EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Database Systems Research Group

Bachelorthesis Computer Science

WITH ORDINALITY in DuckDB

Nico Faden

31.03.24

Examiner

Professor Torsten Grust

Supervisor

Denis Hirn

Nico Faden:
WITH ORDINALITY in DuckDB
Bachelorthesis Computer Science
Eberhard Karls Universität
From 01.09.23 to 31.03.24

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorthesis selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorthesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

30.03.24

Ort, Datum

Nico Faden

Abstract

Converting general-purpose data formats such as arrays or csv files into a format that can be used by Relational Database Management Systems is generally not a problem as most systems explicitly support this conversion. Since data in relational formats is generally orderless, information about the original order of elements within these general purpose formats may be lost during conversion. WITH ORDINALITY is a clause defined in the SQL Standard that can be used to save information about the original order by adding an additional column to the result table of certain functions. In this paper, we implement WITH ORDINALITY for the in-process SQL Database Management System DuckDB . We first find out how WITH ORDINALITY works in PostgreSQL. We then take a look at some examples and precisely define our goal by naming the functions we want to implement WITH ORDINALITY for. We summarize how DuckDB internally operates and get a general overview over the query execution engine. We use this knowledge to start by implementing a rudimentary version of WITH ORDINALITY which we then gradually improve. Finally, we compile our version of DuckDB and put it to the test by executing queries that use the feature and find that we have successfully achieved our goals.

Contents

1

Introduction

1.1 Exploiting The Power Of Relational Database Engines

Modern Database Management Systems (DBMS) have engines that use highly optimised procedures to efficiently perform operations on data, such as sorting or insertions. Because a DBMS can create execution plans tailored to specific queries, it may be worth converting data structures to a relational model to take advantage of these optimised procedures.

Converting general-purpose data formats, such as arrays, into a format that can be used in a relational model is generally not a problem, as modern DBMSs explicitly support this conversion. For a simple structure like an array, `PostgreSQL` uses the `UNNEST(...)` function in order to convert all elements inside the array into row items within a table. Most DBMS contain similar functions that serve the same purpose.

However, since a table that is part of a relational model does not typically preserve order, an issue arises. If the order is not explicitly represented in some form during conversion, some information will inevitably be lost. If we were to convert an array into a table and then convert that table back into an array, even without performing any operations, there is no guarantee that the array will be the same as it was before. The elements can be in random order 1.1.

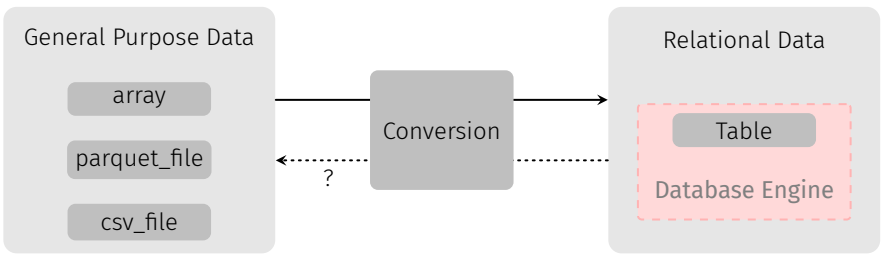

Figure 1.1: One-way conversion into a relational data format without a guaranteed way of preserving order.

This is where `WITH ORDINALITY` enters the picture.

1.2 What Is WITH ORDINALITY ?

1.2.1 Definition and usage

The keyword WITH ORDINALITY was introduced in **PostgreSQL** with version 9.4[1]. When this clause is given at the end of a `table function` call, a numerator column of type `BIGINT`[2] is appended to the result set. This column, referred to as `ordinality`, assigns a unique identifier to each row and provides an order. A `table function`, also referred to as a `table-valued function` or `TVF`, is a function that returns a set of rows and that can be used in the `FROM` clause of a query just the same as any relation. The syntax for WITH ORDINALITY in **PostgreSQL** is as follows:

```
function_call [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ])]]
```

This is a generalization of the SQL-standard syntax for the `UNNEST(...)` function, for which WITH ORDINALITY has originally been implemented for. Note that the `ordinality` column may also be renamed using an alias. This needs to be considered during our development of WITH ORDINALITY .

1.2.2 Example

The `UNNEST(...)` function consumes all items of an array and creates a table where each row contains one item. Consider the following query:

```
1 SELECT
2     *
3 FROM
4     UNNEST(ARRAY['a','b','c'])
5         WITH ORDINALITY
6 AS
7     _(letters, ordinality);
```

letters	ordinality
a	1
b	2
c	3

Table 1.1: The result for the query to the left.

The query produces table 1.1 that contains the unnested array-values. Due to the WITH ORDINALITY clause, an extra column has been added to indicate the order in which the array elements were consumed.

1.3 The Missing Link

WITH ORDINALITY can now support this conversion of real-world data into a relational context and back to the original structure even when the data itself does not provide a sensible order. Remember that a table in a relational model is a set of rows with no fixed order. Even if the DBMS decides to output the rows of the table in the previous query 1.1 in any other way, the original order is still preserved within the `ordinality` column, allowing for the array to be reconstructed

[1] https://www.postgresql.org/docs/current/functions-srf.html
[2] This type has a size of 8 Bytes.

correctly. We can also take advantage of this functionality in more complicated scenarios. For example, take a look at table 1.2. If we unnest these elements like in the query below, the associated `id` is preserved, but there is no additional information about the order of elements within the converted string.

UnnestExample

id	elements
1	first, second, third
2	fourth, fifth, sixth
3	seventh, eighth, ninth

Table 1.2: A table with comma-seperated values within a `string`, associated with an `id`. A structure like this might represent scanned data entries with varying dates of origin.

```
SELECT
    id,
    UNNEST(string_to_array(
        elements,
        ', '))
    AS element
FROM
    UnnestExample;
```

id	element
1	first
1	second
1	third
2	fourth
2	fifth
2	sixth
3	seventh
3	eighth
3	ninth

Table 1.3: The resulting table for the query to the left.

While the order is initially preserved to some extent within the result 1.3 in this specific case, there is no guarantee that the DBMS will output the rows in that exact order. Additionally, any operators that change the order of lines in the output will erase this information (see 1.4). An example of this would be an `ORDER BY` clause:

```
SELECT
    id,
    UNNEST(string_to_array(
        elements,
        ','))
    AS element
FROM
    UnnestExample
ORDER BY
    element;
```

id	element
3	eigth
2	fifth
3	ninth
1	second
2	sixth
1	third
1	first
2	fourth
3	seventh

Table 1.4: The resulting table after invoking `ORDER BY`.

To preserve this order is exactly where `WITH ORDINALITY` proves useful again:

```
SELECT
    id, substring, ordinality
FROM
    UnnestExample,
    UNNEST(string_to_array(
        elements,
        ','))
        WITH ORDINALITY
    AS _(substring,ordinality)
ORDER BY substring;
```

id	substring	ordinality
3	eigth	2
2	fifth	2
3	ninth	3
1	second	2
2	sixth	3
1	third	3
1	first	1
2	fourth	1
3	seventh	1

Table 1.5: The resulting table after invoking `ORDER BY` using `WITH ORDINALITY` .

Using the `ordinality`-column in 1.5 it is now possible to recreate the original `UnnestExample` 1.2 table, regardless of additional operators changing the order of rows. Note that the `ordinality`-index is reset for each consumed array. This will become relevant to our own implementation at a certain point. We have now seen that `WITH ORDINALITY` is a useful tool that can provide an interface for bilateral data conversion to and from a relational system, see Figure 1.2 for further illustration.

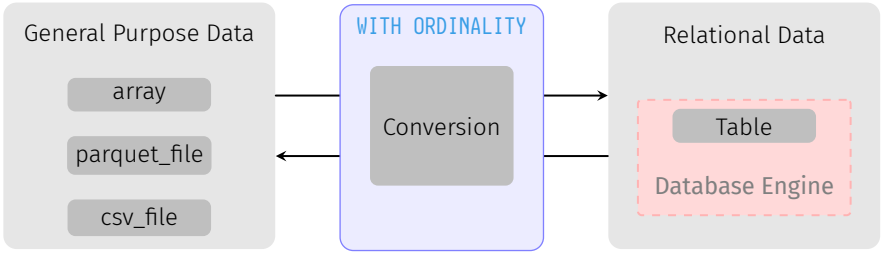

Figure 1.2: Conversion of general data into a relational data format with the ability to re-convert while preserving the original order.

1.4 The Goal Of This Thesis

DuckDB is a young in-process SQL OLAP (2.2) database management system with its first version released in 2019. The objective of this thesis is to implement `WITH ORDINALITY` in this DBMS for each of the following `table functions`.

1.4.1 `GENERATE_SERIES()` and `RANGE()`

`GENERATE_SERIES(start,stop,step)` is a function that takes up to three arguments of type `bigint`[3]. A table is produced and its rows are populated by starting with the value `start` and then iteratively adding the value of `step` for each new entry. The process stops when a value greater than `stop` has been reached, with this last value not being part of the result set. Either `step` or both `start` and `step` can be omitted, with the function then defaulting to `step=1` and `start=0`. `RANGE(v)` is a function that internally calls `GENERATE_SERIES(0,v-1,1)`. See the queries below for examples.

```
1 -- The four queries below are effectively
      identical
2 SELECT * FROM RANGE(10);
3 SELECT * FROM GENERATE_SERIES(9);
4 SELECT * FROM GENERATE_SERIES(0,9);
5 SELECT * FROM GENERATE_SERIES(0,9,1);
```

generate_series
0
1
2
3
4
5
6
7
8
9

Table 1.6: The resulting table for the query to the left

[3]This type, like in **PostgreSQL**, is of size 8 Byte.

As can be seen in table 1.6, all four queries produce the same result as expected. Internally `GENERATE_SERIES(0,v-1,1)` uses values `start=0`, `stop=9` and `step=1`. The query below passes the value `2` as an argument for `step` and `11` for `stop` instead. The result can be seen in table 1.7. Since the increment of `2` causes the next value after `10` to be `12`, the `stop` argument is not included in the resulting table in this case.

```
1 SELECT * FROM GENERATE_SERIES(0,11,2);
```

generate_series
0
2
4
6
8
10

Table 1.7: The resulting table for the query to the left

1.4.2 `REPEAT()` and `REPEAT_ROW()`

`REPEAT(a,b)` is a function that takes two arguments of type `any`[4] and {`bigint`} respectively. It then creates a table with `b` rows each containing the value `a`. `REPEAT_ROW(a,num_rows=b)` works just the same, except that instead of simply providing argument `b`, the keyword `num_rows` has to be prepended before passing the argument. The queries below show examples of how to use these functions.

```
1 -- The the two queries below are
      effectively identical
2 SELECT * FROM REPEAT('This is a string',
      5);
3 SELECT * FROM REPEAT_ROW('This is a
      string', num_rows= 5);
```

repeat
This is a string
This is a string
This is a string
This is a string
This is a string

Table 1.8: The resulting table for the query to the left

[4] This type can be any type defined in **DuckDB** .

```
SELECT * FROM REPEAT(['this','is','array'
    ,'content'], 2);
```

repeat
[this,is,array,content]
[this,is,array,content]

Table 1.9: The resulting table for the query to the left

1.4.3 `UNNEST()`

The `UNNEST()` function takes a `list`[5] or `struct`[6] and unnests the input by one level. There is also an additional `boolean` parameter `recursive` that makes the function fully unnest all `lists` and then fully unnest all `structs`. This can be useful to fully flatten columns. Another numeric parameter `max_depth` can be used to control the depth to which `lists` and `structs` will be unnested. By default, this parameter is set to the numeric limit of the DuckDB-type `idx_t`[7]. Note that `UNNEST()` does not unnest `lists` within `structs`. However, this function is only treated as a `table function` by DuckDB when it is called from within a `FROM` clause. Another detail is that when `UNNEST()` is treated as a `table function`, it does not accept any `structs` as arguments, only a single `list`. In this case it is also not possible to use the `recursive` option. We have already seen an example of a PostgreSQL `UNNEST()` call and the resulting table 1.1. Below is an example of the same query without the additional `WITH ORDINALITY` clause in DuckDB, as well as an example of a non-`table function` call. The results can be seen in table 1.10 and 1.11 respectively.

```
SELECT
    *
FROM
    UNNEST(['a','b','c'])
AS
    _(letters);
```

letters
a
b
c

Table 1.10: The resulting table of the previous query

```
-- If the same call was issued from
    within a FROM clause, an error would
    occur
SELECT
    UNNEST([
        {'a': 1, 'b': 2},
        {'a': 3, 'b': 4}
        ], recursive := true);
```

a	*b*
1	3
2	4

Table 1.11: The resulting table of the previous query where `UNNEST()` is not treated as a `table function`

[5] An ordered sequence of data values of the same type.
[6] A dictionary of multiple named values, where each key is a string, but the value can be a different type for each key.
[7] This type is the same as a 64 bit `unsigned integer`

1.4.4 GLOB()

The GLOB() function accepts a string argument that is used for **glob pattern matching**[8]. It returns a relation containing all filenames within the current directory that match the provided pattern, not to be confused with the GLOB operator that returns a boolean value depending on whether or not a pattern matching file can be found.

