Smoked Duck

SQL Execution of Smoked Duck

by

Sughosh Kaushik

Thesis Advisor : Eugene Wu

Title : Associate Professor of Computer Science

Abstract

Data visualization entails data analysis over massive datasets, and many such visualizations involve interactions with visualization objects. The involvement of humans in such visualizations warrants the interactions to be as responsive as possible. In this regard, the highly responsive interactions can be characterized as computing provenance at interactive speeds of the underlying data in the database system. The recent trend of analytical database systems moving towards vectorized execution engines motivated DuckDB, a vectorized embedded analytical database. Smoked Duck builds a fast provenance system over DuckDB. How Smoked Duck captures provenance necessitates changes to DuckDB's execution engine to support querying the captured provenance. This work makes SQL interfaces on DuckDB for the provenance data captured by Smoked Duck to enable provenance querying at interactive speeds.

A thesis presented for the fulfillment of the requirements for the degree of
Masters in Computer Science on 6/02/2022

Department of Computer Science, Columbia University

Acknowledgments

I started my journey in Wu's class six double one three
Joined the WuLab to get the research pree
Preparing Smoked Duck took a lot of time
With head chefs Haneen and Charlie making it taste fine
Grateful to my parents for teaching me the value of every dime
To my grandmother raising me to have my chime
And to my friends who helped me climb
I could not have done it without Master Wu
To all the folks in the lab, Thank You

An ode to Anime that has kept me sane- Dropping a quote from my favorite anime character
"The mark of a true shinobhi is to never seek glory. They protect from shadows"
- Itachi from Naruto

Contents

List of Figures

Chapter 1

Introduction

1.1 Context and Relevance

In recent years, human-in-the-loop data analysis has received significant attention. This field embraces all the data problems involving human interaction. Extracting insights from data, building knowledge graphs, and creating data repositories are a few problems that human-in-the-loop data analysis addresses. Under the umbrella of extracting insights from data, data visualization is a hugely studied topic. Data visualization is an increasingly growing industry, and the market is expected to reach USD 19.20 billion by 2027 [1].

In Big Data, data visualization tools and technologies are vital to analyze massive amounts of information and make decisions driven by data. Tableau, QlikView, and Microsoft Power BI are a few among the many data visualization tools that offer users the capability to explore, manipulate, and interact with data through dynamic charts, graphs, and rich visualization objects. Figure 1.1 shows an interactive data visualization map borrowed from the Tableau Public gallery featured in "Viz of the Week." The visualization analyzes public holidays worldwide and develops a map showing where a different holiday is held each day during the year. There are two interactions viable in the visualization:- 1) you can select a country on the map to see which holidays they celebrate shown in Figure 1.2, and 2) you can select one of the bars to see which countries celebrate that holiday, shown in Figure 1.3. Both these interactions in the pixel space are mapped onto data space for calculating the interactions. The calculations involve computing SQL queries on the underlying data . The first interaction of selecting a country is introducing a filter predicate to the original query that filters on the selected country name. Similarly, the second interaction filters on the date(day) selected.

The visualization interactions we witnessed in the previous paragraph involve augmenting original SQL queries with additional operators and recomputing them. These operators include filters, aggregations, group bys, and order by. To make the interactions as responsive as possible, the computation

Figure 1.1: Data Visualization

Figure 1.2: Interactive Data Visualization 1

Figure 1.3: Interactive Data Visualization 2

of the SQL queries must be as fast as possible. Due to the increasing size of data sets, recomputing the original query with additional operators make the interactions with visualizations less responsive. In this regard, Fotis and Eugene [5] established a connection between these interactions with visualization and the provenance of the underlying data. With this, the interactions could be rendered by computing the provenance of underlying data. Essentially the selections in the interactions described in Figures 1.2 and 1.3 maps to calculate the backward trace. Section 2.4 describes the operation of backward trace in detail.

Now that we understand that interactions in visualizations are mapped to calculating provenance, there is a need for these calculations to be fast since the interactions must be responsive. This motivates the necessity for systems that compute provenance as quickly as possible. In this regard, Fotis and Eugene developed SMOKE[10] that captures and queries lineage at interactive speeds. SMOKE is a handwritten custom engine designed for row-based execution engines. Smoked Duck [7] inspired by SMOKE is a provenance capture and querying system built on DuckDB [11], a vectorized execution engine used for analytical queries. Smoked Duck uses specific custom hacks to query provenance and requires changes to DuckDB's native execution engine to support the custom hacks and query lineage. This work revolves around practically implementing these changes onto DuckDB and leveraging its fast query execution engine to compute lineage at interactive speeds.

1.2 provenance capture and querying systems

The previous section described two provenance capturing systems: SMOKE [10] and Smoked Duck [7]. Several systems support managing provenance information, from provenance capture to provenance querying. We restrict our study of provenance systems only to relational systems relevant to the thesis.

Relational Systems: Provenance Extension on Relational Model built on PostgreSQL supports provenance through instrumentation and was the first system to support nested sub-queries. GProM [3] - Generic Provenance Middleware supports multiple languages on frontend and backend databases. A system called PUG [8] extends GProM to why-not provenance. The Orchestra update exchange system tracks semiring provenance for schema mappings in a distributed setting. Trio[2] is a system for probabilistic data management that eagerly captures the provenance of SQL queries. SMOKE[10] is an in-memory provenance capture system whose capture implementation utilizes data structures used during query processing. Smoked Duck [7] is an in-memory provenance capture system built on DuckDB that utilizes the data structures generated as a part of late materialization in vectorized engines to capture provenance. These vectorized engines support OLAP workloads, and this work built on DuckDB supports all provenance-related computations on OLAP workloads.

1.3 OLAP systems

While most provenance systems have been implemented on OLTP row-based execution engines, a few systems implement provenance capture on OLAP execution engines. Most systems that consider OLAP workloads are often based on the same traditional row-store systems used for OLTP workloads or on distributed query processing systems such as Apache Spark. While these systems give insight into provenance capture and querying, they will not meet the performance demand of OLAP applications. Database systems like MonetDB, HyPer or DuckDB support the performance ask of OLAP applications by using optimized query processing engines for OLAP specific workloads. DuckDB[11] is an Embeddable Analytical Database that supports OLAP workloads. Smoked Duck [7] leverages the late materialization aspect of DuckDB is capture lineage and supports querying the captured lineage.

1.4 Research Questions

This thesis will enable DuckDB's native execution engine to execute lineage queries accommodating the custom hacks proposed by Smoked Duck, a highly efficient OLAP provenance system. Smoked Duck is a lineage capture and querying system developed at the WuLab built on DuckDB. DuckDB, an analytical embedded database, uses modern techniques such as a vectorized execution engine and a columnar storage layout. To the best of our knowledge, no previous

research on creating a SQL interface for vectorized engines has been done before. Regarding the performance, the work in this thesis mainly concentrates on leveraging the vectorized engine to serve lineage queries. It will research how easily, in the context of lines of code and utilizing the API contracts of DuckDB; we can change the execution engine of DuckDB to serve lineage queries. For example, design decisions that require the complete rewrite of the internals of the DuckDB execution engine will be dropped.

Based on this goal we define the following research questions:

1) How to enable DuckDB's native execution engine to execute queries on the lineage data captured by Smoked Duck?

2) How to implement the SQL interface to capture lineage data into DuckDB's execution engine by maintaining optimal performance?

3) Minimize the changes introduced to DuckDB's native execution engine-use DuckDB's existing API extensions to make the integration of lineage querying into the native execution engine as seamless as possible.

1.5 Outline

This thesis is structured as follows. In Chapter 2, details about provrnance essential to the understanding of this thesis is provided. In Chapter 3 an overview of the DuckDB is provided explaining the data model and execution model. In Chapter 4, the design of Smoked Duck is explained with the custom hacks that Smoked Duck employs to answer lineage queries. Additionally, the changes required in DuckDB to accommodate the custom hacks are motivated. Chapter 5 explains the architecture of the changes to DuckDB's native execution engine. Chapter 6 deals with the performance analysis of the lineage querying over TPC-H queries on DuckDB's execution engine. Chapter 7 concludes with describing all the claims made in the thesis and how each of these claims are supported with relevant evidence.

Chapter 2

Provenance

2.1 A Gentle Introduction to Provenance

Many data management tasks involve investigating the origins and the history of data. The life cycle of data is called data provenance. In this work, we concentrate on data provenance in the context of database management systems. There are several notions of provenance described in database literature, and this work emphasizes why provenance. Below is a brief description of why provenance that is relevant to the discussion of SQL execution of Smoked Duck.

2.2 Why Provenance

In relational databases, Cui et.al[14] formalized why - provenance or Lineage to be the association of tuples in the output relation of a query to a set of tuples in the input relation. Why-Provenance is also referred to as fine-grained provenance or Lineage. Figure 2.1 explains the concept of Lineage in the context of Database systems. Lineage is defined over tuples of a given relation. In Figure 2.1, we see the input relation Personal_Info, query Q that filters on age column resulting in the output relation. The Lineage of query Q of the 0th and 1st output tuples refers to the 0th and the 2nd input tuples, respectively. The Lineage of Q depicts the relational table representation of Lineage with columns oid and iid, which encode information regarding what input IDs contribute to output IDs.

2.3 Lineage Capture

Some systems adapt provenance capture and choose lazy implementation, eager implementation, or both, depending on how much acceptable overhead and when the lineage data is used. The lazy approach postpones capture after base query execution, an eager approach that calculates lineage during base

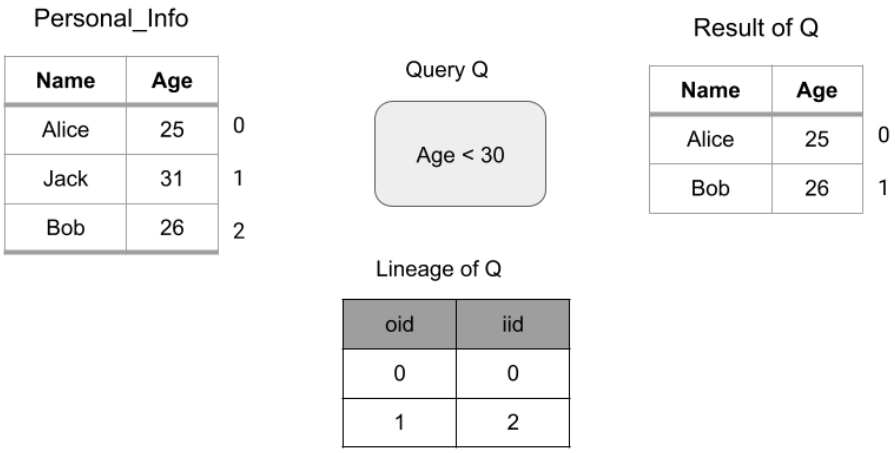

Figure 2.1: Fine-grained provenance

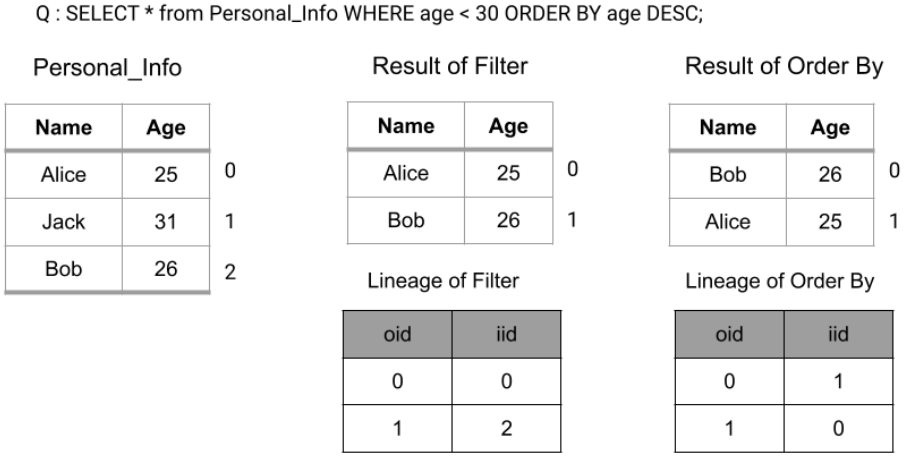

Figure 2.2: Lineage Capture

query execution. Lineage capture logic in these systems falls into two logical or physical categories. Logical approaches use query rewriting [9, 6] and UDFs [13] to annotate output or to store lineage in a separate table and are outside the scope of this work. Systems like Smoked Duck[7] use a physical approach where the physical operators are instrumented to capture fine-grained lineage. Chapter 4 extensively describes the lineage capture mechanism of Smoked Duck. Lineage capture overhead adds to the execution time of base query in physical

