vol. 17, no. 6, pp. 8–19, 1984, doi: [10.1109/MC.1984.1659158](https://doi.org/10.1109/MC.1984.1659158).
- [15] J. Mody, “GPT in 60 lines of NumPy,” 2023, [Online]. Available: <https://jaykmody.com/blog/gpt-from-scratch/#what-is-a-gpt>
- [16] Compuserve, “Graphics Interchange Format (sm), Version 89a.” [Online]. Available: <https://www.w3.org/Graphics/GIF/spec-gif89a.txt>
- [17] D. Hirn, “New Compilation Methods for Complex User-Defined Functions,” 2024.
- [18] A. SHAIKHHA, M. DASHTI, and C. KOCH, “Push versus pull-based loop fusion in query engines,” *Journal of Functional Programming*, vol. 28, p. e10, 2018, doi: [10.1017/S0956796818000102](https://doi.org/10.1017/S0956796818000102).
- [19] Quassnoi, “Happy New Year: GPT in 500 lines of SQL,” Dec. 2023, [Online]. Available: <https://explainextended.com/2023/12/31/happy-new-year-15/>

- [20] T. Fischer, D. Hirn, and T. Grust, “SQL Engines Excel at the Execution of Imperative Programs,” *Proc. VLDB Endow.*, vol. 17, no. 13, pp. 4696–4708, Feb. 2025, doi: [10.14778/3704965.3704976](https://doi.org/10.14778/3704965.3704976).
- [21] A. Vaswani *et al.*, “Attention Is All You Need.” [Online]. Available: <https://arxiv.org/abs/1706.03762>