The DuckDB documentation reads *"Use the question mark (?) wildcard to match any single character, and use the asterisk (*) to match zero or more characters. In addition, use bracket syntax ([ ]) to match any single character contained within the brackets, or within the character range specified by the brackets. An exclamation mark (!) may be used inside the first bracket to search for a character that is not contained within the brackets."* [1]. Example usages of this function can be seen below.

```
-- Returns all files
SELECT * FROM GLOB('*');

-- Returns all text files containing any three characters in the filename
SELECT * FROM GLOB('???.txt')

-- Returns all text files that start with either the letter a, b or c
SELECT * FROM GLOB('[a-c]*.txt');

-- Returns all text files that don't start with the letter a, b or c
SELECT * FROM GLOB('[!a-c]*.txt');
```

1.4.5 READ_CSV() and READ_CSV_AUTO()

Reading Data from CSV files in PostgreSQL *vs.* in DuckDB. A key difference compared to PostgreSQL is the process of importing CSV-files. In PostgreSQL, CSV-files can only be imported using a COPY statement which copies the file's contents into an existing table. The table must have the same number of columns as the file, and each column's content must be of (or convertible to) the same type as the corresponding table column. This means that WITH ORDINALITY is not supported for CSV-files.

DuckDB also allows for this functionality under the same conditions. Additionally, it facilitates reading CSV-files through the table function READ_CSV(). This feature enables the direct usage of file contents without the need to import them into an existing table beforehand. Consider a simple CSV-file ***values.csv*** that contains a number of integers, see illustration 1.3. In order to calculate the sum of these integers in PostgreSQL, we first have to create an appropriate table and then execute our query reading the values from that table. If only the sum is required and the file contents are not needed, it is sensible to consider deleting the table after the execution in order to free memory. The full query might look like the query below.

```
CREATE TABLE temporaryTable (
    vals INTEGER
);
```

[8] https://en.wikipedia.org/wiki/Glob_(programming)

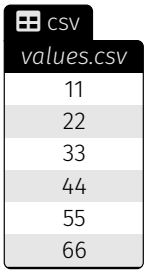

Figure 1.3: A number of integers within a single-column CSV-file

```
4    COPY temporaryTable(vals) FROM 'values.csv';
5    SELECT sum(vals) FROM temporaryTable;
6    DROP TABLE temporaryTable;
```

Using the `READ_CSV()` **table function** call, the same calculation can be executed in **DuckDB** within just one statement:

```
1    SELECT sum(vals) from READ_CSV('test.csv') AS _(vals);
```

In this case, both DBMS can quickly calculate the result of 231. However, depending on the types and the amount of data read into the system, the **PostgreSQL** approach might increase the query execution time. This is not only due to the process of creating and deleting a table alone but may also be impacted by other related mechanisms that spring into action as well, such as creating indexes. The **DuckDB** method of using a function to read the file does not cause such an overhead while at the same time being more handy as well as using a condensed syntax. Furthermore, because `READ_CSV()` is a **table function**, we will implement support for `WITH ORDINALITY`.

How It Works. `READ_CSV()` is a function that takes a filepath in the form of a `string` as an argument as well as some optional parameters like a column delimiter or specifying the `nullstring`. A full list of parameters can be found in the official **DuckDB** overview[9]. It reads the data contained within the file and converts it into a relational format, ready to be used like a regular table within a `FROM` clause. `READ_CSV_AUTO()` does the same, but uses a **CSV-sniffer** to infer all the relevant parameters automatically. The optional argument `auto_detect=true` effectively turns `READ_CSV()` into `READ_CSV_AUTO()` as well.

Example. Say we want to convert the table represented in *table.csv* (see illustration 1.4) into a relational format usable by our DBMS. For a simple **CSV file**, we have multiple options. Example queries and the resulting table 1.12 can be seen below.

[9] https://duckdb.org/docs/data/csv/overview.html

csv

table.csv

numbers, chars
11, a
22, b
33, c
44, d

Figure 1.4: A `CSV file` containing `integers` in one column and `varchars` in the other column, using a comma as delimiter and the first line to declare headers

```
1 -- Query 1
2 SELECT * FROM READ_CSV('table.csv', delim
      =',') AS table;
3 -- Query 2
4 SELECT * FROM READ_CSV_AUTO('table.csv')
      AS table;
```

table

numbers	*chars*
11	a
22	b
33	c
44	d

Table 1.12: The result of converting the `CSV file` into a relation

1.4.6 `PARQUET_SCAN`() and `READ_PARQUET`()

Parquet is a columnar storage format not directly supported by **PostgreSQL**. Just like the function `READ_CSV_AUTO`(), it takes a file path as its argument and also has additional parameters that are not very relevant to us and can be found in the offical **DuckDB** overview[10]. `PARQUET_SCAN`() and `READ_PARQUET`() are synonymous functions that execute the same code. As **DuckDB** is a young DBMS that is actively being worked on, such double occupancies of function names are often the result of improvements being made to the system while at the same time trying not to break established code.

[10] https://duckdb.org/docs/data/parquet/overview.html

2

The Innards Of The Duck

2.1 What The Duck Aims To Do

Data scientists as well as data analysts spend the majority of their time working with large amounts of data. DBMSs are well suited to efficiently deal with frequently required operations such as sorting, aggregating or joining data through SQL queries. As such, it seems to be a straightforward decision to make use of a DBMS in this field. However, according to surveys like the annual *Stackoverflow.co* developer survey[1], a lot of developers utilise tools such as the Pandas library to manipulate data instead. This is most likely due to most DBMSs being difficult to install, properly set up as well as maintain. In analytical scenarios where large amounts of data are being moved and worked on, data transfer to and from the client can also be slow.

For online transactional processing (`OLTP`), `SQLite` is a popular solution that mitigates some of these problems by being a cross-platform DBMS that needs no configuration, setup or administration. DuckDB as a relational in-process database aims to address these issues for analytical processing (see Section 2.2). It can be compiled for all major operating systems and CPU architectures, providing client APIs for a number of languages such as `C`, `C++`, `Java`, `Python`, `R` and others. In order to properly implement our `WITH ORDINALITY` function, we first need to understand how **DuckDB** operates.

2.2 OLAP Workload

While OLTP databases are used for procedures such as order entry or payment processing, requiring efficient insertion and deletion of rows, analytical DBMSs are used to analyse large amounts of data like historical data or aggregates from multiple sources. Examples include prediction of customer behaviour, identifying profitability and analyisis of market trends. The amount of data to be processed is generally much larger than OLTP workloads, often ranging from multiple terabytes to petabytes. In contrast to OLTP, efficient implementation of aggregation and value comparison is crucial, while write processes have a lower impact on overall performance in relation to the workload. To accomplish this, analytical DBMSs often follow a columnar data model, as does **DuckDB** .

[1] https://survey.stackoverflow.co/2023/#overview

2.3 Columnar Data Storage Format

In this data model, a table is physically stored and held in memory by placing column values in consecutive order, as opposed to chaining rows of data. See Figure 2.1.

Unicode

c_1	c_2	c_3
A	L	W
B	M	X
C	N	Y
D	O	Z

Table 2.1: A table that stores unicode characters

Figure 2.1: Items of table *Unicode* are stored in memory in a columnar manner

In this example there is a table (see table 2.1) that contains `Unicode` characters. The individual column values are stored consecutively in memory. This means that the next value within the same column will be placed, in this case, 4 bytes after the address of the first value. This increases spatial locality for `OLAP` workloads, reducing the number of hard disk seeks and significantly improving overall performance. Seek operations are a major bottleneck, so minimizing them is crucial. Note that this is just a general example of a columnar data format. Generally, **DuckDB** stores pointers for variable-length character strings even for single characters. These strings are located in a special heap (see 2.4). In `DuckDB`, the container format used to store in-memory data during execution is called a `Vector`.

2.4 Vectors

A `Vector` logically represents an array that contains data of a single type. In relational terms, it is a horizontal slice of a column. Multiple `Vectors` combined then make up an entire column.

The source code for the `Vector` can be found in *vector.hpp*. Every `Vector` contains the following protected fields:

```
class Vector {
    [...]
protected:
    VectorType vector_type;
    LogicalType type;
    data_ptr_t data;
    ValidityMask validity;
    buffer_ptr<VectorBuffer> buffer;
    buffer_ptr<VectorBuffer> auxiliary;
};
```

- `vector_type` specifies the physical `vector format` (more information on this will be provided shortly).
- `type` specifies the logical type that is stored within the vector, like `integer` or `bool`.
- `data` holds a pointer to the stored data.
- `validity` contains the validity mask of the vector, which is a bitset that signifies null-values. For non-null values it is set to 1, for null values it is set to 0.
- `buffer` points to the buffer that holds the data of the vector.
- `auxiliary` points to an optional buffer that may hold auxiliary data such as the aforementioned `string` heap.

DuckDB utilises various `vector formats` to represent the same logical data in different physical forms. This allows for compressed execution in some cases. The base `vector formats` are defined in *vector_type.hpp* as an `enum`:

```
enum class VectorType : uint8_t {
    FLAT_VECTOR,
    FSST_VECTOR,
    CONSTANT_VECTOR,
    DICTIONARY_VECTOR,
    SEQUENCE_VECTOR
};
```

- The `FLAT_VECTOR` type is the most general type of vector and is simply stored as an uncompressed contiguous array. There is no difference between the physical and logical representation.
- The `FSST_VECTOR` type holds `string` data compressed with `FSST`[2] .
- The `CONSTANT_VECTOR` type is physically stored as a single value, logically representing an array that contains only this value. Instead of writing this value for every row like in a `FLAT_VECTOR`, only one instance needs to be stored.
- The `DICTIONARY_VECTOR` type is physically made up of two related `Vectors`: a *child vector* as well as a *selection vector*. The former holds the actual values while the *selection vector* holds indexes pointing to the value within the *child vector*. This can greatly increase compression when dealing with repetitive values, *e.g.*, large arrays of `strings`.

The `SEQUENCE_VECTOR` will become relevant to our implementation later on. It is used for logically representing incremental sequences by physically storing a `base` as well as an `increment` value. When decompressing, this increment is then repeatedly added to the previous value to fill up the `Vector`, starting at `base`. It is the `vector format` we will use to populate our `ordinality` column. There is also a set of *complex types* like `list vectors` and `map vectors`, however, these formats are not very relevant to our implementation. Every format can be converted to a `unified vector format` as described by Kuiper, Raasveldt, and Mühleisen [2] in order to universally use a `Vector` regardless of the type, acting as a generic view over the vector contents.