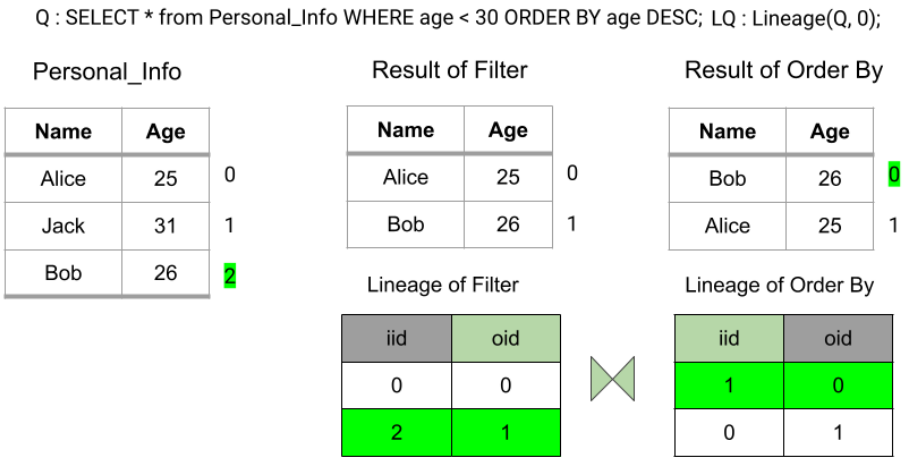

Figure 2.3: Lineage Querying

approaches and thus the lineage capture overhead must be minimized.

Figure 2.2 explains what lineage capture means, which follows the definition of Lineage. Suppose we have the Query Q, which has more than one operator: Filter and Order By is executed on the input relation Personal_Info. The result of each operator depicted in Figure 2.2 is annotated with IDs. The Lineage of each operator in the query is called operator lineage. Capturing the Lineage of all the operators in the query refers to lineage capture. The Ids associated with the tuples of the filter operator make up the iids of the Lineage of Order By. The Lineage of the filter operator has 0 and 1 as oids. The corresponding iids are 0 and 2—similarly, the Lineage of Order By operator. The Lineage of the filter operator and order by operator make up the lineage capture.

2.4 Lineage Querying

There are two aspects of Lineage querying- querying operator lineage and end-to-end lineage. Let's assume the Query Q showed in Figure 2.3, which queries the operator lineage. Suppose the end-user shoots a query " Select * from Lineage_of_Filter "the user wants to query the operator lineage of the filter operator, as shown in Figure 2.3. This is querying operator lineage. Suppose there is a lineage query LQ, as shown in Figure 2.3 that takes in input the base query string and the output ID 0, which asks for the tuples in the input relation that contribute to the output tuple with the output ID 0 we term LQ as an end-to-end lineage query. Generally, most applications warrant computing end-to-end lineage for a single output ID or a set of output IDs. Hence we assume a base case of computing lineage of a single output ID. To compute the end-to-end lineage, the operator lineage tables must be joined. In the example

shown in Figure 2.3, the lineage of filter and lineage of Order By is joined on the columns oid and iid, respectively marked in green. The iid 2 of the lineage of the filter operator passes the join condition and represents the input ID that contributes to the output ID 0, both marked with green. In applications like interactive visualization, fast computation of end-to-end lineage is vital to provide a seamless interaction with the visualization objects.

Chapter 3

DuckDB

DuckDB is an embedded analytical database system that uses a vectorized pull-based engine. It also supports a vectorized push-based execution engine, but we consider the pull-based engine implementation for this thesis. In this section, we introduce the internals of DuckDB- the data model and the execution model to understand the system on which Smoked Duck is built.

3.1 Data Model

First we understand the data model of DuckDB and then go over the execution model of DuckDB.

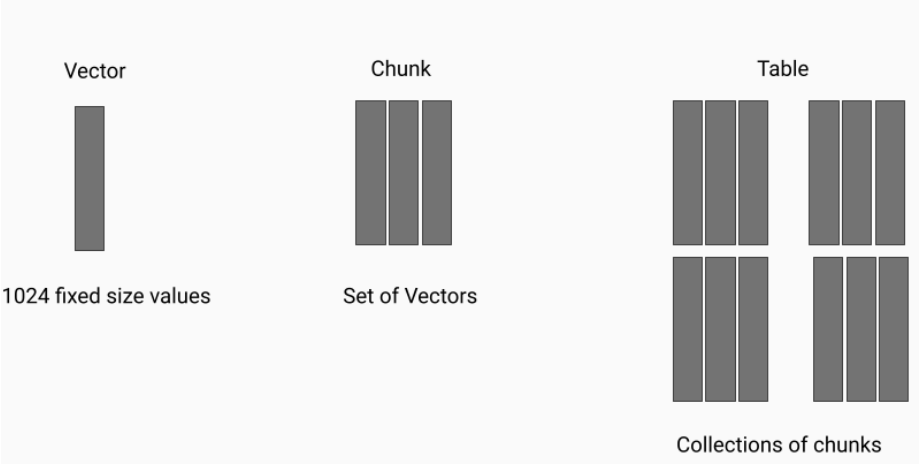

Figure 3.1: Data Model of DuckDB

A vector in DuckDB is a fixed size set of values. Values are logical types supported by DuckDB. In general, the size of the vector is set to be 1024. Next,

we understand the idea of chunks. Chunks in DuckDB are a set of vectors. They represent a horizontal subset of a result set or query intermediate or base table[11]. A table representation in DuckDB is a collection of chunks. This hierarchy of data models has a set of values representing vectors, a set of vectors representing chunks, and a set of chunks representing tables. This hierarchy is depicted in Figure 3.1.

Adding chunks to tables in DuckDB follows by merging the chunks if they are partially filled. The merging of chunks is done to promise a constant time lookup in the SetValue() and GetValue() methods of tables in DuckDB. This phenomenon is described in Figure 3.2. Data Chunks 1 and 2 are appended to the Chunk Collection collection. The first Data Chunk is appended as it is. Now the collection has a partially filled chunk. When the second partially filled Data Chunk is added to the collection, the collection merges the second chunk with the chunk it's already holding to promise a constant-time lookup.

Figure 3.2: Appending chunks to a table in DuckDB

3.2 Execution Model

DuckDB supports both pull-based and push-based execution engines. For this work, we consider a single-threaded pull-based execution engine. Specifically, this thesis builds on a vectorized pull-based execution engine. In a vectorized pull-based execution engine, the root operator node recursively calls the child operator node until the child operator stops providing data to the calling operator. Figure 3.3 shows an example of a vectorized pull-based execution engine. The Aggregation operator at the root calls the next() method on the Filter operator, which calls the Scan operator. The scan operator scans the persistent

data in DuckDB and returns a vector of N tuples to the calling operator. This mechanism is termed as vectorized pull-based execution.

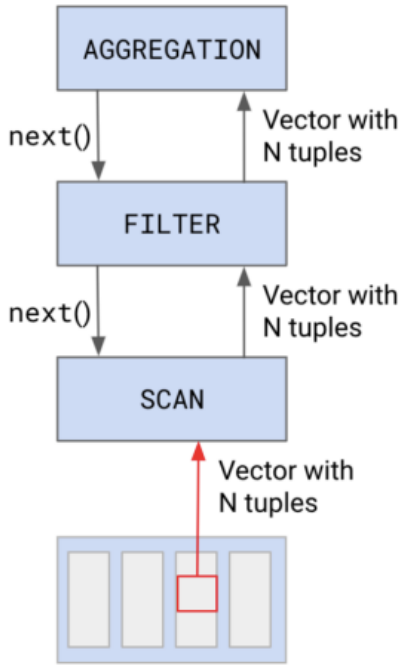

Figure 3.3: Vectorised pull-based execution engine

We see an example of the vectorized execution in Figures 3.4 and 3.5. In figure 3.4, there is an input relation Personal_Info constituting 2 data chunks with three tuples, each having columns Name and Age. Each data chunk is executed through a pipeline. In figure 3.5, we see that each data chunk goes through a filter operator where the filter predicate filters on the age column. DuckDB generates a Selection Vector data structure that encodes input offsets that pass the filter predicate. For the first data chunk, the selection vector encodes the values 0 and 2 representing Alice and Jack, whose age is 25 and 26, respectively, to pass the filter predicate of age less than 30. The selection vector is then sliced with the input relation to materialize the output. In the next section, we introduce selection vectors in DuckDB. Like selection vectors, DuckDB generates artifacts during operator execution that encode local IDs or global IDs. Local IDs are IDs relative to a chunk, and global IDs are those relative to a table. Local IDs added to the respective chunk offsets give the global IDs.