[2]Stands for *Fast Random Access String Compression*

2.5 Engine

DuckDB utilises a *columnar-vectorized query execution engine* to make full use of the data format. The engine interprets queries and executes pre-compiled code on entire `vectors` of data. This approach minimizes interpretation overhead and virtual function calls by running specialized code for the types present in the query. Examples of this query interpretation techniques include *key normalization*, *statically pre-compiled mem*-functions* and switching between different sorting algorithms depending on the types contained within the query. Kuiper and Mühleisen [3] describes these methods in detail.

Instead of using a regular *volcano* model where rows of data are pulled from the root node of the `physical operator` tree, **DuckDB** uses a vectorized push-based model. This means that each operator itself defines how it returns its result, with the granular unit moved between operators being **DataChunk**s containing `Vectors`, instead of rows. This allows for the operators to make use of **SIMD**[3] operations, making `vector` processing highly CPU efficient.

2.6 Chunks

A `DataChunk` effectively represents a subset of a relation, *i.e.*, a horizontal slice of a table. They are the intermediate representation used by the execution engine and hold a set of `Vectors` of the same length. This length is typically set to 2048 entries in *vector.hpp* but can be configured to be any power of two. The *duckdb_vector_size* function may be used to obtain the current `Vector` size. The relevant source code for `DataChunk` can be found in *data_chunk.hpp*. As we will see later, **data** as well as **count** will be important to our implementation.

```
class DataChunk {
    public:
        //! The vectors owned by the DataChunk.
        vector<Vector> data;

        [...]

    private:
        //! The amount of tuples stored in the data chunk
        idx_t count;
        //! The amount of tuples that can be stored in the data chunk
        idx_t capacity;
        //! Vector caches, used to store data when ::Initialize is called
        vector<VectorCache> vector_caches;
    };
```

[3] Stands for *Single Instruction/Multiple Data*.

2.7 Overview

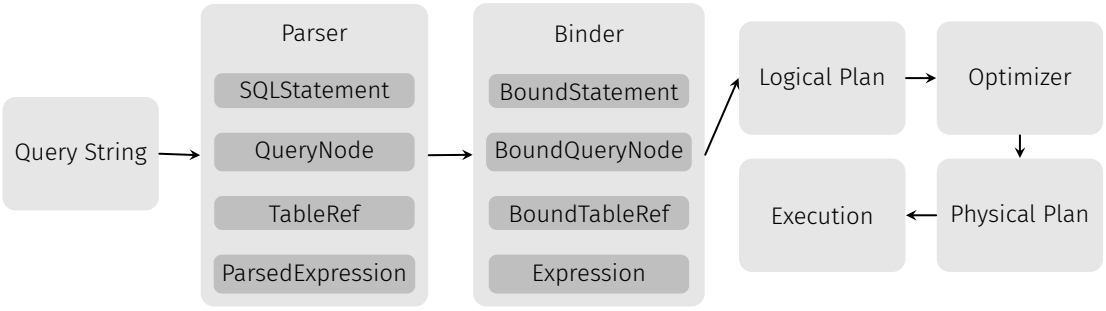

Figure 2.2: Step by step query execution in **DuckDB** .

Figure 2.2 shows the different stages of query processing. The `Query String` is read by the `Parser` and converted into four objects called `nodes`:

- `SQLStatement`: this represents a complete `SQL` statement. The type of the `SQLStatement` represents the kind of statement, *e.g.*, a `SELECT` statement. If the `Query String` contains multiple queries there may be multiple `SQLStatement`s.
- `QueryNode`: represents either a `SELECT` statement or a set operation.
- `TableRef`: represents any table source, *e.g.*, a base table or table-producing function. This will be important later.
- `ParsedExpression`: represents an expression within an `SQL` statement, *e.g.*, a column reference or a constant value.

The `Binder` then converts these nodes into their bound equivalents. This means that types and tables are resolved and aggregate or window functions are extracted.

The `Logical Planner` creates a logical query tree, which the `Optimizer` then tries to improve upon by running query optimizers. The most important optimizer for us will be `Projection Pushdown`, where filters are pushed down into the query plan. This operation may change the way that columns are referenced internally.

The `Physical Planner` then transforms the optimized logical query tree into a physical operator tree. This tree is executed in the `Execution` phase.

3

Implementation of WITH ORDINALITY in DuckDB

In the previous Section 2, we summarized the most important internal mechanisms of DuckDB in relation to our goal. We have seen what DuckDB is trying to achieve and what its workload looks like as a result of that. We have also seen how the DBMS internally represents data and how that data is moved through the operators within the engine. Now that we have a basic understanding of how DuckDB works, we can begin writing our code. The source code can be found in the official DuckDB repository[1]. The pull request for this thesis has the number #9014[2].

3.1 Adding the ordinality column

Our initial objective is to add a new column to the query output. In order to achieve this, we will skip ahead to the binding phase for now and experimentally add a new column whenever a table function is called. In *bind_table_function.cpp* we can find a member function called *BindTableFunctionInternal*, see Figure 3.1. By adding lines 7 – 8 after *bind* has been called on the table function, we add a new type as well as a new name to the return_types and return_names array respectively. Memory is allocated for the column, but as the operators are not instructed on how to handle it, it will be filled with random values pulled from the memory addresses. Since we do not always want to invoke WITH ORDINALITY by default, we need to find a way to properly parse the query first. By calling any table function, the output now has the ordinality column appended to it, see table 3.1.

[1] https://github.com/duckdb/duckdb
[2] https://github.com/duckdb/duckdb/pull/9014

```
1  Binder::BindTableFunctionInternal(TableFunction &table_function, [...]) {
2      [...]
3      if (table_function.bind || table_function.bind_replace) {
4          [...]
5          bind_data = table_function.bind(context, bind_input, return_types,
               return_names);
6          //hacky temporary solution
7          return_types.emplace_back(duckdb::LogicalType::INTEGER);
8          return_names.emplace_back("Ordinality");
9      }
10     [...]
11 };
```

Figure 3.1: *bind_table_function.cpp*

```
SELECT
    *
FROM
    UNNEST([1,2,3]);
```

unnest	ordinality
1	-16650000000
2	-16650000000
3	-16650000000

Table 3.1: The resulting table for the query to the left.

3.2 Activating `WITH ORDINALITY` in the Parser

DuckDB uses a repackaged standalone version of the PostgreSQL parser. This makes the process easier since `WITH ORDINALITY` is already built into the parser, it just needs to be activated. In *transform_table_function.cpp* we find the relevant code in lines 2–4, see Figure 3.2. The result that this method returns is a `TableFunctionRef`, which is a subclass that inherits from `TableRef`. We will add a new field `with_ordinality` to this `TableRef` class in *table_function_ref.cpp*, see line 15 – 16 in Figure 3.3. We initialize the value as `false` so that it is only overwritten as `true` when `WITH ORDINALITY` is detected in the parser. We replace the Exception in *transform_table_function.cpp*, see line 10 in Figure 3.4.

```
1  unique_ptr<TableRef> Transformer::TransformRangeFunction(duckdb_libpgquery::
       PGRangeFunction &root) {
2      if (root.ordinality) {
3          throw NotImplementedException("WITH ORDINALITY not implemented");
4      }
5      if (root.is_rowsfrom) {
6          throw NotImplementedException("ROWS FROM() not implemented");
7      }
8      if (root.functions->length != 1) {
9          throw NotImplementedException("Need exactly one function");
10     }
11     [...]
12     return std::move(result);
13 }
```

Figure 3.2: *transform_table_function.cpp*

From the `TableFunctionRef` we can now know in the Binder whether or not `WITH ORDINALITY` has been invoked. If we look at the code again, we can see that the call to `BindTableFunctionInternal` only takes a reference to a `TableFunction` rather than a `TableFunctionRef`. This means that we also need to add our `with_ordinality` `boolean value` to the `TableFunction` class in *table_function.hpp*, see line 9 in Figure 3.5. We can then propagate the information by copying the boolean value from the `TableFunctionRef` to the `TableFunction` in *bind_table_function.cpp*, see line 4 in Figure 3.6.

```
class TableFunctionRef : public TableRef {
        public:
                static constexpr const TableReferenceType TYPE = TableReferenceType
                    ::TABLE_FUNCTION;
        public:
                DUCKDB_API TableFunctionRef();
                unique_ptr<ParsedExpression> function;
                vector<string> column_name_alias;

                // if the function takes a subquery as argument its in here
                unique_ptr<SelectStatement> subquery;

                // External dependencies of this table function
                unique_ptr<ExternalDependency> external_dependency;

                // for WITH ORDINALITY
                bool with_ordinality = false;
                [...]
    };
```

Figure 3.3: *table_function_ref.cpp*

```
unique_ptr<TableRef> Transformer::TransformRangeFunction(duckdb_libpgquery::
    PGRangeFunction &root) {
        if (root.is_rowsfrom) {
            throw NotImplementedException("ROWS FROM() not implemented");
        }
        if (root.functions->length != 1) {
            throw NotImplementedException("Need exactly one function");
        }
        [...]
        auto result = make_uniq<TableFunctionRef>();
        result->with_ordinality = root.with_ordinality;
        [...]
}
```

Figure 3.4: *transform_table_function.cpp*

```
class TableFunction : public SimpleNamedParameterFunction {
    public:
        [...]
        bool projection_pushdown;
        [...]
        //! Additional function info, passed to the bind
        shared_ptr<TableFunctionInfo> function_info;
        //! whether WITH ORDINALITY has been invoked
        bool with_ordinality = false;
};
```

Figure 3.5: *table_function.hpp*

```
1  unique_ptr<BoundTableRef> Binder::Bind(TableFunctionRef &ref) {
2      [...]
3      auto table_function = function.functions.GetFunctionByOffset(best_function_idx)
           ;
4      table_function.with_ordinality = ref.with_ordinality;
5      [...]
6      auto get = BindTableFunctionInternal(table_function, [...]);
7      [...]
8      return make_uniq_base<BoundTableRef, BoundTableFunction>(std::move(get));
9  }
10
11 Binder::BindTableFunctionInternal(TableFunction &table_function, [...]) {
12     [...]
13     if (table_function.bind || table_function.bind_replace) {
14         [...]
15         bind_data = table_function.bind(context, bind_input, return_types,
               return_names);
16         if (table_function.with_ordinality) {
17             return_types.emplace_back(duckdb::LogicalType::INTEGER);
18             return_names.emplace_back("Ordinality");
19         }
20     [...]
21     }
22     [...]
23     return std::move(get);
24 };
```

Figure 3.6: *bind_table_function.cpp*

Finally, we can alter our previous code regarding `BindTableFunctionInternal` in *bind_table_function.cpp* to only add an additional column with the name *Ordinality* when `WITH ORDINALITY` has been parsed, see line 16 – 19 in Figure 3.6. The previous call to `UNNEST()` will regularly execute again instead of producing Table 3.1, unless `WITH ORDINALITY` is invoked.

3.3 Writing the `PrepareOrdinality` method

It is now time to populate the column with the correct values. DuckDB internally differentiates between two types of `table functions`: `TableFunction` and `TableInOutFunction`. The latter not only produces but may also consume a table-like structure such as an `array`. At the point of writing this thesis, the only function that falls within the second category is `UNNEST()`. We first define `PrepareOrdinality` for regular `table functions` of type `TableFunction` in *physical_table_scan.cpp*, see Figure 3.7 line 1 – 7. The method takes references to both a `DataChunk` as well as an `idx_t`, which is an index type defined by DuckDB as a 64-bit `unsigned integer`. If the size of the chunk is greater than `0`, we determine the index of the `ordinality` column in the `data` array of the `DataChunk`. For now, we do this by assuming it is always the last column, as it is the last column added during the `table function` binding phase.

We then make use of the `Sequence()` method defined in *vector.cpp*, which can be seen in Figure 3.8. `Sequence()` changes a `Vector` of any type into a `SEQUENCE_VECTOR`. We choose `ord_index` as starting

point, adding increments of 1 until the end of the original `Vector` size is reached. Note that the function `chunk.size()` yields the size of the `Vectors`, as all `Vectors` within a `DataChunk` are of the same size. The `ord_index` is used to remember what the last `ordinality` value in the previous `Vector` was. This is crucial in order to correctly number the entire column over all `Vectors`.

```
1  void PhysicalTableScan::PrepareOrdinality(DataChunk &chunk, idx_t &ord_index) const
       {
2      idx_t ordinality = chunk.size();
3      if (ordinality > 0) {
4          idx_t ordinality_column = column_ids.size() - 1;
5          chunk.data[ordinality_column].Sequence(ord_index,1, ordinality);
6      }
7  }
8
9  class TableScanLocalSourceState : public LocalSourceState {
10     public:
11         [...]
12         unique_ptr<LocalTableFunctionState> local_state;
13         idx_t ord_index = 1;
14     };
15
16 SourceResultType PhysicalTableScan::GetData(ExecutionContext &context, DataChunk &
       chunk,
17                                             OperatorSourceInput &input) const {
18     [...]
19     function.function(context.client, data, chunk);
20
21     if (function.with_ordinality) {
22         PrepareOrdinality(chunk, state.ord_index);
23     }
24
25     if (chunk.size() == 0) {
26         return SourceResultType::FINISHED;
27     } else {
28         state.ord_index += chunk.size();
29         return SourceResultType::HAVE_MORE_OUTPUT;
30     }
31 }
```

Figure 3.7: *physical_table_scan.cpp*

```
1  void Vector::Sequence(int64_t start, int64_t increment, idx_t count) {
2      this->vector_type = VectorType::SEQUENCE_VECTOR;
3      this->buffer = make_buffer<VectorBuffer>(sizeof(int64_t) * 3);
4      auto data = reinterpret_cast<int64_t *>(buffer->GetData());
5      data[0] = start;
6      data[1] = increment;
7      data[2] = int64_t(count);
8      validity.Reset();
9      auxiliary.reset();
10 }
```

Figure 3.8: *vector.cpp*

The location where `PrepareOrdinality` is called is important as we need to keep track of the `ord_index` across multiple `DataChunks`. This necessitates holding the value in memory until the operator has completely finished execution. DuckDB uses a morsel driven parallelism described by Leis et al. [4] and the construct that represents a morsel in DuckDB is called `LocalSourceState`. We add the `ord_index` to the `TableScanLocalSourceState`, which is a subclass of `LocalSourceState` used in the execution of `table functions`, see line 13 in Figure 3.7. We can now call this method from within the `GetData` member function right after the specific `table function` call happens, see line 21 – 23. We also make sure to update `ord_index` before returning `SourceResultType`, which is used to tell the engine whether a data source has been exhausted, see line 28.

```
1  void PhysicalTableInOutFunction::PrepareOrdinality(DataChunk &chunk, idx_t &
       ord_index, bool &ord_reset) const {
2      idx_t ordinality = chunk.size();
3      if (ordinality > 0) {
4          if (ord_reset) {
5              ord_index = 1;
6              ord_reset = false;
7          }
8          idx_t ordinality_column = column_ids.size() - 1;
9          chunk.data[ordinality_column].Sequence(ord_index,1, ordinality);
10     }
11 }
```

Figure 3.9: *physical_tableinout_function.cpp*

We implement the method for `TableInOutFunctions` in *physical_tableinout_function.cpp* as well, see Figure 3.10 line 1 – 11. The function works the same as for regular `table functions`, except that we additionally check a boolean named `ord_reset`. We use this value to reset the `ordinality` index to 1. `TableInOutFunctions` may consume multiple inputs from different sources, *e.g.*, a call to `UNNEST()` containing multiple arrays. In order to mimic the behaviour of PostgreSQL and allow for the functionality described in the introduction part of this thesis, we reset the index whenever the `TableInOutFunction` draws its data from a new source. The `TableInOutLocalState` contains a `boolean` called `new_row` that is set to `true` whenever this is the case. We add both `ord_index` as well as `ord_reset` to this `TableInOutLocalState` class, see line 10 – 11 in Figure 3.10. We also have to call our *PrepareOrdinality* method at the right time within the *PhysicalTableInOutFunction::Execute* member function, see lines 18 – 20 as well as 31 – 33. This way, both the case where a `projection` as well as the case where no `projection` happens is covered. If `new_row` is set, we have to make sure to reset our index, see line 27. We otherwise proceed just like we did in the implementation for regular `table functions` and update the `ord_index` after the chunk has been processed, see line 38. `WITH ORDINALITY` properly works for all functions with the exception of `READ_CSV()`.

```
class TableInOutLocalState : public OperatorState {
        public:
                TableInOutLocalState() : row_index(0), new_row(true) {
                }

                unique_ptr<LocalTableFunctionState> local_state;
                idx_t row_index;
                bool new_row;
                DataChunk input_chunk;
                idx_t ord_index = 1;
                bool ord_reset = false;
};
[...]
OperatorResultType PhysicalTableInOutFunction::Execute(DataChunk &chunk, [...])
    const {
        [...]
        if (projected_input.empty()) {
                duckdb::OperatorResultType result = function.in_out_function(
                    context, data, input, chunk);
                if (function.with_ordinality) {
                        PrepareOrdinality(chunk, state.ord_index, state.ord_reset);
                }
                return result;
        }
        if (state.new_row) {
                [...]
                state.row_index++;
                state.new_row = false;
                state.ord_reset = true;
        }

        auto result = function.in_out_function(context, data, state.input_chunk,
            chunk);
        if (function.with_ordinality) {
                PrepareOrdinality(chunk, state.ord_index, state.ord_reset);
        }
        if (result == OperatorResultType::FINISHED) {
                return result;
        }
        [...]
        state.ord_index += chunk.size();
        return OperatorResultType::HAVE_MORE_OUTPUT;
}
```

Figure 3.10: *physical_tableinout_function.cpp*

3.4 Fixing `READ_CSV()`

If we try to execute the `READ_CSV()` function, we are met with an error message that reads `"MultiFileReader::CreatePositionalMapping - global_id is out of range in global_types for this file"`. This message leads to a file called *multi_file_reader.cpp*. Looking at the code in Figure 3.11, we can see that the `InternalException` is thrown within the `CreateNameMapping()` member function of `MultiFileReader`. This function is responsible for mapping the column names found in the

csv-file to the columns requested by the query so that the Vectors returned by the query are filled with the correct csv data. global_column_ids contains identifiers of the columns requested in the SELECT clause. These identifiers are indexes that point to the corresponding Vectors in the data array of the DataChunk. global_types contains the type of each column contained in the file as determined by a csv sniffer that looks at the file at an earlier point during execution. This csv sniffer as described by Döhmen, Mühleisen, and Boncz [5], apart from column types, also automatically detects dialects and headers as well as date and time formats. When creating the mapping, the function checks every requested index and makes sure that this index is not higher than the amount of columns found within the file. Otherwise, it throws the InternalException found in line 10. In our case, this happens because an index to our newly added ordinality column is included in global_column_ids when it is requested in the SELECT clause. As the function only expects the original number of columns without the column added by the WITH ORDINALITY clause, the index cannot be found in the file. This also means that the query seen in Figure 3.12 on *table.csv* (1.12) will not cause an error, unlike the query seen in Figure 3.13, as the latter query does not cause the ordinality column index to be passed to the function.

```
1  void MultiFileReader::CreateNameMapping(
2      const vector<column_t> &global_column_ids,
3      const vector<string> &global_names,
4      const vector<LogicalType> &global_types, [...]) {
5      for (idx_t i = 0; i < global_column_ids.size(); i++) {
6          [...]
7          auto global_id = global_column_ids[i];
8          if (global_id >= global_types.size()) {
9              throw InternalException(
10                 "MultiFileReader::CreatePositionalMapping - global_id is out of
                       range in global_types for this file");
11         }
12         [...]
13     }
14 }
```

Figure 3.11: *multi_file_reader.cpp*

```
1 SELECT * FROM READ_CSV('table.csv') WITH ORDINALITY;
```

Figure 3.12: A query that causes an error

```
1 SELECT numbers, chars FROM READ_CSV('table.csv') WITH ORDINALITY;
```

Figure 3.13: A query that does not cause an error

To fix this, we have to tell the function to skip the ordinality column entirely. For now, we can naively achieve this by subtracting 1 from global_column_ids.size() when WITH ORDINALITY is active, as we assume that the ordinality column is the last column. However, this will be problematic

when the column order is changed in the `SELECT` clause. We implement are more sophisticated solution in Section 3.7. As we only want to subtract when `WITH ORDINALITY` is active, we need to propagate our `with_ordinality` **boolean** into the function call. Using a Debugger we trace the function calls and find out that the last level within the **call stack** where we have direct access to information from the **Binder** is `CSVFileScan(...)` found in the file *csv_file_scanner.cpp*. The call stack can be seen in Figure 3.14.

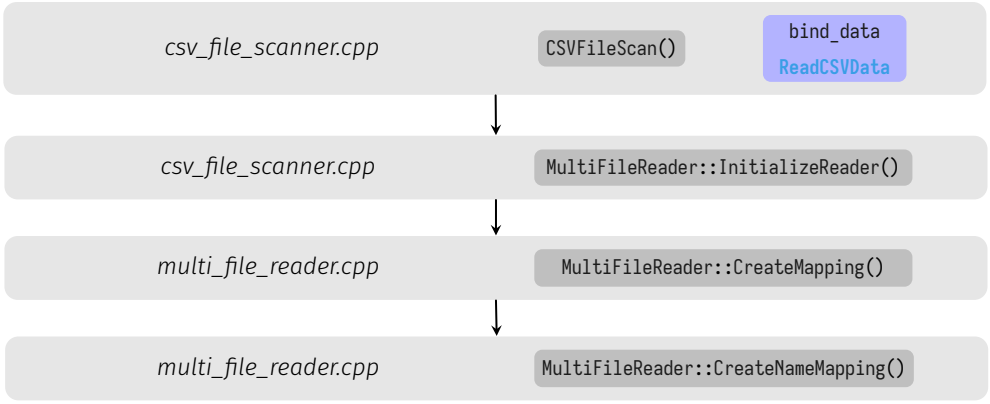

Figure 3.14: The callstack of the function initialization until the name mapping where we encounter an error

We add the **boolean** `with_ordinality` to the `bind_data` found in the `CSVFileScan()` function, which is of type `ReadCSVData`. More specifically, `ReadCSVData` is a subclass of `TableFunctionData`, but we only add the boolean field to `ReadCSVData` in *read_csv.hpp* (Figure 3.15, line 13) as the other **table functions** do not need to carry this additional information in their **function data**. We propagate this value from the `TableFunction` in the **Binder**, see Figure 3.16. There is already a **conditional** block that checks whether the currently handled function is `READ_CSV()` or `READ_CSV_AUTO()`. The `bind_data` is **typecast** as `ReadCSVData` (line 10) to access the `with_ordinality` field, which then copies the value of the same field in `TableFunction` (line 11).

```
1     struct ReadCSVData : public BaseCSVData {
2         vector<LogicalType> csv_types;
3         vector<string> csv_names;
4         vector<LogicalType> return_types;
5         vector<string> return_names;
6         shared_ptr<CSVBufferManager> buffer_manager;
7         unique_ptr<BufferedCSVReader> initial_reader;
8         vector<unique_ptr<BufferedCSVReader>> union_readers;
9         bool single_threaded = false;
10        MultiFileReaderBindData reader_bind;
11        vector<ColumnInfo> column_info;
12        CSVStateMachineCache state_machine_cache;
13        bool with_ordinality = false;
14        }
15    };
```