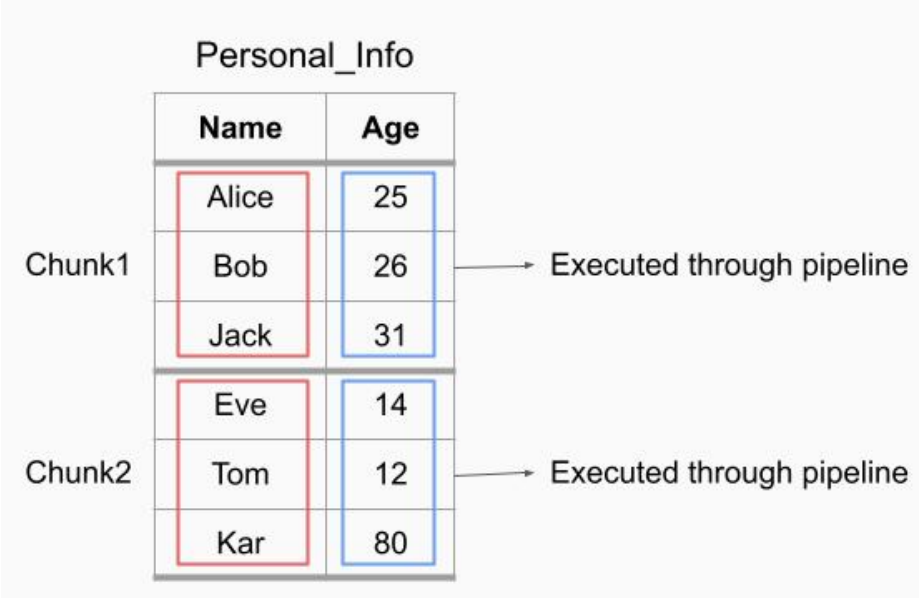

Figure 3.4: Pipelined Execution

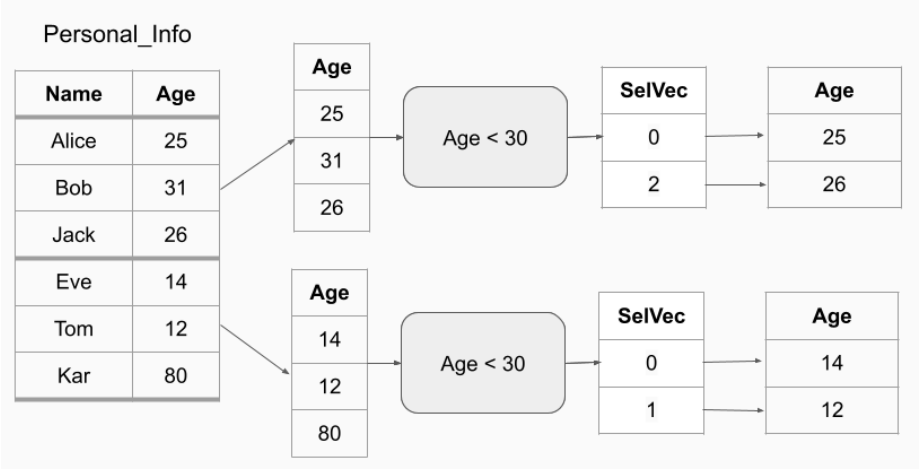

Figure 3.5: Artifacts of DuckDB generated during operator execution

3.3 Selection Vector

Selection vectors are data structures that are generated in vectorized or columnar engines as a late materialization strategy[4]. The selection vectors encode

either bits or input offsets depending on the design of the execution engine. The selection vectors help to reduce data movement by not fully materializing rows from input columns in columnar execution engines. The selection Vector in DuckDB is a 32-bit unsigned integer pointer that points to an array of 4-byte integers.

3.4 Generic Interfaces

DuckDB exposes generic interfaces for all the entities involved in the database execution. In the light of this work, we consider two generic interfaces, the index interface, and the operator interface. All indexes defined in DuckDB must adhere to the contracts prescribed by the index interface. The indexing interface has a type that determines the type of index. The index interface also exposes specific methods like index_scan, searchForValue, and others that must be implemented if a new index is introduced in DuckDB.

Similarly, every operator implemented in DuckDB extends the PhysicalOperator interface. The PhysicalOperatorState is embedded in the PhysicalOperator interface. The PhyiscalOperatorState has access to the chuck that the child operator in the base query plan has and the state of the child operator, which is again a PhysicalOpeartorState.

3.5 PRAGMA function

The PRAGMA statement is a SQL extension adopted by DuckDB from SQLite. PRAGMA statements can be issued similarly to regular SQL statements. PRAGMA commands may alter the internal state of the database engine and can influence the subsequent execution or behavior of the engine. If the need exists to skip the internals of the database like the binder, planner, optimizer, etc., then PRAGMA functions can be defined and declared.

Chapter 4

Smoked Duck

This chapter introduces Smoked Duck, a lineage capture and querying system built on DuckDB. We describe how Smoked Duck captures and queries lineage, the custom hacks that Smoked Duck employs to minimize the lineage capture and query overhead, and finally, how the gaps that the native execution engine must fulfill to accommodate the custom hacks and execute lineage queries.

4.1 Lineage capture

In this section, we go about the lineage capture mechanism in Smoked Duck. As pointed out in the Execution model of DuckDB, the artifacts generated during DuckDB's operator execution encode lineage. But these artifacts are flushed out of memory, and the lineage encoded is lost. Smoked Duck pins these artifacts into memory and captures lineage. As noted in the previous section, the artifacts either encode the local IDs or the global IDs. We see an example of lineage capture of the filter operator and the aggregation operator, which encode local IDs and global IDs.

Figures 4.1 and 4.2 explain the process of selection vector capture employed by Smoked Duck in DuckDB. The input relation Personal_Info has 2 data chunks with three tuples each. The first chunk of data is executed through the first pipeline. DuckDB generates the selection vector that encodes IDs 0 and 2 that refer to Alice and Jack that pass the filter predicate. Smoked Duck pins this Selection Vector into memory with the corresponding chunk offset and wraps it into a data structure called Lineage Data. We use this chunk offset to calculate the global IDs.

Similarly, data chunk 2 passes through the filter operator, and DuckDB generates a selection vector encoding input IDs 0 and 1 that pass the filter predicate. Smoked Duck pins this selection vector into memory with the chunk offset of 1024. Thus by pinning the selection vectors of all input data chunks of DuckDB into memory, Smoked Duck captures operator lineage, as shown in figure 4.3.

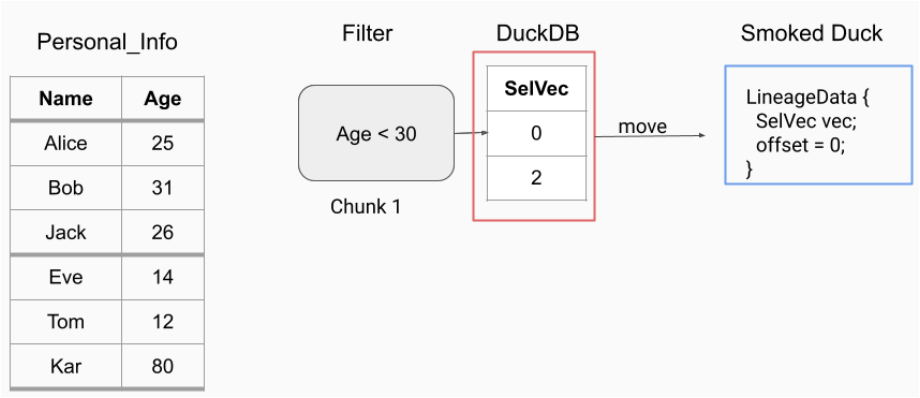

Figure 4.1: Lineage Capture of Filter Operator

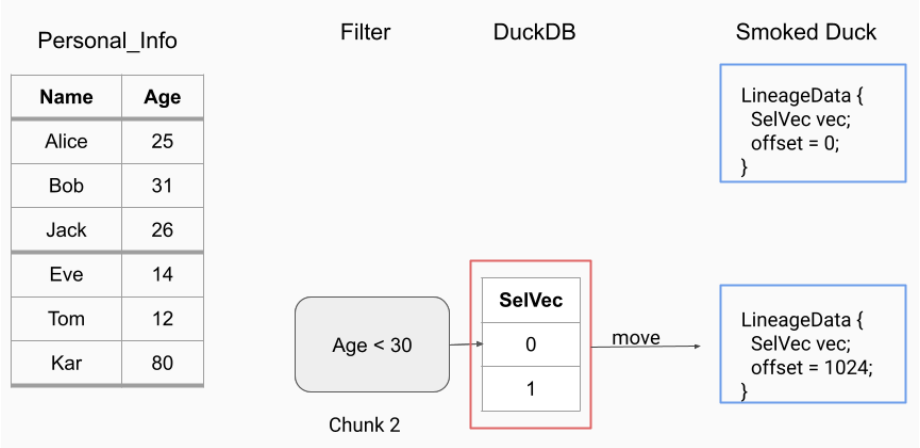

Figure 4.2: Lineage Capture of Filter Operator

We look at another example of operator lineage capture in Smoked Duck for the simple aggregation operator. In figure 4.4, the average of the age column in the input relation is calculated. We see that all the input tuples are consumed for the operator execution for the simple aggregate operator. Hence the lineage of the output relation in the case of a simple aggregate operator is all the input tuples. Thus the IDX structure encodes all the global input IDs pinned into memory, and the simple aggregate operator lineage is thus captured.

Now that we understand how Smoked Duck captures lineage, we look into how the captured lineage is queried and motivate the work of this thesis.

Figure 4.3: Operator Lineage captured for all data chunks

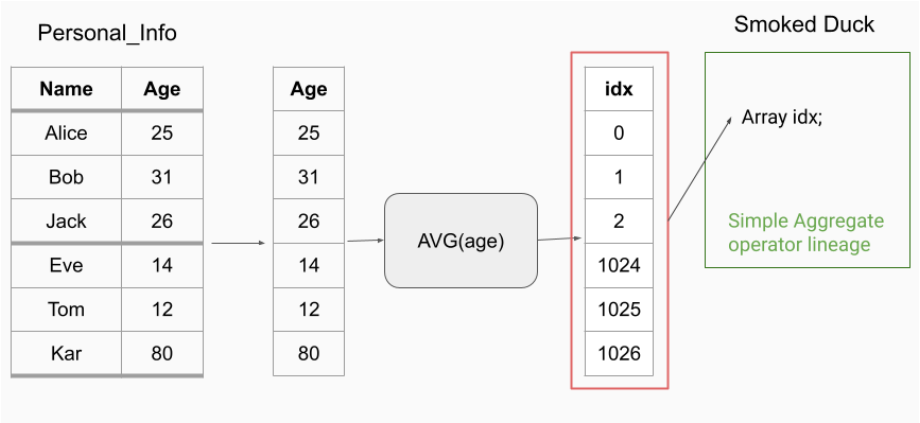

Figure 4.4: Simple Aggregate Lineage capture

4.2 Lineage Querying

We discussed how Smoked Duck pins the DuckDB artifacts into memory and captures lineage. Now that the operator lineage has been captured, the lineage must be amenable to querying. One of the naive ways of ensuring that the captured lineage is queryable by the native execution engine, DuckDB, is to register the lineage data as a relational table in DuckDB during lineage capture.

Figure 4.5 describes the process of registering the operator lineage data structure of Smoked Duck into memory to a relational table in DuckDB. On the left of figure 4.5, we see a vector of lineage data that makes up the operator lineage in Smoked Duck. Each Lineage data in the operator lineage is iterated. In every

iteration, an output chunk is prepared from the lineage data and appended to the Lineage table in DuckDB during capture. The selection vector in the Lineage data make up the input ID column in the output chunk. A monotonically increasing sequence of numbers make up the output ID column in the output chunk. This output chunk is appended to the lineage table in DuckDB.

Similarly, the second output chunk is prepared using the lineage data. We see that the input ID column has the values 1024 and 1025. The selection vector in the Lineage data consists of values 0 and 1 from Figure 4.2. The values 0 and 1 are local IDs and must be converted to global IDs using chunks offsets. Hence we see the values 1024 and 1025 in the input ID column of output chunk 2. The output ID column in output chunk 2 is a monotonically increasing sequence of numbers. The output chunk two is appended to the lineage table in DuckDB. Thus the operator lineage captured is formatted as a relational table in DuckDB during capture.

4.3 Formatting lineage as a DuckDB table

Registering the operator lineage as a relational table is accompanied by registering table artifacts in DuckDB like the column types, column names, and others that prove to be quite expensive. Additionally, Figure 4.7 refers to the operation of appending partially filled chunks to a chunk collection in DuckDB. Since DuckDB performs chunk merging, as explained in Figure 3.2, registering and appending operator lineage data to relational tables become expensive. Thus lineage capture causes significant overhead, which affects the base query execution time. In Chapter 6 : Performance, section 1 describes the analysis of lineage capture time with and without formatting lineage as a relational table. But we need to minimize the base query execution time. Therefore Smoked Duck proposes to stop registering lineage data as relational tables during capture.