Figure 3.15: *read_csv.hpp*

```
1  Binder::BindTableFunctionInternal(TableFunction &table_function, [...]) {
2      if (table_function.bind || table_function.bind_replace) {
3          bind_data = table_function.bind(context, bind_input, return_types,
               return_names);
4          if (table_function.with_ordinality) {
5              return_types.emplace_back(duckdb::LogicalType::INTEGER);
6              return_names.emplace_back("Ordinality");
7          }
8          [...]
9          if (table_function.name == "read_csv" || table_function.name == "
               read_csv_auto") {
10             auto &csv_bind = bind_data->Cast<ReadCSVData>();
11             csv_bind.with_ordinality = table_function.with_ordinality;
12             if (csv_bind.single_threaded) {
13                 table_function.extra_info = "(Single-Threaded)";
14             } else {
15                 table_function.extra_info = "(Multi-Threaded)";
16             }
17         }
18         [...]
19     }
20 }
```

Figure 3.16: *bind_table_function.cpp*

To further propagate this **boolean** value to the `CreateNameMapping()` function we add new parameters to the calling functions as seen in Figure 3.17 and Figure 3.19. We can declare the parameters as `const` since we don't need to change the value during the call. We read this value from the `bind_data` in the initializing function call as seen in Figure 3.18. For `CreateNameMapping()` we can now declare a new `idx_t num_column_ids` to use as boundary for iterating over all column identifiers. If `with_ordinality` is `true`, we simply subtract 1 from the amount of columns to be checked. This way our **ordinality** column will be skipped in the process, see lines 8 – 12 in Figure 3.19.

```
1  struct MultiFileReader {
2      [...]
3      //! Create all required mappings from the global types/names to the file-local
           types/names
4      DUCKDB_API static void CreateMapping([...], const bool with_ordinality);
5      static void InitializeReader([...], bool with_ordinality) {
6              FinalizeBind([...]);
7              CreateMapping([...], with_ordinality);
8              [...]
9      }
10     [...]
11     private:
12     static void CreateNameMapping([...], const bool with_ordinality);
13 };
```

Figure 3.17: *multi_file_reader.hpp*

```
CSVFileScan::CSVFileScan([...])
    : [...] {
    if (bind_data.initial_reader.get()) {
        [...]
        MultiFileReader::InitializeReader([...], bind_data.with_ordinality);
        return;
    } else if (!bind_data.column_info.empty()) {
        [...]
        MultiFileReader::InitializeReader([...], bind_data.with_ordinality);
        InitializeFileNamesTypes();
        return;
    }
    [...]
    MultiFileReader::InitializeReader([...], bind_data.with_ordinality);
    InitializeFileNamesTypes();
}
```

Figure 3.18: *csv_file_scanner.cpp*

```
void MultiFileReader::CreateMapping([...], const bool with_ordinality) {
    CreateNameMapping([...], with_ordinality);
    [...]
}
[...]
void MultiFileReader::CreateNameMapping([...], const bool with_ordinality) {
    [...]
    idx_t num_column_ids = global_column_ids.size();
    if (with_ordinality) {
            num_column_ids -= 1;
        }
    for (idx_t i = 0; i < num_column_ids; i++) {
        auto global_id = global_column_ids[i];
        if (global_id >= global_types.size()) {
            throw InternalException(
                "MultiFileReader::CreatePositionalMapping - global_id is out of
                    range in global_types for this file");
        }
        [...]
    }
    [...]
}
```

Figure 3.19: *multi_file_reader.cpp*

3.5 Enabling and Disabling `WITH ORDINALITY` for Specific `Table Functions`

As we have seen in the previous subsection, `WITH ORDINALITY` does not automatically work correctly for all `table functions` as some functions may need to be altered. If a new `table function` is added to **DuckDB** in the future, the author of that function should be able to choose whether or not they want to enable our feature. To achieve this, we first add another `boolean ordinality_implemented` to all `TableFunctions`, see Figure 3.20. Instead of setting the default value to

`false` in the **header file**, we follow the **DuckDB** convention and instead set the default within the **constructor** in *table_function.cpp* for both `with_ordinality` and `supports_ordinality`, see Figure 3.21. Note that there is an additional overloaded constructor that needs to be changed as well.

```
1  class TableFunction : public SimpleNamedParameterFunction {
2      public:
3          [...]
4          //! Whether or not the table function supports projection pushdown. If not
               supported a projection will be added
5          //! that filters out unused columns.
6          bool projection_pushdown;
7          [...]
8          //! whether WITH ORDINALITY has been invoked
9          bool with_ordinality;
10         //! whether WITH ORDINALITY has been implemented for this function
11         bool ordinality_implemented;
12 };
```

Figure 3.20: *table_function.hpp*

```
1  TableFunction::TableFunction([...])
2      : SimpleNamedParameterFunction([...], with_ordinality(false),
           supports_ordinality(false) {
3  }
4  [...]
5  TableFunction::TableFunction()
6      : SimpleNamedParameterFunction([...], with_ordinality(false),
           supports_ordinality(false) {
7  }
```

Figure 3.21: *table_function.cpp*

Every **table function** in **DuckDB** uses a `RegisterFunction` to initialize its values. We use this function to overwrite `with_ordinality` as well as `supports_ordinality` for every function mentioned in Section 1.4, except for `READ_PARQUET()` and `PARQUET_SCAN()`. An example of this can be seen in Figure 3.22, line 6. The **Binder** now additionally checks if `WITH ORDINALITY` is supported for the requested table function and throws an exception if it is not, see Figure 3.23 line 5–7. By checking this before `BindTableFunctionInteral` is called by `Bind` we make sure not to create any unnecessary overhead if the function is not supported while at the same time not having to adjust our already established code (see Figure 3.6). We use the built-in `toString()` method to get the name of the **table function** through the `TableFunctionRef`.

```
1  void RangeTableFunction::RegisterFunction(BuiltinFunctions &set) {
2      [...]
3      TableFunction range_function({LogicalType::BIGINT}, RangeFunction,
           RangeFunctionBind<false>, RangeFunctionInit);
4      range_function.cardinality = RangeCardinality;
5      // WITH ORDINALITY implemented for both range and generate_series
6      range_function.with_ordinality = false;
7      range_function.supports_ordinality = true;
8      [...]
9  }
```

Figure 3.22: *range_function.cpp*

```
1  unique_ptr<BoundTableRef> Binder::Bind(TableFunctionRef &ref) {
2      [...]
3      auto table_function = function.functions.GetFunctionByOffset(best_function_idx)
           ;
4      table_function.with_ordinality = ref.with_ordinality;
5      if (table_function.with_ordinality && !table_function.supports_ordinality) {
6          throw NotImplementedException("WITH ORDINALITY not implemented for " + ref.
               ToString());
7      }
8      [...]
```

Figure 3.23: *bind_table_function.cpp*

3.6 READ_PARQUET() and PARQUET_SCAN()

As can be seen in Figure 3.24, the initialization of the READ_PARQUET() function also uses the CreateNameMapping() function of the MultiFileReader class. Since we have altered the arguments passed to the MultiFileReader class functions according to Figure 3.17 and Figure 3.19 in the previous section without overloading the functions, the call to MultiFileReader::InitializeReader() from within InitializeParquetReader() now expects an additional boolean value. If we look at the code in Figure 3.25, we can see that bind_data is now of type ParquetReadBindData which directly inherits from TableFunctionData.

One potential solution for this problem is to add the boolean field to ParquetReadBindData similar to how a field was added to ReadCSVData like in Figure 3.15. However, it is important to note that READ_PARQUET() is an extension that is not directly located within the source folder of DuckDB . Instead, *parquet_extension.cpp* can be found in a seperate *extension* folder. This means that the code library cannot be included in *bind_table_function.cpp* and, as a result, the bind_data cannot be typecast to ParquetReadBindData like it is done for ReadCSVData in line 10 of Figure 3.16, making it impossible to access the with_ordinality field.

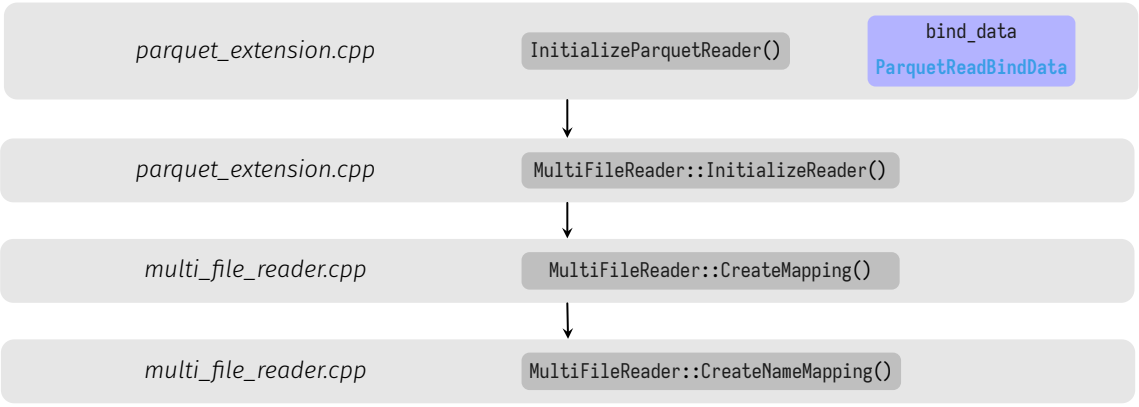

Figure 3.24: The callstack of the function initialization until the name mapping

```
1  struct ParquetReadBindData : public TableFunctionData {
2      shared_ptr<ParquetReader> initial_reader;
3      vector<string> files;
4      atomic<idx_t> chunk_count;
5      atomic<idx_t> cur_file;
6      vector<string> names;
7      vector<LogicalType> types;
8      vector<shared_ptr<ParquetReader>> union_readers;
9      idx_t initial_file_cardinality;
10     idx_t initial_file_row_groups;
11     ParquetOptions parquet_options;
12     MultiFileReaderBindData reader_bind;
13     [...]
14 }
15
16 static void InitializeParquetReader([...], const ParquetReadBindData &bind_data) {
17     auto &parquet_options = bind_data.parquet_options;
18     auto &reader_data = reader.reader_data;
19     if (bind_data.parquet_options.schema.empty()) {
20             MultiFileReader::InitializeReader(reader, parquet_options.file_options,
                    bind_data.reader_bind, bind_data.types,
21                                               bind_data.names, global_column_ids,
                                                     table_filters, bind_data.files[0],
22                                               context);
23         return;
24     }
25     [...]
26 }
```

Figure 3.25: *parquet_extension.cpp*

Another potential solution is to to add the field to the parent class `TableFunctionData`. In the next section, we will need an additional field of type `idx_t` in `bind_data` which is used to properly denote the `column index` of the `ordinality` column instead of assuming that it will always be the last `Vector` within a given `DataChunk`. In that section, however, we will find out that adding `with_ordinality` to `TableFunctionData` is not a feasible solution because of the `UNNEST()` function.

Instead, we add the `with_ordinality` **bolean** as well as the `original_ordinality_id` **index** to the parent class of `TableFunctionData`, which is `FunctionData`. See Figure 3.26, line 9 – 10. The reason for this will be explained shortly. We initialize both fields with the appropriate default value by defining a new `FunctionData` constructor, see Figure 3.27, line 6 – 8. We adjust the `InitializeParquetReader` function seen in Figure 3.25 in the next section.

```
1  struct FunctionData {
2      DUCKDB_API virtual ~FunctionData();
3
4      DUCKDB_API FunctionData();
5      FunctionData(const FunctionData &other);
6      DUCKDB_API virtual unique_ptr<FunctionData> Copy() const = 0;
7      DUCKDB_API virtual bool Equals(const FunctionData &other) const = 0;
8      DUCKDB_API static bool Equals(const FunctionData *left, const FunctionData *
           right);
9      bool with_ordinality = false;
10     idx_t original_ordinality_id = 0;
11
12     [...]
13 }
```

Figure 3.26: *function.hpp*

```
1  namespace duckdb {
2
3      FunctionData::~FunctionData() {
4      }
5
6      FunctionData::FunctionData() :
7          with_ordinality(false),
8          original_ordinality_id(0) {}
9
10     [...]
11 }
```

Figure 3.27: *function.cpp*

3.7 Code Refactoring

In this section, we will improve the way we emit the **ordinality** column during the **name mapping** phase when reading input from external files and deduplicate the `WITH ORDINALITY` logic. At the end of the previous section we decided to add the fields `with_ordinality` and `original_ordinality_id` to `FunctionData` instead of `TableFunctionData`. This is because the **bind data** for the `UNNEST()` function does not inherit from `TableFunctionData`, but instead inherits from `FunctionData` directly as can be seen in Figure 3.28. The full inheritance for the **bind data** of all **table functions** can be seen in Figure 3.29. The only way to access the fields in all **bind data** classes relevant to **table functions** is to add them to `FunctionData`. This avoids expensive dynamic castings and branches to decide

what kind of **bind data** is being dealt with and additionally simplifies the code. The downside is that every function, not only **table functions** now carries this additional information. We decide that this is not only the less expensive strategy in most cases, but can also something that can be refactored more easily in the future should the **DuckDB** maintainers decide to change the inheritance of `UnnestBindData`.

```
1  struct UnnestBindData : public FunctionData {
2      explicit UnnestBindData(LogicalType input_type_p) : input_type(std::move(
           input_type_p)) {
3      }
4
5      LogicalType input_type;
6
7  public:
8     [...]
9  };
```

Figure 3.28: *unnest.cpp*

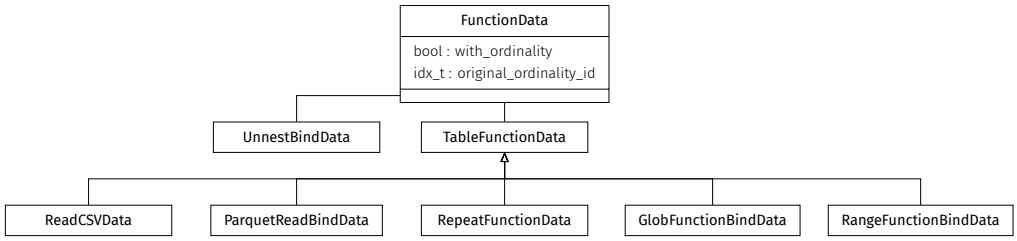

Figure 3.29: Inheritance of **table function bind data**

We adjust our code in the **Binder** from Figure 3.16 to be more generic as we no longer need to differentiate between `READ_CSV()` and other **table functions**, see Figure 3.30 line 5 – 12. At the request of the **DuckDB** maintainers, we change the `LogicalType` of the **ordinality** `Vector` to `BIGINT`. We also assert that the size of `return_types` and `return_names` is identical. For the `original_ordinality_id`, we use the size of `return_types` as at this point during execution we can be sure that the **ordinality** column is the last `Vector` in the `DataChunk`. We remove the now no longer needed `with_ordinality` field from `ReadCSVData` that we added in Figure 3.15.

```
Binder::BindTableFunctionInternal(TableFunction &table_function, [...]) {
    if (table_function.bind || table_function.bind_replace) {
        [...]
        bind_data = table_function.bind(context, bind_input, return_types,
            return_names);
        idx_t id = return_types.size();
        if (table_function.with_ordinality) {
            D_ASSERT(id == return_names.size());
            return_types.emplace_back(LogicalType::BIGINT);
            return_names.emplace_back("ordinality");
            bind_data->original_ordinality_id = id;
            bind_data->with_ordinality = true;
        }
        [...]
    return std::move(get);
}
```

Figure 3.30: *bind_table_function.cpp*

As of now, we have defined the `PhysicalTableInOutFunction::PrepareOrdinality()` method as well as the `PhysicalTableScan::PrepareOrdinality()` function in *physical_tableinout_function.cpp* and *physical_table_scan.cpp* respectively, see Figure 3.7 and Figure 3.9. These methods essentially execute the same logic. We will remove these functions and create a new file *ordinality_data.hpp* at *src/include/duckdb/function/table/*. Within this file we define the struct `OrdinalityData` which will now hold the logic for `WITH ORDINALITY` , see Figure 3.31. We also remove the `ord_index` as well as `ord_reset` fields from the `TableInOutLocalState` as well as `TableScanLocalSourceState`.

```
1  #pragma once
2  #include "duckdb/common/types/data_chunk.hpp"
3  namespace duckdb {
4  struct OrdinalityData {
5      bool with_ordinality = false;
6      idx_t column_id;
7      idx_t idx = 1;
8      bool reset = false;
9
10     void SetOrdinality(DataChunk &chunk, const vector<column_t> &column_ids) {
11         const idx_t ordinality = chunk.size();
12         if (ordinality > 0) {
13             if (reset) {
14                 idx = 1;
15                 reset = false;
16             }
17             D_ASSERT(chunk.data[column_id].GetVectorType() == duckdb::VectorType::
                       FLAT_VECTOR);
18             D_ASSERT(chunk.data[column_id].GetType().id() == duckdb::LogicalType::
                       BIGINT);
19             chunk.data[column_id].Sequence(idx, 1, ordinality);
20         }
21     }
22 };
23 }
```

Figure 3.31: *ordinality_data.cpp*

The `OrdinalityData` struct holds two **boolean** fields called `with_ordinality` and `reset`. `with_ordinality` is used to hold information on whether the `WITH ORDINALITY` feature has been activated in the query and `reset` is used to reset the **ordinality index** whenever new input is consumed when executing a `TableInOutFunction`. The struct also holds two fields of type `idx_t`. `column_id` is used as an index that points to the **ordinality** column within a `DataChunk` and `idx` represents the **ordinality index** which is used to populate the **ordinality** `Vector` with the correct values.

The method `SetOrdinality()` replaces the `PrepareOrdinality()` methods. It works by first making sure that the size of the `DataChunk`, and therefore the `Vectors` contained within, is greater than `0`. Otherwise, no work has to be done. If `reset` is set to `true`, `idx` is set to `1` and `reset` is set to `false` again to make sure the `idx` is only reset again during the next call when `reset` is manually set to `true` again outside of the method. We then assert that the `Vector` located at `chunk.data[column_id]`, which is the **ordinality** `Vector`, is of `VectorType::FLAT_VECTOR` and of `LogicalType::BIGINT`. Once we have made our assertions, we change the `FLAT_VECTOR` into a `SEQUENCE_VECTOR` by calling the `Sequence()` method the same as we did in Figure 3.7 and Figure 3.9. The adjusted code for `TableInOutFunctions` in Figure 3.10 can be seen in Figure 3.32. Note that we copy the `with_ordinality` and `original_ordinality_id` values from `bind_data` in line 13 – 14. Otherwise, the logic is the same except that the values `ord_index` and `ord_reset` have been renamed and are now part of `ordinality_data`.

```
#include "duckdb/function/table/ordinality_data.hpp"
[...]
class TableInOutLocalState : public OperatorState {
    public:
        TableInOutLocalState() : row_index(0), new_row(true) {
        }
        unique_ptr<LocalTableFunctionState> local_state;
        idx_t row_index;
        bool new_row;
        DataChunk input_chunk;
        OrdinalityData ordinality_data;
};
[...]
OperatorResultType PhysicalTableInOutFunction::Execute(DataChunk &chunk, [...])
    const {
        state.ordinality_data.with_ordinality = bind_data->with_ordinality;
        state.ordinality_data.column_id = bind_data->original_ordinality_id;
        if (projected_input.empty()) {
                [...]
                if (function.with_ordinality) {
                        state.ordinality_data.SetOrdinality(chunk, column_ids);
                        state.ordinality_data.idx += chunk.size();
                }
                return result;
        }
        if (state.new_row) {
                [...]
                state.ordinality_data.reset = true;
        }
    [...]
        if (state.ordinality_data.with_ordinality) {
                state.ordinality_data.SetOrdinality(chunk, column_ids);
        }
        if (result == OperatorResultType::FINISHED) {
                return result;
        }
        if (result == OperatorResultType::NEED_MORE_INPUT) {
                // we finished processing this row: move to the next row
                state.new_row = true;
        }
        state.ordinality_data.idx += chunk.size();
        return OperatorResultType::HAVE_MORE_OUTPUT;
}
```

Figure 3.32: *physical_tableinout_function.cpp*

For `TableFunctions` the adjustment for `PhysicalTableScan::GetData()` in Figure 3.7 is similar, see Figure 3.33. For determining the correct `Vector` index of the **`ordinality`** column, we are met with a new challenge as **`table functions`** support **`projection pushdown`** as well as **`filter pruning`**. **`Projection pushdown`** emits columns (`Vectors`) from the `DataChunk` that are not requested in the `SELECT` clause of a query (and not needed for any filter). This may change the index of the **`ordinality`** column. For example, if we call `READ_CSV()` on a **`csv file`** that contains two columns, `A` and `B`, then the index of column `A` is 0, the index of column `B` is 1 and the index of the **`ordinality`** column is `2` during the **`binding`** phase. The value of the `original_ordinality_id` field would be set

to 2 in Binder::BindTableFunctionInternal. However, if the query only selects column A as well as the ordinality column, then column B would be omitted from the DataChunk by the projection pushdown and the real index for the Vector that corresponds to the ordinality column would then be 1 instad of 2. To reflect this, the column_ids vector gets rearranged to hold the indexes in the correct order. If the query selects the ordinality column and then column B (SELECT ordinality, B FROM ...), the column_ids vector looks like this: [2,1]. If we want to access the ordinality Vector within column_ids, we would now have to use the index 0 instead of 2, as the Vectors have been rearranged in the DataChunk. Filter pruning makes the function use a different order denoted in the projection_ids vector.

We implement these checks in the constructor of TableScanLocalSourceState, see Figure 3.34. We first check if the WITH ORDINALITY clause has been invoked by checking if with_ordinality is true in line 6. If this is not the case then we do not need to do any further work. The only table functions that support filter pruning or projection pushdown are READ_CSV() and READ_PARQUET(), so we check if the function is either one of these in lines 7–10. If not, we can copy with_ordinality of the bind_data into with_ordinality of the ordinality_data and the original_ordinality_id into the column_id field as this index is still correct. If it is one of the functions that support filter pruning or projection pushdown, we check if projection_pushdown is true in line 11. If not, we also simply copy the values. If a projection pushdown occured, we need to make sure that the ordinality Vector is still present within the DataChunk. If not, we do not need to fill the Vector with any values, so we set with_ordinality to false unless we find the Vector again. We check for this Vector by comparing the indexes found in the column_ids vector with the original_ordinality_id. If filter_prune is true, we do this by looking up the correct indexes for column_ids in projection_ids. If we have found a match, we re-activate the WITH ORDINALITY feature by setting with_ordinality to true and copy the current index into column_id.

```
1  SourceResultType PhysicalTableScan::GetData(DataChunk &chunk, [...]) const {
2      function.function(context.client, data, chunk);
3
4      if (chunk.size() == 0) {
5          return SourceResultType::FINISHED;
6      } else {
7          if (state.ordinality_data.with_ordinality) {
8              state.ordinality_data.SetOrdinality(chunk, column_ids);
9              state.ordinality_data.idx += chunk.size();
10         }
11         return SourceResultType::HAVE_MORE_OUTPUT;
12     }
13 }
```

Figure 3.33: *physical_table_scan.cpp*

```
class TableScanLocalSourceState : public LocalSourceState {
public:
    TableScanLocalSourceState([...]) {
        [...]
        if (op.bind_data) {
            if (op.bind_data->with_ordinality) {
                if (op.function.name == "read_csv_auto"
                || op.function.name == "read_csv"
                || op.function.name == "read_parquet"
                || op.function.name == "parquet_scan") {
                    if (op.function.projection_pushdown) {
                        ordinality_data.with_ordinality = false;
                        if (op.function.filter_prune) {
                            for (idx_t i = 0; i < op.projection_ids.size(); i++) {
                                const auto &column_id = op.column_ids[op.
                                    projection_ids[i]];
                                if (column_id < op.names.size()
                                && op.bind_data->original_ordinality_id ==
                                    column_id) {
                                    ordinality_data.column_id = i;
                                    ordinality_data.with_ordinality = true;
                                    break;
                                }
                            }
                        } else {
                            for (idx_t i = 0; i < op.column_ids.size(); i++) {
                                const auto &column_id = op.column_ids[i];
                                if (column_id < op.names.size()
                                && op.bind_data->original_ordinality_id ==
                                    column_id) {
                                    ordinality_data.column_id = i;
                                    ordinality_data.with_ordinality = true;
                                    break;
                                }
                            }
                        }
                    } else {
                        ordinality_data.with_ordinality = true;
                        ordinality_data.column_id = op.bind_data->
                            original_ordinality_id;
                    }
                } else {
                    ordinality_data.with_ordinality = true;
                    ordinality_data.column_id = op.bind_data->
                        original_ordinality_id;
                }
            }
        }
        }

    unique_ptr<LocalTableFunctionState> local_state;
    OrdinalityData ordinality_data;
};
```