Now suppose that we have a lineage query Q: Select * from operator_lineage that queries the operator lineage of a given base query, as shown in Figure 4.8. Since the operator lineage is not formatted as a relational table, DuckDB's native scan operator cannot scan the data structure operator lineage pinned into memory by Smoked Duck. The first SQL interface that this work builds helps bridge this gap. The native execution engine provides a custom scan operator that scans the operator lineage data structure and materializes it into a relational table depicted in Figure 4.9. Chapter 5 covers the complete description of the scan operator.

4.4 End-to-End lineage querying

Presently we have shifted costs of materializing lineage data as a relational table from capture to querying. Additionally, in section 1.2, we describe two aspects of lineage querying, one is querying the operator lineage, and the other is querying the end-to-end lineage. We established that querying end-to-end

lineage involves joining operator lineage tables. Generally, querying end-end lineage involves querying specific output IDs. Hence, performing hash joins that involve full scans for joining specific lineage output IDs adds additional costs to an already expensive lineage querying. We have established two costs that need to be cut down to make the lineage querying faster. Smoked Duck proposes solutions to overcome the two mentioned costs.

Smoked Duck addresses the cost of formatting operator lineage as a relational table. Figure 4.10 describes how the join operation is performed over operator lineage data structures rather than the relational tables. We have a lineage query with input base query and output id oid and assume the base query plan as shown in Figure 4.10. Now the oid must be joined with the order by operator lineage, and the native execution engine must join the values matching the join condition with the filter operator lineage. We do not have the materialized input ID and the output ID columns to join the operator lineage data structure. Smoked Duck proposes maintaining a pointer between operator lineage data structures. With the pointer, DuckDB's execution engine can search the matched join values of the parent operator lineage in the child operator lineage. Chapter 5, section 5.1 explains how DuckDB's execution engine passes the pointer between operators.

Smoked Duck addresses the cost of full scans during hash joins performed for finding the lineage of a given output ID and a base query. Figure 4.11 depicts the problem of iterating through all the pinned artifacts to search for the matching values in the operator lineage, making the join expensive. Smoked Duck proposes indexes to avoid iterating through all the artifacts present in the operator lineage depicted as the SOLUTION in Figure 4.11. Chapter 5, section 5.2 describes in detail what the indexes are for each operator as defined by Smoked Duck and how the indexes are added to the DuckDB.

Additionally, the native execution engine still cannot serve the lineage queries. Figure 4.12 describes the need for a custom lineage query plan providing evidence why the native query planner cannot build the right plan to serve lineage queries. The lineage_query takes the input oid that needs to be loaded into a chunk scan operator. Additionally, suppose the base query has a binary operator like the hash join, which has a probe and build sides. In that case, the planner of the native execution engine must load the lineage of the build side of join into a chunk scan and create a separate pipeline. The native execution engine's query planner cannot accommodate the chunk scans mentioned in the lineage query plan. Thus we need a custom plan for lineage querying. Chapter 5 describes how the custom plan is integrated into DuckDB.

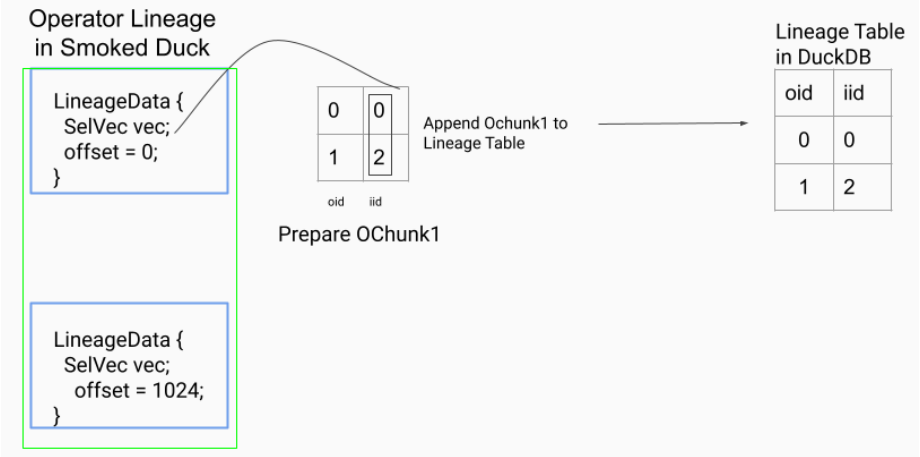

Figure 4.5: Appending operator lineage to a relational table in DuckDB

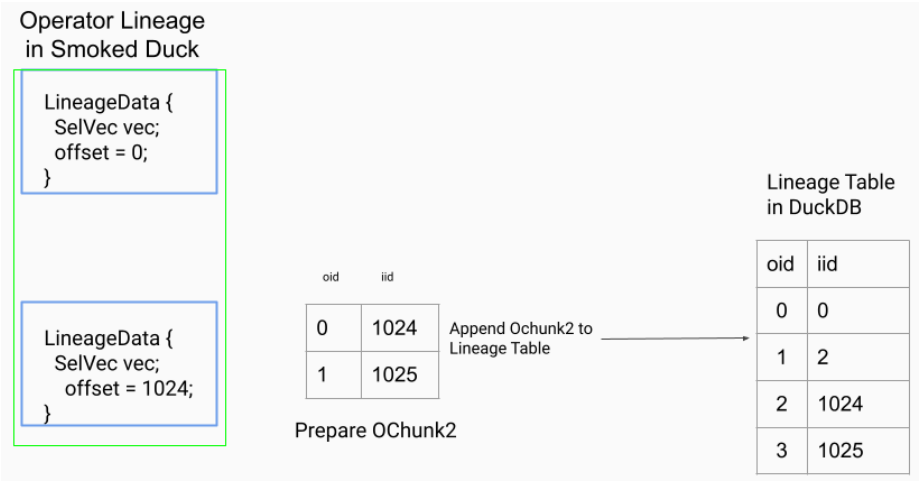

Figure 4.6: Appending operator lineage to a relational table in DuckDB

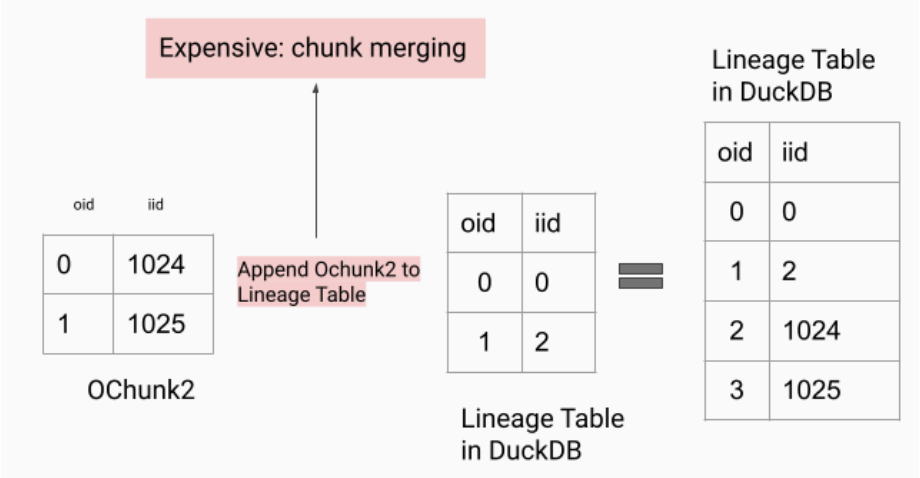

Figure 4.7: Appending partially filled output chunk to a table in DuckDB

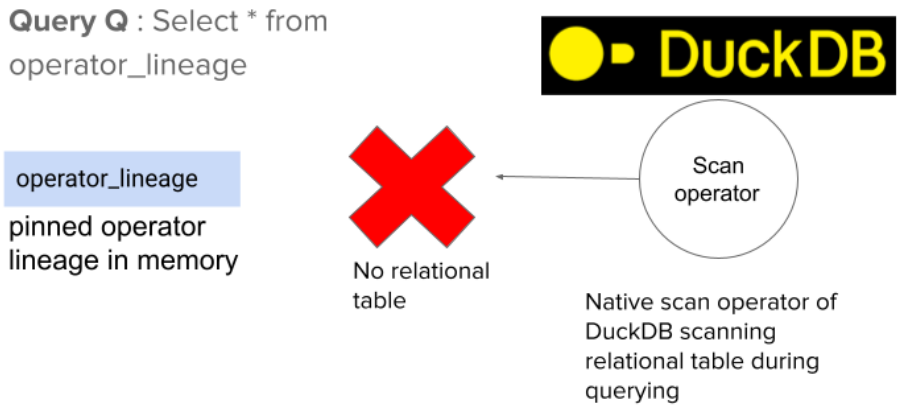

Figure 4.8: No native relational table for operator lineage

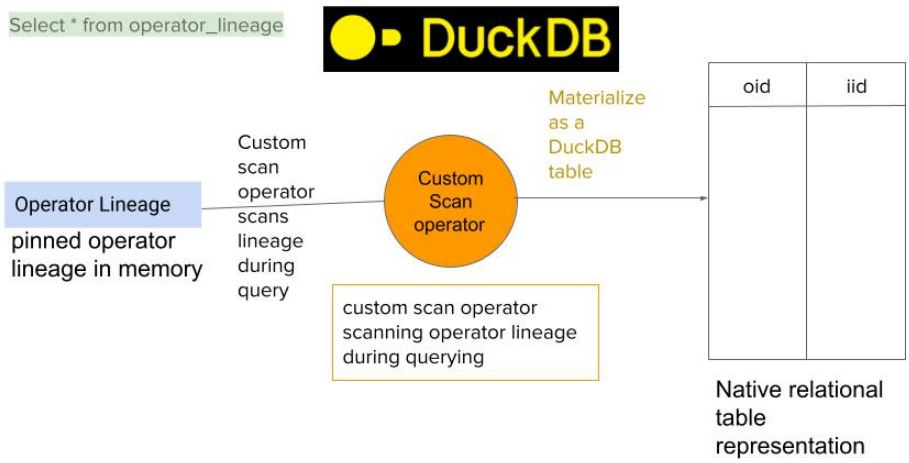

Figure 4.9: Custom Scan operator introduced in DuckDB

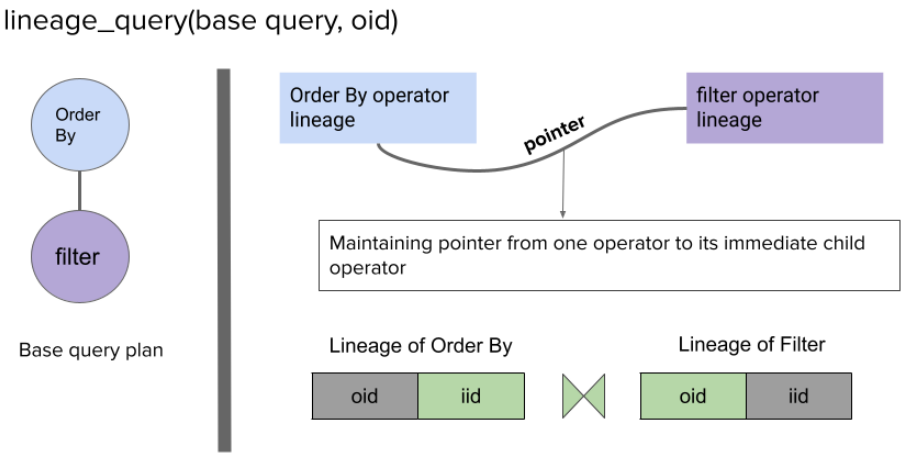

Figure 4.10: Maintaining child pointers between operator lineages

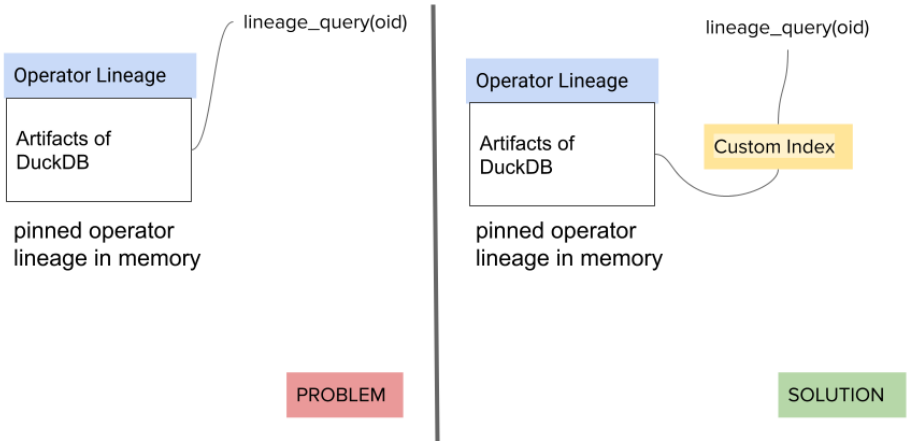

Figure 4.11: Motivation for custom Index

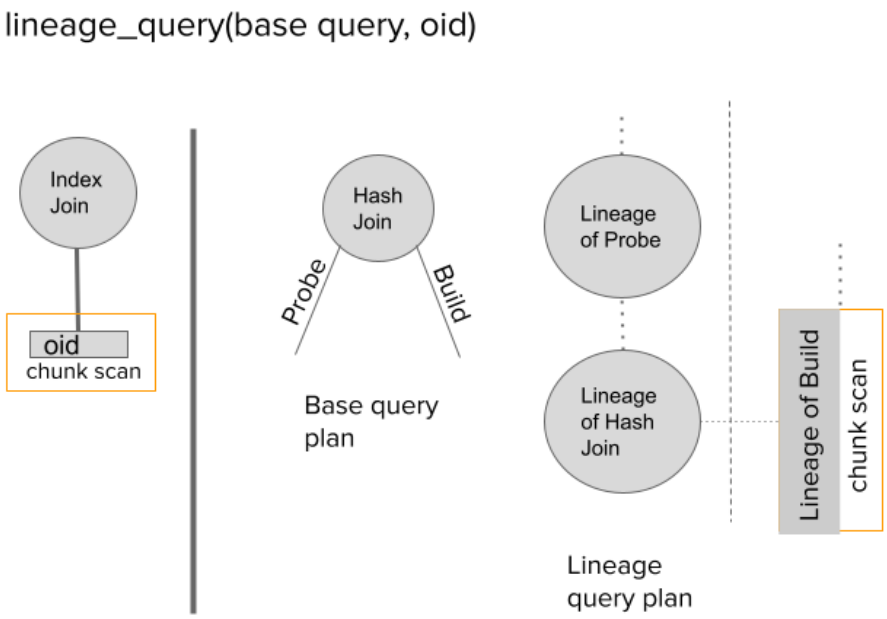

Figure 4.12: Need for a custom lineage query plan

Chapter 5

Architecture

The architecture section describes the architecture of additions to DuckDB's execution engine to support fast lineage querying. This work changes DuckDB's present execution engine to support fast lineage querying. The first section describes changes to query operator lineage, and the second section to query end-to-end lineage.

5.1 Querying operator lineage

To enable DuckDB's execution engine to query operator lineage, we introduce the custom scan operator that scans the underlying operator lineage data structure pinned into memory by Smoked Duck during lineage capture.

5.1.1 Motivation for the custom scan operator

We revisit the motivation for building the custom scan operator. The SQL execution of Smoked Duck entails building a custom scan operator that helps in querying lineage data captured in DuckDB during base query execution. The motivation behind using a custom scan operator arises because lineage data is not registered as a DuckDB table. Registering data as a table in any database is associated with collecting table artifacts, aka metadata about the table like the column types, column names and others. Additionally the cost of registering data specifically in DuckDB is accompanied by merging partially filled lineage chunks to chunks filled without empty slots between values into a chunk collection represented as a DuckDB table. DuckDB ensures that every chunk in the chunk collection is filled in order to promise an $O(1)$ lookup in GetValue() and SetValue().

Thus to overcome the cost of formatting the operator lineage as a DuckDB table, the operator lineage is not registered as a table. In the figure 5.1, we see two plans that depict the motivation behind using a lineage scan operator. The first plan uses a table scan operator to scan the underlying lineage data

structure. But since the scan operator is built to scan relational tables, it is unsuccessful in scanning the operator lineage. The second plan depicts a custom lineage scan operator to scan the operator lineage data structure.

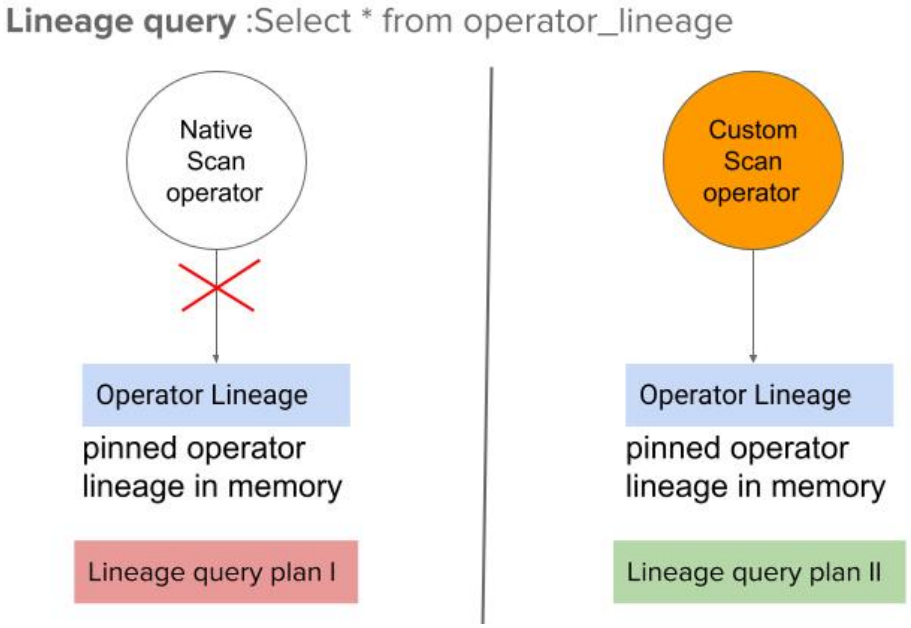

Figure 5.1: Plan with lineage scan

5.1.2 Understanding DuckDB's execution logic

The following section discusses the implementation details of the custom scan operator. To insert the logic of creating a custom scan operator into the system, we must consider how the system creates and executes the SQL plan. The system here is DuckDB and has the following sequence of operations that it performs before executing the physical operator. DuckDB uses a Postgres SQL parser to parse the SQL queries and builds a parse tree consisting of statements and expressions. The logical planner consists of the binder and the plan generator. The binder binds expressions to their column names and types, and the plan generator constructs the logical plan from the built parse tree. The planner converts the logical plan to a physical plan, and the optimizer optimizes the physical plan. The DuckDB execution engine then executes the physical plan.

5.1.3 Options of adding the custom scan operator logic

DuckDB has existing scan operators like TABLE_SCAN, CHUNK_SCAN, EMPTY_SCAN, and RECURSIVE_CTE_SCAN. The native scan operators have

their specific use cases and are invoked during the application of the use cases. The scans have their respective logical and physical operators. The logical plan decides where the logical scan operators must be placed, and these logical scans are converted into physical scans. The custom LINEAGE_SCAN draws inspiration from the TABLE_SCAN operator. LINEAGE_SCAN needs a custom logical operator and a physical operator at the onset. With more investigation, LINEAGE_SCAN is yet another TABLE_SCAN that scans Lineage Tables. One clever ruse is to check if the logical table scan is associated with a lineage table using the table name and creating a physical lineage scan instead of a physical table scan. The advantage of this approach is twofold: one is we do not meddle with DuckDBs binder for creating the logical lineage scan operator, and the other is writing as minimal code as possible to achieve lineage scan.

5.1.4 Design of custom lineage scan

The custom lineage scan operator must scan the operator lineage data structure pinned into memory during lineage capture. The custom lineage scan physical operator extends the generic physical operator interface and implements the GetChunkInternal method where the logic of custom scan operator is embedded. Whether the data structure must be sent as it is, should they be put into a chunk and adhere to the DuckDB's contract between operators to communicate as chunks? Figure 5.2 represents the two plausible design choices. If we choose to break the contract and start communicating with other operators with the operator lineage, each existing operator must be instrumented to communicate with operator lineage. This design choice has two shortcomings: 1) leads to changing all operators in DuckDB and amounts to many changes 2) Moving away from vectorized execution and thus defeating the performance optimization of query executor. Thus this design is not a viable choice of implementation. Hence we choose to transform the lineage data into chunks, and the lineage scan sends these chunks to the parent operator.

The following design choice is to copy the lineage data into the chunk or have pointers to these data in the chunk. Figure 5.3 shows both the mentioned scenarios. On the left-hand side, the light red slab represents the lineage data, and the dark red represents the chunk depicting copying the lineage data into the chunk. The diagram in green represents copying the pointers of lineage data into the chunks rather than the data itself. The consequence of copying lineage data is duplicating data, leading to a huge memory footprint. For rough estimation copying four bytes of integer data, every time lineage is queried, the memory footprint escalates linearly with the queried lineage data, which could be a few gigabytes.

Instead of copying operator lineage into chunks and moving operator lineage data around every time it is queried, we use pointers to operator lineage. Pointers are 8-byte addresses (considering a 64-bit machine) to operator lineage which are selection vectors in the case of Smoked Duck. If the pointers to the data are copied and moved around, then the cost of copying an 8-byte address is cheaper than copying 1024 4-byte integers. Additionally, the memory foot-

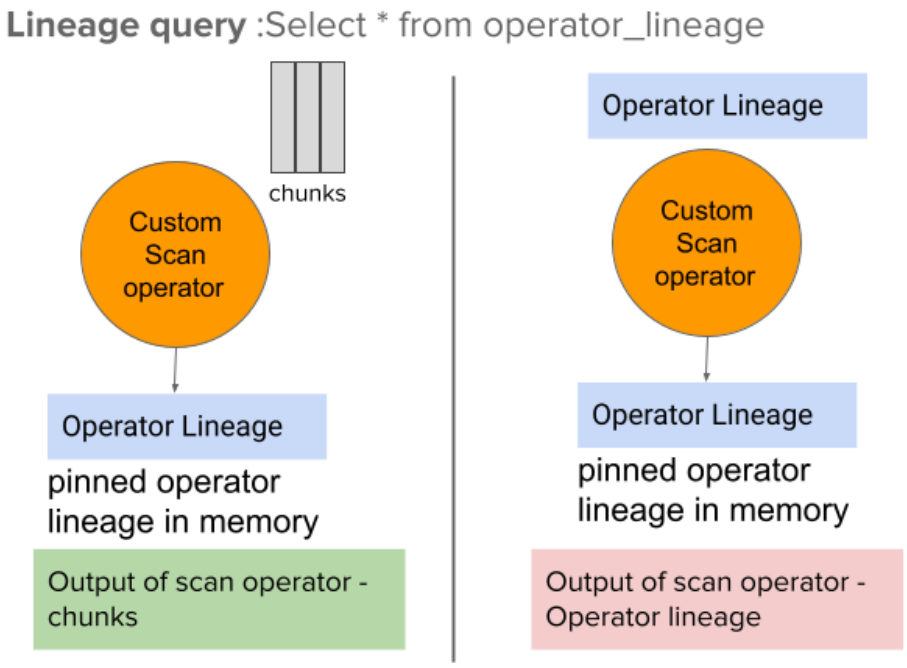

Figure 5.2: output of the scan operator

print of an 8-byte address is lesser than 1024*4-byte integers. This use-case is the worst-case time complexity time calculation, but there is an edge case where a selection vector can hold just a 4-byte integer in which scenario, copying this 4-byte integer into a chunk becomes cheaper than copying an 8-byte address pointing to the selection vector holding a 4-byte integer. On average, we assume this is a rare occurrence and implement copying pointers to operator lineage into chunks.