Figure 3.34: *physical_table_scan.cpp*

Apart from using the `original_ordinality_id` field to find the correct `column_id` for the **ordinality**

Vector, we also want to use original_ordinality_id to omit the correct column in the MultiFileReader ::CreateNameMapping() function. To do this, we need to propagate this information through the call stack in Figure 3.14 and Figure 3.24 the same way we did for with_ordinality. We make the appropriate changes to our code for the MultiFileReader class in Figure 3.35 as well as the initializing function calls for READ_CSV() and READ_PARQUET() in Figure 3.36 and Figure 3.37 by adding the constant parameter original_ordinality_id of type idx_t. We also finally enable table_function. supports_ordinality in Figure 3.37 line 14.

```
1  struct MultiFileReader {
2      DUCKDB_API static void CreateMapping(
3          [...],
4          const bool with_ordinality,
5          const idx_t original_ordinality_id);
6
7      static void InitializeReader([...], const bool with_ordinality,
8                                   const idx_t original_ordinality_id) {
9          [...]
10         CreateMapping([...], with_ordinality,
11                       original_ordinality_id);
12         [...]
13     }
14
15     static void CreateNameMapping([...],
16         const bool with_ordinality,
17         const idx_t original_ordinality_id);
18 }
```

Figure 3.35: *multi_file_reader.hpp*

```
CSVFileScan::CSVFileScan([...])
    : [...] {
    if (bind_data.initial_reader.get()) {
        [...]
        MultiFileReader::InitializeReader([...],
            bind_data.with_ordinality,
            bind_data.original_ordinality_id);
        return;
    } else if (!bind_data.column_info.empty()) {
        [...]
        MultiFileReader::InitializeReader([...],
            bind_data.with_ordinality,
            bind_data.original_ordinality_id);
        InitializeFileNamesTypes();
        return;
    }
    [...]
    MultiFileReader::InitializeReader([...],
        bind_data.with_ordinality,
        bind_data.original_ordinality_id);
    InitializeFileNamesTypes();
}
```

Figure 3.36: *csv_file_scanner.cpp*

```
static void InitializeParquetReader([...]) {
        [...]
        if (bind_data.parquet_options.schema.empty()) {
                MultiFileReader::InitializeReader(reader, parquet_options.
                    file_options, bind_data.reader_bind, bind_data.types,
                                                  bind_data.names,
                                                      global_column_ids,
                                                      table_filters, bind_data.
                                                      files[0],
                                                  context, bind_data.
                                                      with_ordinality, bind_data.
                                                      original_ordinality_id);
                return;
        }
    [...]
}
[...]
static TableFunctionSet GetFunctionSet() {
                [...]
                table_function.supports_ordinality = true;
        [...]
}
```

Figure 3.37: *parquet_extension.cpp*

After these changes, we are able to use `original_ordinality_id` in the `CreateNameMapping` function. We revert our changes made in Figure 3.19 and instead skip the column by breaking out of the current column iteration when the associated identifier equals `original_ordinality_id` as seen in Figure 3.38 line 6 – 10. Finally, we disable multithreaded execution for `TableInOutFunctions`

when `WITH ORDINALITY` is requested as parallel execution would require mutual exclusion access to `ordinality_data` over all `TableInOutLocalStates` or `LocalTableFunctionStates` as well as knowledge about the morsel-driven parallelism implemented in DuckDB that goes beyond the scope of this thesis. We achieve this by changing the `return value` for the `ParallelOperator()` declared in the `PhysicalTableInOutFunction` class, see Figure 3.39. Instead of always returning `true` this function now returns `false` when `with_ordinality` is `true`, making every `TableInOutFunction` a single threaded operation when the `WITH ORDINALITY` clause is given.

```
1  void MultiFileReader::CreateNameMapping([...],
2                                          const bool with_ordinality, const idx_t
                                               original_ordinality_id) {
3          [...]
4          for (idx_t i = 0; i < global_column_ids.size(); i++) {
5                  [...]
6                  auto global_id = global_column_ids[i];
7                  if (with_ordinality && global_id == original_ordinality_id) {
8                          // this is the column (added during binding) used to
                               display ordinality information and is therefore not
9                          // found in the file
10                         continue;
11                 }
12         [...]
13     }
14     [...]
15 }
```

Figure 3.38: *multi_file_reader.cpp*

```
1  class PhysicalTableInOutFunction : public PhysicalOperator {
2  public:
3          [...]
4          bool ParallelOperator() const override {
5                  return !function.with_ordinality;
6          }
7      [...]
8  }
```

Figure 3.39: *physical_tableinout_function.hpp*

4

Testing WITH ORDINALITY

In Section 1.4 we have defined the goal of this thesis as implementing WITH ORDINALITY for all table functions. In this chapter, we will find out if the queries found in Section 1.4 now work as expected when adding the WITH ORDINALITY clause as well as take a look at some additional test queries. An error in the sqllogictester, which is an SQL logic test suite adapted from SQLite, will force us to learn more about how DuckDB tests its queries. This leads us to the discovery of an oversight in the test logger as well as an oversight in our own code so far. We use the source code of the latest commit[1] within the pull request at the time of writing this thesis and use it to compile the debug version of DuckDB by issuing the command *make debug* in the terminal. This version of DuckDB will be used for all tests within this chapter.

4.1 GENERATE_SERIES() and RANGE()

The queries next to Table 4.1 produce the expected outcome, which is an additional column called ordinality that iterates over all elements starting at 1. Note that the column name is generate_series instead of range for the latter three queries. With the query next to Table 4.2 we check if the idx field in ordinality_data properly keeps track of the ordinality index even when dealing with a new Vector. By choosing a value of 3000, we exceed the standard Vector size of 2048 and see that our method produces the expected output, which is an ordinality column that can count higher than the length of a single Vector. The query next to Table 4.3 is used to check whether our method still works when columns are renamed using the AS clause and rearranged in the SELECT clause. The query produces the expected output, which is the range column renamed to x and the ordinality column renamed to y with their order switched.

[1] SHA checksum: 61786a2c43b41a0a6e83560e688fa41729d2ae68

```sql
-- The four queries below are effectively identical
SELECT * FROM RANGE(10) WITH ORDINALITY;
SELECT * FROM GENERATE_SERIES(9) WITH ORDINALITY;
SELECT * FROM GENERATE_SERIES(0,9) WITH ORDINALITY;
SELECT * FROM GENERATE_SERIES(0,9,1) WITH ORDINALITY;
```

range	ordinality
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10

Table 4.1: The resulting table for the queries to the left

```sql
SELECT * FROM RANGE(3000) WITH ORDINALITY;
```

range	ordinality
0	1
1	2
2	3
3	4
4	5
...	...
2996	2997
2997	2998
2998	2999
2999	3000

Table 4.2: The resulting table for the query to the left

```
SELECT y,x FROM RANGE(5) WITH ORDINALITY
    AS _(x,y);
```

y	x
1	0
2	1
3	2
4	3
5	4

Table 4.3: The resulting table for the query to the left

4.2 `REPEAT()` and `REPEAT_ROW()`

The queries next to Table 4.4 produce the expected output, which is the string `"This is a string"` repeated five times as well as an `ordinality` column next to it. The query next to Table 4.5 is used to test the same behaviour for a different type, in this case `Integer`, as well as testing if duplicating the `ordinality` column properly works. The query succeeds and produces the expected output as well, as does the query next to Table 4.6.

```
-- The the two queries below are
    effectively identical
SELECT *
FROM REPEAT('This is a string', 5)
    WITH ORDINALITY;

SELECT *
FROM REPEAT_ROW('This is a string',
    num_rows= 5)
    WITH ORDINALITY;
```

repeat	ordinality
This is a string	1
This is a string	2
This is a string	3
This is a string	4
This is a string	5

Table 4.4: The resulting table for the queries to the left

```
SELECT x,y,y FROM REPEAT(7,5) WITH
    ORDINALITY AS _(x,y);
```

repeat	ordinality	ordinality
7	1	1
7	2	2
7	3	3
7	4	4
7	5	5

Table 4.5: The resulting table for the query to the left

```
SELECT * FROM
    REPEAT(['this', 'is', 'array', '
        content'], 2)
WITH ORDINALITY;
```

repeat	*ordinality*
['this', 'is', 'array', 'content']	1
['this', 'is', 'array', 'content']	2

Table 4.6: The resulting table for the query to the left

4.3 `UNNEST()`

The query next to Table 4.7 produces the expected output as the `ordinality` column contains the correct values. The query next to Table 4.8 taken from Section 1.3 unnests the `Strings` correctly, allowing for the original order to be preserved as intended.

```
SELECT
    *
FROM
    UNNEST(['a','b','c'])
WITH ORDINALITY
AS
    _(letters,ordinality);
```

letters	*ordinality*
a	1
b	2
c	3

Table 4.7: The resulting table of the query to the left

```
SELECT
    id, substring, ordinality
FROM
    UnnestExample,
    UNNEST(string_to_array(
        elements,
        ','))
        WITH ORDINALITY
    AS _(substring,ordinality)
ORDER BY substring;
```

id	substring	ordinality
3	eigth	2
2	fifth	2
3	ninth	3
1	second	2
2	sixth	3
1	third	3
1	first	1
2	fourth	1
3	seventh	1

Table 4.8: The resulting table of the query to the left, executed on *UnnestExample* from Table 1.2

4.4 GLOB()

For the following test queries we will run **DuckDB** in a directory that contains files with these names in no particular order:

abc.txt
123.txt
1234.txt
cookbook.pdf
cookbook.txt

The results can be seen in Table 4.9, Table 4.10, Table 4.11 and Table 4.12. All queries return the correct file names as well as a properly populated `ordinality` column. The first query returns all files, the second query returns all text files containing any three characters in the filename, the third query returns all text files that start with either the letter `a` , `b` or `c` and the fourth query returns all text files that do not start with the letter `a` , `b` or `c`.

```
1 SELECT * FROM GLOB('*') WITH ORDINALITY;
```

glob	*ordinality*
abc.txt	1
123.txt	2
1234.txt	3
cookbook.pdf	4
cookbook.txt	5

Table 4.9: The resulting table of the query to the left

```
1 SELECT * FROM GLOB('???.txt') WITH
      ORDINALITY;
```

glob	*ordinality*
abc.txt	1
123.txt	2

Table 4.10: The resulting table of the query to the left

```
1 SELECT * FROM GLOB('[a-c]*.txt') WITH
      ORDINALITY;
```

glob	*ordinality*
abc.txt	1
cookbook.txt	2

Table 4.11: The resulting table of the query to the left

```
SELECT * FROM GLOB('[!a-c]*.txt') WITH
    ORDINALITY;
```