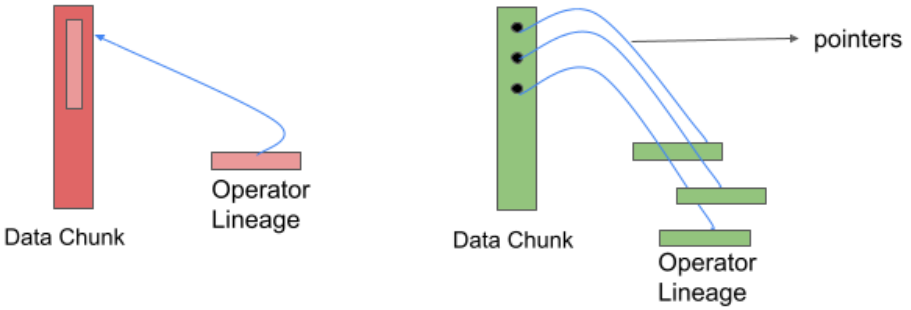

Figure 5.3: Copying data vs pointers into chunks

One of the plausible options while copying the pointers to selection vectors in operator lineage is to defragment the selection vectors, i.e., if we have four selection vectors each of a fixed size of 1024 but have only 256 values filled. We have two options: either traverse the plan four times each with a pointer to the selection vector or once by merging all the four vectors into one vector of 1024 values and make the pointer point to the one selection vector. There are two parrying costs here: 1. The cost of merging all the four partially filled selection vectors entails iterating every selection vector and copying the values into another until we have fixed-size selection vectors filled with 1024 values. 2. The number of times Iterating the query plan as the number of partially filled chunks. Both costs are the same since the cost of the first operation is proportional to the number of chunks and the second operation is also proportional to the number of chunks. Hence the worst-case time complexity is the same for both choices—O(number of partially filled chunks). Thus the lineage scan operator copies pointers to underlying lineage data and returns the pointer chunks to the calling parent operator (pull-based engine).

5.1.5 Binder exception

The binder in DuckDB binds all the expressions to their respective tables or views. During binding, the binder searches for the table name in the TableCatalogueEntry, and if it does not find the table name, it throws a "Table not found" exception. To overcome this exception and establish a convenient way to access the operator lineage data of a given base query, empty lineage tables are maintained with a pointer to the respective operator lineage data. When the client queries for lineage data of a specific operator of a base query, the lineage scan operator accesses the operator lineage data by getting the table from the TableCatalogueEntry and accessing the pointer to operator lineage from the table info.

5.1.6 Disabling optimizer

We introduce the custom scan operator because there are no relational tables for DuckDB's execution engine to query. Suppose there is a lineage query to fetch operator lineage. Since there are no relational tables, the cardinality of such a table is zero. The statistics optimizer of DuckDB adds an EmptyResult operator to the plan that returns an empty chunk instead of the scan operator as an optimization strategy. We alter the metadata table_info of lineage tables to make sure the cardinality of the lineage_table is more than zero so that the EmptyResult operator is not added to the plan. The other option is to disable the optimizer since the lineage tables are not registered anyway. Thus, the optimizer is of no use, and for this workflow, we disable the optimizer.

5.2 Querying end-to-end lineage

To enable DuckDB's execution engine to answer end-to-end lineage queries, we adopt changes suggested by DuckDB into Smoked Duck. The workflow for calculating end-to-end lineage is as follows: client interfaces with an API, the API generates a custom lineage query plan using the base query plan, and the custom plan is run on DuckDB's execution engine. Following a similar order, this section begins by defining a client interface for querying end-to-end lineage, proceeds with how a custom plan is determined, and lineage indexes are defined and integrated into DuckDB. Finally, it describes the changes made to the index join operator to perform the index join over operator lineages.

5.2.1 Client interface

The client interface for calculating end-to-end lineage warrants a PRAGMA function in DuckDB. Since the DuckDB's planner is unable to generate the lineage query plan, we need to skip the planner of DuckDB. We use PRAGMA functions in DuckDB to skip the internals of DuckDB's database engine. Section 3.5 defines what PRAGMAs in DuckDB are capable of and Section 4.2 describes the need to skip the planning phase of DuckDB's execution engine that consists of generating the logical plan, optimizing the logical plan and finally converting the logical plan to a physical plan.

The custom scan operator already allows the user to query operator lineage-specific to an operator of a base query. However, an end-to-end lineage that joins all the lineage tables captured as part of a base query is commonly sought in specific applications like interactive visualizations. Hence the question arises as to what the interface for the client should be for requesting the backward lineage of a given base query? We define the interface lineage_query that takes in a query string and an output ID for which the end-user asks the lineage.

5.2.2 Custom lineage query plan

In the previous section, we introduced the client interface to query lineage by defining a PRAGMA function lineage_query. The implementation of the PRAMA follows by fetching the base query plan from the query string function parameter. Smoked Duck stores a mapping of the query string to the base query plan. We use the mapping to fetch the base query plan and use this base query plan to generate a custom lineage query plan.

We discuss how we transform the base query plan to a lineage query plan. The lineage query plan is a plan with index joins. The output ID from the lineage_query function parameter is joined with the lineage of the root operator in base query plan. The matched values from this join is then joined with the child operator. Thus we need to generate a plan with joins that join the lineage of the operators in an inverted manner to the base query plan as shown in Figure 5.6 A.

We use the DuckDB constructs to build the custom plan. The output ID function parameter is put into a chunk_scan operator of DuckDB. Chunk Scan is a physical operator in DuckDB that stores data in chunks conforming to DuckDB's data model. We then use the index join operator of DuckDB, which takes in a chunk_scan as the input and indexes into the lineage of the root operator. The matched values of the index join operator are then fed to another index join operator that indexes into the lineage of the next immediate child operator in the base query plan, as depicted in Figure 5.6 A. For binary operators like hash join, which has two child operators in the base query plan, the corresponding lineage query has two separate pipelines, one for the probe side and the other for the build side. The build side lineage is loaded into a chunk_scan, and the rest of the lineage query plan pipeline follows the lineage of operators in the base query plan.

Additionally, the lineage query plans in Figure 5.6 A, B, C is suggested by Smoked Duck describing the three scenarios of stitching the data back together:

1) The fastest approach (Fig 5.6 Query A) would be to Union data from each pipeline. However, this loses the information relevant to deriving Provenance Polynomials [12] and instead produces Lineage [14]. 2) One way to capture provenance polynomials (Fig. 6 Query B) is to add an aggregation to the end of each pipeline containing either a Group By or an Aggregation without Group By to coalesce all ids that are added within the semiring into a single list. Now each pipeline will contain a single row per requested output id - we can join on this output id to yield provenance polynomials. In this representation, ids in an internal list are added together within a semiring, and separate columns indicate ids multiplied together. The downside of this approach is we must calculate an expensive array aggregation. 3) Approach similar to perm[6] to avoid performing the aggregation instead duplicate lineage for ids that join with an aggregation encoding another provenance polynomial representation. This returns separate rows corresponding to the same output id added together. Comparing the provenance models, we find that they encode the same polynomial, but this approach distributes the addition over the multiplication, i.e. (a + b + c) x d = (a x d) + (b x d) + (c x d)

A user can select which approach makes sense for their use case based on performance and provenance model requirements.

5.2.3 Creating custom lineage index in DuckDB

The previous section discussed a custom plan performing index joins that utilizes the indexes built on operator lineages. This section outlines how we define the lineage indexes using DuckDB's generic index interface. Before understanding where the custom indexes fit into the DuckDBs framework, we delve into how DuckDB offers index support for its current indexes.

Currently DuckDB supports a min-max index and an Adaptive Radix Tree (ART) index. The typical workflow for creating an index: The end-user explicitly defines the index through the SQL in DuckDB. DuckDB executes a PhysicalCreateIndex operator, registering the index in an IndexCatalogEntry with

the associated metadata like the indexed table name and the column names. We do not explicitly create an index using SQL for the custom index on the operator lineage because we use a custom plan for creating the index join operators and need a way to hook the index with the custom lineage plan's index join operators. The creation of index join operators requires the index as one of the arguments, and the custom index is created to make believe the execution engine of a DuckDB index. When the custom plan generator builds an index join operator, we create the lineage index during the custom lineage plan construction. Thus the custom index does not follow the typical DuckDB workflow.

5.2.4 Lineage Indexes based on Smoked Duck

Now that we define the lineage index interface we discuss what these indexes are. Section 4.2 and figure 4.11 motivate the need for indexing operator lineage data. Smoked Duck employs the Lineage Capture logic of operators based on whether the operator materializes data early or late. Smoked Duck introduces certain indexes built during lineage capture based on the materialization strategy and operator logic. The first diagram in Figure 5.4 depicts the child pointer, which directly indexes the child chunk for operators that support late materialization. The second diagram in Figure 5.4 illustrates indexing into the child operator lineage with global IDs for operators that support early materialization. A brief description of these indexes follows:

1) Operators with late materialization: Operators like filter, limit, and pipeline side of joins, i.e., the probe side of joins, maintain position lists as a part of lineage index, which would help index into the child chunk lineage directly without converting the chunk local value to the chunk global value. Since the child chunk lineage is accessed from the index, the execution engine must support passing the child lineage across operators to allow usage of child chunk lineage when necessary. The child pointers are embedded in the operator state of each IndexJoin operator that is built as a part of a custom lineage query plan. The parent IndexJoin operator state has access to the child IndexJoin operator state and thus accesses the pointer embedded in the child operator state.

Figure 5.5 represents the following lineage indexes defined in the Lineage Index construct built using the index interface of DuckDB.

2) Operators with early materialization: The build side of joins and aggregations uses an early materialization strategy and encodes global lineage ids. If we need to index the child chunk lineage, we need to convert the global ids to local chunk ids. Smoked Duck maintains an index array of cumulative chunk sizes enabling a binary search over the index array to find out the child chunk ids.

3) Joins: The build side of joins in DuckDB stores memory addresses in a hashtable, non-contiguous in the storage format, and performs the join probe on the hashtable of memory addresses. Smoked Duck proposes an index to store ranges of each block and an offset number for each previous block so the proper id can be calculated for a given block. This index is built incrementally as and when the hash-table requests for a new block of memory.

id = ((addr - block_start) / obj_size) + offset

4) Group By: Depending on the number of buckets, 2 different indexes are proposed. If the number is small, Smoked Duck does a post-processing step: creating a map from bucket identifier to a vector of ids of values that were grouped together in the bucket. If the number is huge, a zone map is built and a skipping scan is performed to fetch the requested ids.

5) Aggregate without group by : Map the output id to every id in the child operator for aggregations without a group. Hence the necessity to signify that the operation is an aggregation without a group; hence a flag is passed through the operators. The flag tells the operators that all the child ids must be returned. Thus the operator execution design must support the passing of flags.

When the custom planner sets the simple aggregate flag, the operators that only partition or re-order the set of lineage ids but do not remove any ids; do not change the lineage. Hence, the planner removes these operators from the custom lineage query plan. If the planner encounters a filter operator in the base query plan, the IndexJoin over the filter operator lineage is added to the custom lineage query plan. The planner sets the aggregation to False.

Since the lineage of operators like the build side of Hash Join, Group By, Aggregations, etc., consume all of the input chunks, the underlying lineage data loaded into chunks is different than that of other operators. To make use of the vectorized execution and improve the performance, the lineage data of the input chunks are loaded until the data size reaches 1024- the fixed vector size in DuckDB from section 3.1.

6) Early Materialization + Shuffle : For Order By, the lineage is captured in the following manner: a single vector with all the global ids without any chunk level lineage captured. Hence, no indexes are maintained as part of the group by lineage capture.

5.2.5 Changes to IndexJoin operator in DuckDB

All the previous sections describe how the custom hacks proposed by Smoked Duck are integrated into DuckDB to support fast lineage querying. We expect DuckDB's execution engine to incorporate the lineage indexes seamlessly and run the custom lineage query plan. Since the DuckDB type casts the IndexType in the IndexJoin operator, to use lineage indexes in the IndexJoin operator of DuckDB, a few changes need to be introduced in DuckDB's IndexJoin operator implementation. Index join in DuckDB is implemented as a nested loop join. The algorithm for the DuckDb's nested loop join is discussed in the next paragraph.

The index join algorithm joins data received from the child operator with the values from the index. In DuckDB's IndexJoin operator, GetRHSMathes() method searches every value from the child operator in the index. The method collects all the row-ids of values that satisfy the join condition. The Output() method receives the row-ids of the values satisfying the join condition from the GetRHSMatches() method and fetches all the columns that need to be materialized per the Select clause. Once all the columns required are materialized,

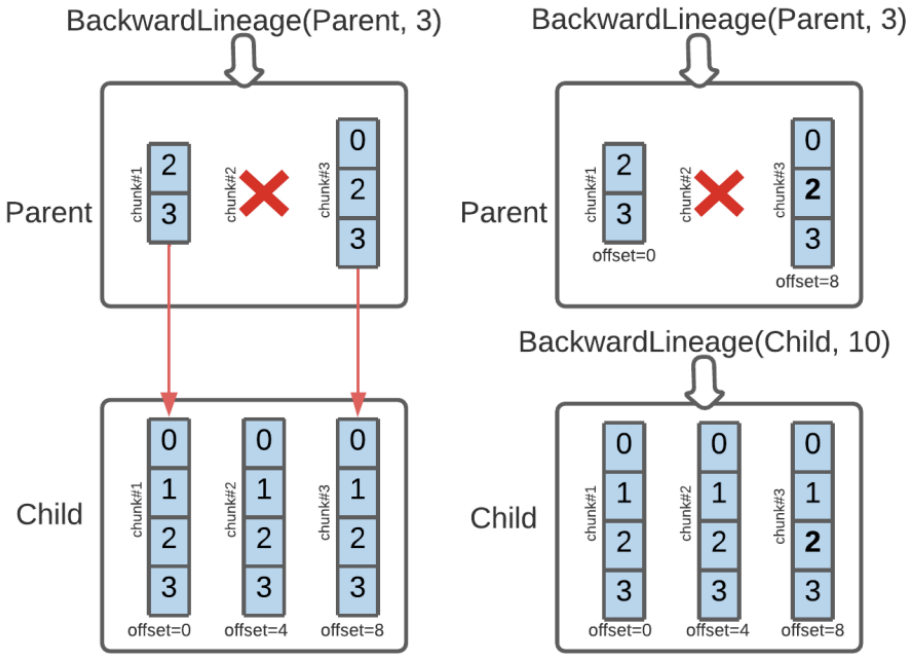

Figure 5.4: Child pointers

the Output() method fills the output chunk with the respective output values.

In implementing the GetRHSMatches() method, DuckDB casts the generic index to a specific index and proceeds with the rest of the execution logic. Additionally, lineage data doesn't require explicit materialization that the Output() method does. Because of the reasons mentioned, we create a separate workflow for the lineage indexes based on the IndexType attribute of the generic index interface. We skip the GetRHSMatches() method and embed join logic in the Output() method of the IndexJoin operator in DuckDB. Thus we fully integrate Smoked Duck's fast lineage querying into DuckDB.

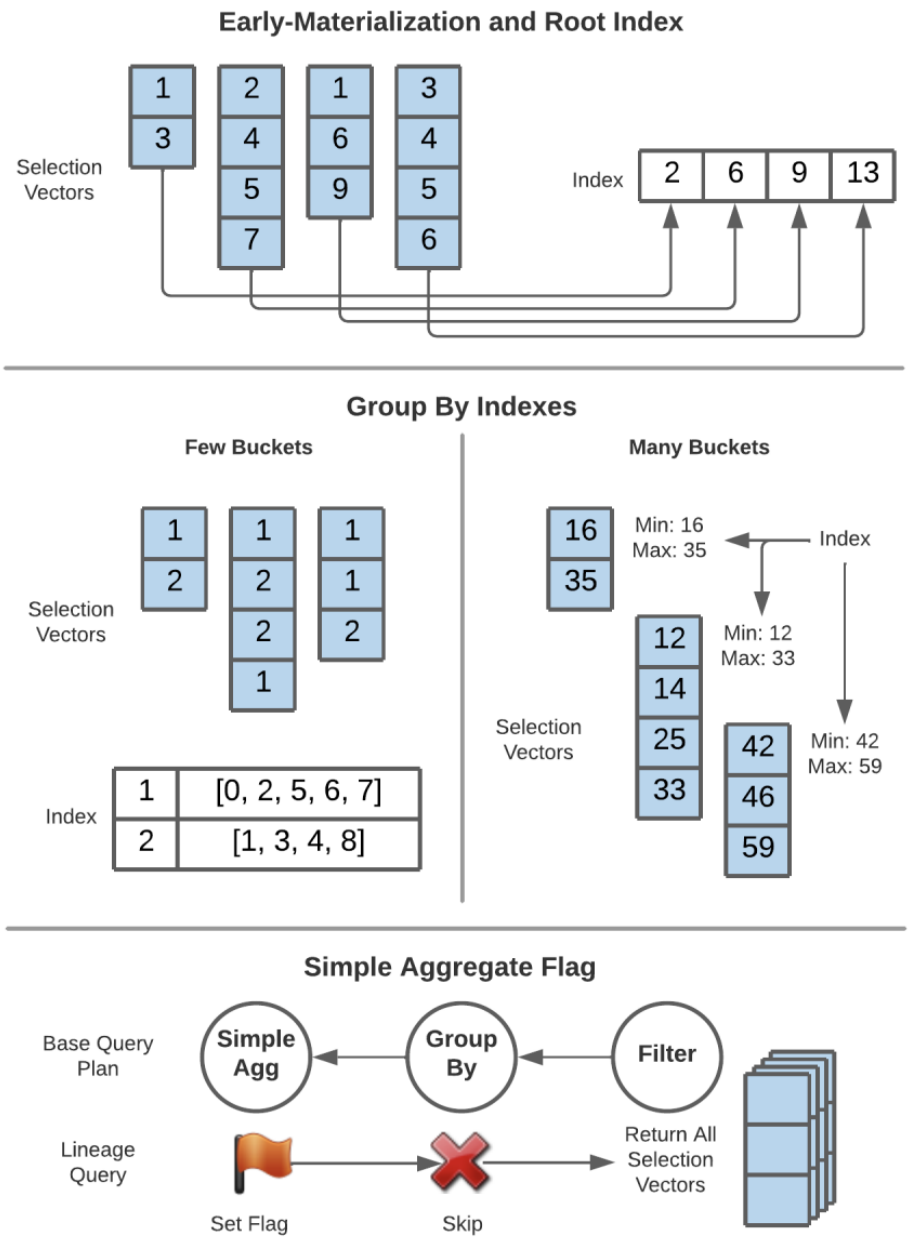

Figure 5.5: Indexes for operator lineage

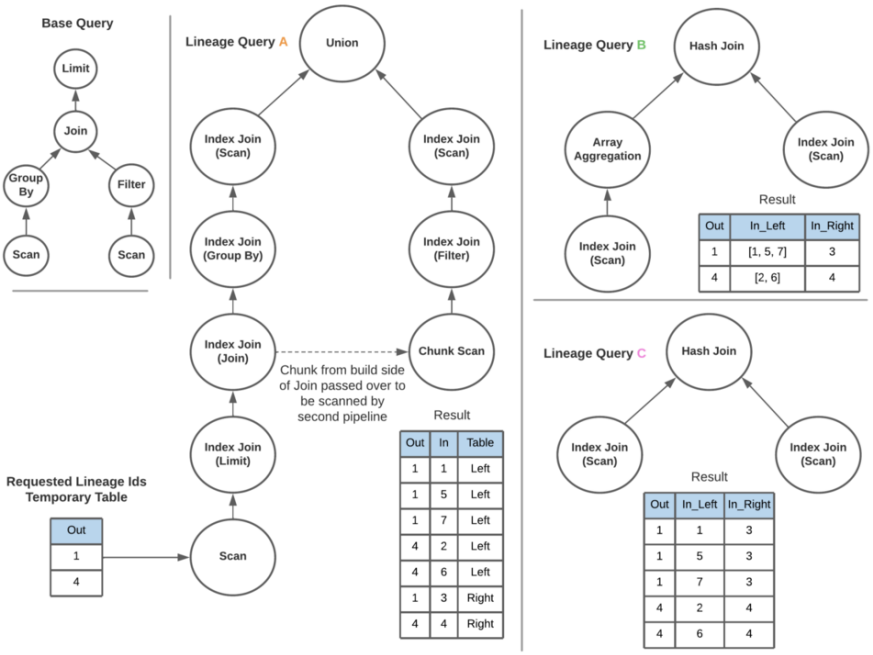

Figure 5.6: Custom lineage plan

Chapter 6

Experiments

In this chapter, we publish results of run times of lineage capture and lineage querying run on DuckDB's native execution engine. We use the TPC-H benchmark for evaluating the performance of lineage queries. The TPC-H is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications that have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The experiments are designed to validate the efficacy of using DuckDB's execution engine to answer end-to-end lineage queries.

6.1 Lineage capture

This section discusses the lineage capture time for all TPC-H queries. Figure 6.1 depicts a bar graph representing the time taken for lineage capture with and without formatting operator lineage data as a relational table in DuckDB. The X-axis of the graph represents the time in seconds and the Y-axis represents the TPC-H query number. The TPC-H queries are run on a scale factor of 1 which corresponds to 1 GB of data. We expect the lineage capture overhead of operator lineage being formatted as a relational table during capture represented by red bars to be more than that of operator lineage not formatted as a relational table during capture represented by green bars in Figure 6.1.