glob	ordinality
123.txt	1
1234.txt	2

Table 4.12: The resulting table of the query to the left

4.5 READ_CSV() and READ_CSV_AUTO()

We execute the queries next to Table 4.13 on *table.csv* (see Table 1.4) from Section 1.4. The output for both queries is the same as in Table 1.12, except that the `ordinality` column containing the correct values has been appended to the table as well. With the next two queries seen next to Table 4.14, we check if `READ_CSV()` and `READ_CSV_AUTO()` behave correctly when reading from multiple files. As they are `TableFunctions` instead of `TableInOutFunctions`, they should not reset the `ordinality` index when reading from a new file and instead treat the function as if reading from a single file. The output correctly reflects this. The query next to Table 4.15 is used to check if `READ_CSV()` and `READ_CSV_AUTO()` can handle inputs that stretch over multiple `Vectors`. We do this by first creating a large table and exporting it as a `csv file` which can then be used as an input. As the `ordinality` column correctly counts up to value `5000` both times, `WITH ORDINALITY` passes the test.

```
SELECT * FROM READ_CSV('table.csv', delim
    =',')
    WITH ORDINALITY AS table;
SELECT * FROM READ_CSV_AUTO('table.csv')
    WITH ORDINALITY AS table;
```

numbers	chars	ordinality
11	a	1
22	b	2
33	c	3
44	d	4

Table 4.13: The resulting table of the queries to the left

```
SELECT * FROM READ_CSV(['table.csv','table.csv'], delim=',')
    WITH ORDINALITY AS table;
SELECT * FROM READ_CSV_AUTO(['table.csv','table.csv'])
    WITH ORDINALITY AS table;
```

numbers	chars	ordinality
11	a	1
22	b	2
33	c	3
44	d	4
11	a	5
22	b	6
33	c	7
44	d	8

Table 4.14: The resulting table of the queries to the left

```
CREATE TABLE my_tbl (a int, b int, c varchar);

INSERT INTO my_tbl SELECT range, 5000 - range, 'string' FROM range(5000);

COPY my_tbl TO 'output.csv' (HEADER, DELIMITER ',');

SELECT * FROM READ_CSV_AUTO('output.csv', header=true)
    WITH ORDINALITY AS _(X,Y,Z);
SELECT * FROM READ_CSV('output.csv', header=true)
    WITH ORDINALITY AS _(X,Y,Z);
```

x	y	z	ordinality
0	5000	'string'	1
1	4999	'string'	2
2	4998	'string'	3
...	...	...	...
4997	3	'string'	4998
4998	2	'string'	4999
4999	1	'string'	5000

Table 4.15: The resulting table of the queries to the left

4.6 READ_PARQUET() and PARQUET_SCAN()

We create a **parquet file** that is equal to the **csv file** seen in Table 1.4 and modify the queries from the previous section, see the queries next to Table 4.16 and Table 4.17, to test these functions for the same behaviour. Both functions pass the test as well.

```
SELECT * FROM READ_PARQUET(['table.
    parquet','table.parquet'])
    WITH ORDINALITY AS table;
SELECT * FROM PARQUET_SCAN(['table.
    parquet','table.parquet'])
    WITH ORDINALITY AS table;
```

numbers	chars	ordinality
11	a	1
22	b	2
33	c	3
44	d	4
11	a	5
22	b	6
33	c	7
44	d	8

Table 4.16: The resulting table of the queries to the left

```
CREATE TABLE my_tbl (a int, b int, c
    varchar);

INSERT INTO my_tbl SELECT range, 5000 -
    range, 'string' FROM range(5000);

COPY my_tbl TO 'output.parquet';

SELECT * FROM PARQUET_SCAN('output.
    parquet')
    WITH ORDINALITY AS _(X,Y,Z);
SELECT * FROM READ_PARQUET('output.
    parquet')
    WITH ORDINALITY AS _(X,Y,Z);
```

x	y	z	ordinality
0	5000	'string'	1
1	4999	'string'	2
2	4998	'string'	3
...	...	...	...
4997	3	'string'	4998
4998	2	'string'	4999
4999	1	'string'	5000

Table 4.17: The resulting table of the queries to the left

4.7 Fixing the `test logger`

Even though all our tests manually run without errors, we notice that some `READ_CSV()` tests fail because of a failed checksum when running them through the `sqllogictester`. This `sqllogictester` can be used to write `test files` such as the one seen in Figure 4.1. The `statement ok` keyword can be prepended to an `SQL statement` and checks if this statement executes without error. The `query` keyword, followed by the amount of columns to be expected by the result, can be prepended to a query to check it against the output that follows the four *minus* symbols (-) after the query. Only if both the query result and the expected result are the same will the test succeed. Instead of writing this output out entirely, there is also an option to use the `checksum` keyword and provide a `hash value` for the expected result table instead.

```
# name: test/sql/table_function/readcsv_ordinality.test
# description: Table functions
# group: [table_function]

statement ok
CREATE TABLE my_tbl (a int, b int, c varchar);

statement ok
INSERT INTO my_tbl SELECT range, 10 - range, 'string' FROM range(10);

statement ok
COPY my_tbl TO '__TEST_DIR__/output.csv' (HEADER, DELIMITER ',');

query IIII
select * from read_csv_auto('__TEST_DIR__/output.csv', header=true) with ordinality
     AS _(X,Y,Z);
----
0       10      string  1
1       9       string  2
2       8       string  3
3       7       string  4
4       6       string  5
5       5       string  6
6       4       string  7
7       3       string  8
8       2       string  9
9       1       string  10
```

Figure 4.1: *An example `test file` that fails the `sqllogictester` test*

```
void SQLLogicTestLogger::WrongResultHash(QueryResult *expected_result,
    MaterializedQueryResult &result,
                                         const string &hashval, const string
                                             expected_hash) {
    PrintHeader("Expected result: \n");
    if (expected_result) {
        expected_result->Print();
    } else {
        std::cerr << "???" << std::endl;
    }
    PrintErrorHeader("Wrong result hash!");
    PrintLineSep();
    PrintSQL();
    PrintLineSep();
    PrintHeader("Expected result hash:");
    PrintHeader(expected_hash);
    PrintLineSep();
    PrintHeader("Actual result hash:");
    PrintHeader(hashval);
    PrintLineSep();
    result.Print();
}
```

Figure 4.2: *sql_test_logger.cpp*

When receiving feedback from the **sqllogictester**, however, we notice that the feedback does not contain the actual **hash value** of the real query result, only the **hash value** that we provide in the **test file**. We search for the **Strings** contained in the feedback, find the corresponding code and notice that there is an oversight as the actual **result hash** not only does not get printed, but is not even passed to the responsible function in the first place. We fix this by adding lines $2-3, 13-14$ and $16-17$ to `SQLLogicTestLogger::WrongResultHash()`, see Figure 4.2. This also means that we have to adjust the function declaration, see Figure 4.3 lines $6-7$. We can then pass the arguments to the function from within the **result helper** as seen in Figure 4.4.

```
1  class SQLLogicTestLogger {
2  public:
3          [...]
4          void WrongResultHash(QueryResult *expected_result,
5          MaterializedQueryResult &result,
6          const string &hashval,
7              const string expected_hash);
8  }
```

Figure 4.3: *sql_test_logger.hpp*

```
1  bool TestResultHelper::CheckQueryResult(const Query &query, ExecuteContext &context
       ,
2      duckdb::unique_ptr<MaterializedQueryResult> owned_result) {
3      [...]
4      if (hash_compare_error) {
5                          [...]
6                          logger.WrongResultHash(expected_result,
7                  result,
8                  hash_value,
9                  expected_hash);
10             return false;
11     }
12 }
```

Figure 4.4: *result_helper.hpp*

After fixing this problem, we can see that the **result hash** differs from the **actual hash**. After investigating further, we figure out how the **sqllogictester** runs its tests. Besides executing the query and checking its result against the **result hash**, the query is also **serialized** and then **deserialized**. This **deserialized query** is then run and gets compared to the **result hash** as well. **Serializing** a query means converting the internal query structure of **DuckDB** back into the original **query string**, while **deserializing** means parsing a **query string** and representing it using the internal structure. This means that the **serialization** process for functions containing `WITH ORDINALITY` does not work properly yet. In some files starting with the keyword `"serialization"` we find a comment that reads `"This file is automatically generated by scripts/generate_serialization.py \\ Do not edit this file manually, your changes will be overwritten"`. After running this script, we find that lines 5 and 12 in Figure 4.5 have been automatically added.

This means that the `with_ordinality` field that we added to `TableFunctionRef` in the previous section has not been properly **serialized** and **deserialized**. After compiling **DuckDB** again, the **sqllogictester** no longer throws an error. (Note that these changes are already part of the **DuckDB** version compiled from the source code of the commit used in this chapter.)

```
1  void TableFunctionRef::Serialize(Serializer &serializer) const {
2          TableRef::Serialize(serializer);
3          serializer.WritePropertyWithDefault<unique_ptr<ParsedExpression>>(200, "
               function", function);
4          serializer.WritePropertyWithDefault<vector<string>>(201, "column_name_alias
               ", column_name_alias);
5          serializer.WritePropertyWithDefault<bool>(202, "with_ordinality",
               with_ordinality);
6  }
7
8  unique_ptr<TableRef> TableFunctionRef::Deserialize(Deserializer &deserializer) {
9          auto result = duckdb::unique_ptr<TableFunctionRef>(new TableFunctionRef());
10         deserializer.ReadPropertyWithDefault<unique_ptr<ParsedExpression>>(200, "
               function", result->function);
11         deserializer.ReadPropertyWithDefault<vector<string>>(201, "
               column_name_alias", result->column_name_alias);
12         deserializer.ReadPropertyWithDefault<bool>(202, "with_ordinality", result->
               with_ordinality);
13         return std::move(result);
14 }
```

Figure 4.5: *serialize_tableref.cpp*

5 Conclusion and Future Work

5.1 Conclusion

We have seen that WITH ORDINALITY can be used to preserve the order of elements found in generic data types when converting them to a relational format. We have defined the table functions for which we want to implement WITH ORDINALITY in DuckDB and taken a look at how DuckDB works internally. We found out that DuckDB , as an OLAP oriented DBMS, uses Vectors collected in DataChunks that represent a horizontal slice of a table to move data through the execution engine. We have seen the different stages of the execution engine that a query goes through and have used this knowledge to find a starting point. By reactivating the already present WITH ORDINALITY functionality in the Parser, we were able to immediately parse query strings that request our feature. We first implemented naive logic that assumed that the ordinality column is always found in the last Vector within a DataChunk. The functions we then wrote were able to correctly fill the ordinality column with data as long as no projections were present. We then fixed the READ_CSV() function by omitting the ordinality column from being considered when creating a mapping from the names found in the csv files to the column names found in the SELECT clause and made it possible to activate or deactivate our feature for specific table functions. We refactored our code and had to make a decision between dynamic pointer castings and carrying additional information in the FunctionData of functions that do not need it because the bind data of UNNEST() has a different inheritance as the other table functions. We decided on the latter approach. We made sure to run WITH ORDINALITY single threaded and tested our code against real queries after compiling DuckDB . We found an oversight that we fixed and successfully implemented WITH ORDINALITY for table functions in DuckDB .

5.2 Future Work

As WITH ORDINALITY only supports single threaded execution right now, a goal for the future is to support multithreading. This could potentially be achieved by moving the necessary fields from local table function states into global table function states, which hold information about all morsels used in the morsel-driven parallelism, and using mutexes to write the values. The inheritance of the UNNEST() function could be sensibly changed to make room for efficiency improvement as well. Finally, the constructor of TableScanLocalSourceState still manually differentiates between functions by using string literals. This could be avoided by adding another field to table functions that denotes whether the function supports filter pruning and projection pushdown, as new functions could then easily be implemented.

Bibliography

[1] *DuckDB 0.10 Documentation*. https://duckdb.org/docs/index.html.

[2] Laurens Kuiper, Mark Raasveldt, and Hannes Mühleisen. "Efficient External Sorting in DuckDB." In: *BICOD*. 2021, pp. 40–45.

[3] Laurens Kuiper and Hannes Mühleisen. "These Rows Are Made for Sorting and That's Just What We'll Do". In: *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE. 2023, pp. 2050–2062.

[4] Viktor Leis et al. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 743–754.

[5] Till Döhmen, Hannes Mühleisen, and Peter Boncz. "Multi-hypothesis CSV parsing". In: *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. 2017, pp. 1–12.