6.2 Lineage Querying

We record the performance of end-to-end lineage querying of the 0th output ID from the output of selected TPC-H queries. We compare the performance of end-to-end lineage querying time of HashJoin of lineage tables scanned with the custom scan operator and IndexJoin of lineage tables scanned with the index suggested by Smoked Duck. Figure 6.2 depicts a bar graph representing the time for querying with and without indexes over operator lineage in DuckDB.

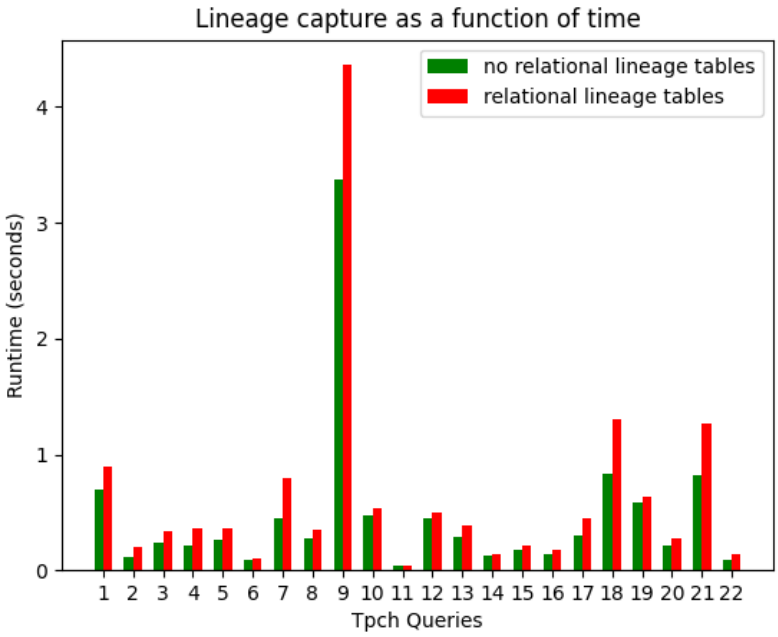

Figure 6.1: Comparison of lineage capture

The graph's X-axis represents the time in seconds, and the Y-axis represents the TPC-H query number. The TPC-H queries are run on a scale factor of 1, which corresponds to 1 GB of data. We witness that index join of operator lineage data structures pinned into memory by Smoked Duck outperforms hash join of operator lineage tables. Additionally, we publish the time taken by the index join of operator lineage data structures to calculate the end-to-end lineage of the 0th tuple in the output relation of TPC-H queries with a scale factor of 1. Figure 6.3 depicts a line graph referring to run times of calculating end-to-end lineage over TPC-H queries. A few runtime values of TPC-H queries show huge overheads like queries 11, 12, 13, 14, and 15. The joins on these queries must be investigated further to realize the performance overhead and do performance tuning over the codebase.

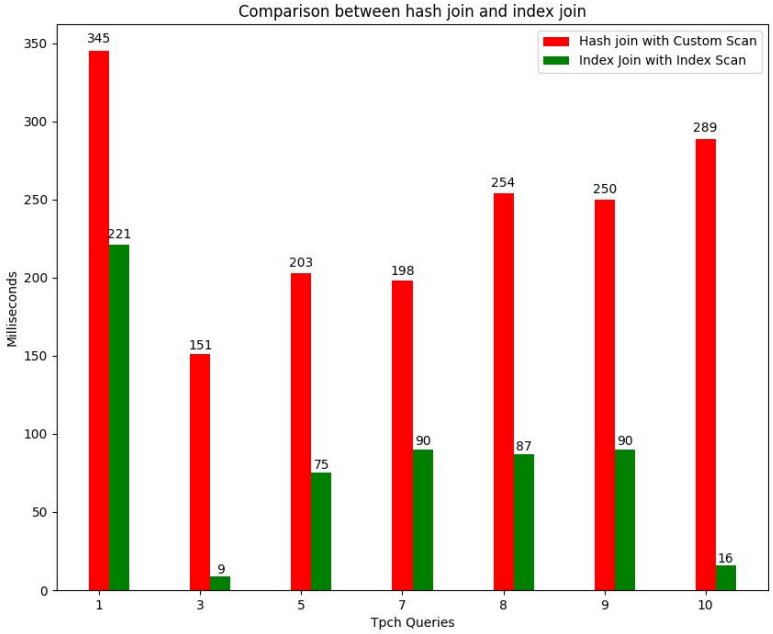

Figure 6.2: Comparison of end-to-end lineage querying

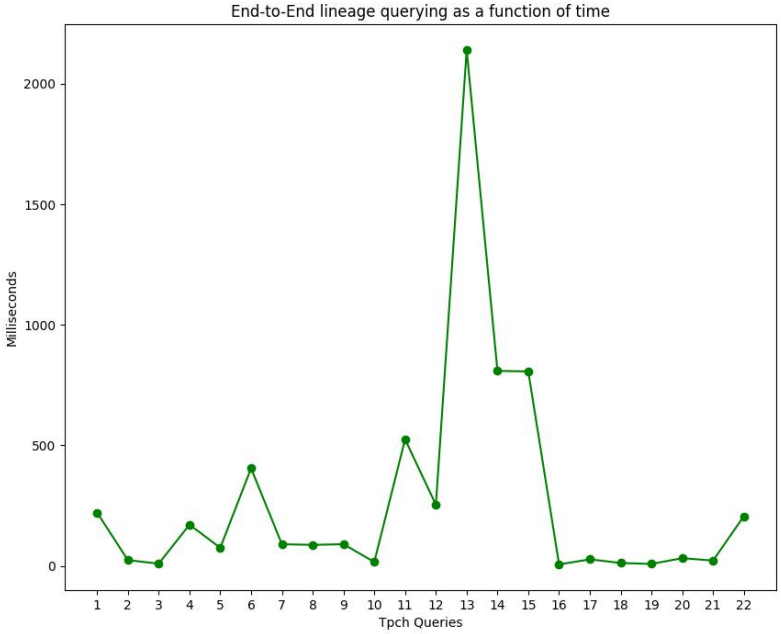

Figure 6.3: Time taken for end-to-end lineage querying using indexes

Chapter 7

Conclusion

In this thesis, we have explored integrating a lineage querying system into DuckDB's query execution engine. We have done this in a two-step process first by changing DuckDB's execution engine to support querying operator lineage, followed by querying end-to-end lineage. We successfully integrate the fast lineage querying mechanism suggested by Smoked Duck into DuckDB's execution engine enabling DuckDB users to query the underlying lineage.

The main contribution of this work is enabling DuckDB's fast vectorized engine to execute lineage queries. We have demonstrated practically the changes required to DuckDB to support fast lineage querying. Additionally, we have presented all plausible approaches to minimize lines of code and adhere to DuckDB's API contracts while integrating the changes into DuckDB. With the implementation of the proposed methods, we have shown that DuckDB's native execution engine achieves a good performance of lineage querying. Furthermore, this work enables DuckDB users to query both operator lineage and compute end-to-end lineage. Another insight of this work is practically proving the claims of Smoked Duck: late materialization vectorized execution engines can support fast lineage querying by accommodating custom indexes and changes to the internal execution engine.

7.1 Research Questions

The work in this thesis validates the claims made in section 1.4. We revisit the claims made in section 1,4 and also describe how the present work validates those claims

1) How to enable DuckDB's native execution engine to execute queries on the lineage data captured by Smoked Duck?

In Chapter 4, we explain how Smoked Duck captures lineage. We point out the costs associated with formatting captured lineage in a relational format of the native execution engine and how Smoked Duck overcomes this cost. This leads to the DuckDB's execution engine being unable to query the operator lin-

eage of the underlying data. We then motivate what the DuckDB's execution engine must implement to query the operator lineage. Additionally, Smoked Duck proposes specific changes like joining operator lineage data not materialized as a relational table and custom indexes on operator lineage to make the end-to-end computation much faster, as depicted in section 4.4. These suggested changes are integrated into the native execution engine of DuckDB by using DuckDB's internal APIs, as shown in Chapter 5. Thus we successfully incorporate the changes proposed by Smoked Duck and enable DuckDB's execution engine to query operator lineage and end-to-end lineage.

2) How to implement the SQL interface to capture lineage data into DuckDB's execution engine by maintaining optimal performance?

Chapter 5 discusses the architecture of the changes to DuckDB's native execution engine to serve fast end-to-end lineage queries. The implementation of the changes has always followed the performance overhead into consideration. For example, in the design of a custom scan operator, copying data pointers instead of lineage data around saves the cost of copying and improves the performance of querying operator lineage. Furthermore, by implementing the proposals of Smoked Duck like custom indexes and not materializing lineage as relational tables, this work enables lineage querying at interactive speeds. Additionally, implementing caching of lineage data to salvage the vectorized execution helps us achieve fast querying.

3) Minimize the changes introduced to DuckDB's native execution engine-use DuckDB's existing API extensions to make the integration of lineage querying into the native execution engine as seamless as possible.

This work prioritizes minimizing the changes made to the DuckDB's execution engine. For example, while adding the custom scan operator, we choose to introduce a new physical operator and do not change the logical planner and optimizer. We replace the TABLE_SCAN physical operator with the custom scan operator, minimizing the native execution engine changes as shown in section 5.1.3. As pointed out in section 5.2.5, the changes to the IndexJoin operator were imminent because of how DuckDB uses the index interface in IndexJoin else; the custom indexes defined using DuckDB's generic interface must have been seamlessly defined and integrated into the execution engine. Hence throughout this work, we adhere to DuckDB's API extensions and use them for implementing both the custom scan operator and the custom indexes.

Bibliography

[1] Market research report, fortune business research - https://www.fortunebusinessinsights.com/data-visualization-market-103259: :text=the *published 2018*.

[2] O. B. A. D. S. C. H. S. N. T. S. Agrawal, P. and J. Widom. An introduction to uldbs and the trio system.

[3] B. G. S. L. X. N. Bahareh Arab, Su Feng and Q. Zeng. Gprom - a swiss army knife for your provenance needs.

[4] S. R. M. D. J. Abadi and N. Hachem. Column-stores vs. row-stores: how different are they really?

[5] E. W. Fotis Psallidas. Provenance for interactive visualizations. *HILDA, 2018*.

[6] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. *In 2009 IEEE 25th International Conference on Data Engineering, pages 174–185. IEEE, 2009*.

[7] C. Haneen and E. Wu. Smoked duck. *Yet to be published*.

[8] B. L. Lee, S. and B. Glavic. Pug: a framework and practical implementation for why and why-not provenance.

[9] C. H. O. Benjelloun, A. D. Sarma and J. Widom. An introduction to uldbs and the trio system. *Technical report, Stanford InfoLab, 2006*.

[10] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *Proc. VLDB Endow.*, 11(6):719–732, feb 2018.

[11] M. Raasveldt and H. Mühleisen. Duckdb: An embeddable analytical database. 2019.

[12] G. K. T. J. Green and V. Tannen. Provenance semirings. *Proceedings of the ACM Symposium on Principles of Database Systems (PODS), pp. 675–686, 2007*.

[13] B. D. T. Müller and T. Grust. You say'what', i hear'where'and'why':(mis-) interpreting sql to derive fine-grained provenance. *arXiv preprint arXiv:1805.11517, 2018.*

[14] J. W. Y. Cui and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *TODS, 25(2):179–227, 2000.*