UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Adding Parallelism to Shredded Yannakakis

Max Van Gastel

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Stijn VANSUMMEREN

BEGELEIDER :
Mevrouw Liese BEKKERS

2024
2025

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Adding Parallelism to Shredded Yannakakis

Max Van Gastel

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Stijn VANSUMMEREN

BEGELEIDER :

Mevrouw Liese BEKKERS

Hasselt University

A thesis submitted for the degree of MSc Computer Science

Adding Parallelism to Shredded Yannakakis

Author:

Van Gastel Max

Promotor:

Prof. dr. Stijn Vansummeren

Mentor:

Mevrouw Liese Bekkers

Academic Year 2024-2025

Dankwoord

For starters, I would like to thank Prof. Dr. Vansummeren and Ms. Bekkers for their guidance during the realization of this thesis. The biweekly meetings were vital for assessing my contributions and pushing me further. During the academic period in which this thesis was written, I unfortunately incurred a serious physical injury, temporarily hampering my progress. I do not think I can come up with a better way to handle this situation than how they approached it, which helped ease the frustrations that come along with such an occurrence. Whenever I sent out emails regarding the thesis, they always rapidly responded with fitting and helpful replies, for which I am very grateful, and allowed me to not get discouraged during mental blocks.

I would also like to thank my parents for their support, especially for the extra care they took of me when I was recovering from the previously mentioned injury. This allowed me to keep my obligations in mind and finish the thesis in a timely manner. Lastly, I would like to thank my friends, both those on- and off-campus, for being a very necessary form of relief.

Abstract

Join operations are an important part of query engines, enabling the combination of data across multiple tables. As datasets grow, the efficiency with which joins are computed must increase alongside it. To this end, algorithms like Yannakakis' algorithm exist to speed up these calculations. Recent work has implemented a tweaked version of the algorithm to make it usable in practice. To further increase query evaluation speeds, computing calculations in parallel could prove beneficial. This thesis aims to implement parallelism in the implementation of Shredded Yannakakis in Apache DataFusion. To achieve this, an exchange operator was created that functions as a meta-operator, partitions data flowing through the query, and allows multiple workers to process the partitions concurrently. The operator was integrated into 2-phase NSA plans that accompany Shredded Yannakakis to parallelize the entire runtime. Care was taken to preserve the efficiency of the original algorithm while allowing it to run multi-threaded. Evaluation of the parallelized runtimes shows that both vertical and horizontal parallelism are enabled via the new operator, improving most of the measured runtimes. Picking the correct number of partitions to split the data into and preventing data skew across partitions proved to be an important factor in the efficiency of the parallelization. Following these results, the thesis further proves that careful integration of parallelism can bring performance increases to practical query engines.

Samenvatting

Doorheen deze masterproef werd er gewerkt aan het paralleliseren van een database engine implementatie. Bij het uitvoeren van queries is er nood aan de join operator om invoer van verschillende bronnen samen te brengen. Deze operator is een relatief "dure" operator, wat wil zeggen dat deze berekeningen vaak complexer en langer zijn dan de andere operatoren van de evaluatie. Aangezien deze operator vaak voorkomt in de praktijk is de snelheid waaraan hij correct uitvoer kan geven dus een belangrijke factor in de prestatie van een database engine. Een veelvoorkomend probleem van join operatoren is dat deze subresultaten kan opleveren die niet meedragen aan het uiteindelijke eindresultaat. Dit zorgt ervoor dat er tijdens het evalueren van een query onnodig werk verricht wordt, die de uitvoersnelheid zal verlagen.

Yannakakis' algoritme

Een oplossing voor dit probleem is het gebruik van Yannakakis' algoritme. Dit algoritme kan gebruikt worden om de joins van een subset van queries efficiënt te evalueren door de onnodige datapunten te filteren. Om dit uit te kunnen voeren, moet er eerst aan de hand van het GYO-algoritme nagegaan worden of de te evalueren query een acyclische query is, om dan verder te gaan naar het Yannakakis-algoritme. Yannakakis' algoritme wordt in twee delen opgesplitst en vindt plaats aan de hand van de uitvoer van het GYO-algoritme. Deze uitvoer noemt men een join-tree en is een boomstructuur waarin de relaties van de query in aanwezig zijn. De eerste stap noemt men de semi-join reductie, hier wordt er aan de hand van de semi-join operator en de join-tree gewerkt. Er zal aan de hand van de join-tree een volgorde van semi-join operaties opgezet worden die bottom-up en top-down werken. Deze operaties zullen ervoor zorgen dat alle onnodige tuples voor deze berekeningen niet meer beschouwd worden. De tweede stap van het algoritme bevat dan de effectieve joins die uitgevoerd horen te worden.

Door de grote hoeveelheid werk die nodig is voor de voorbewerking van de data tijdens Yannakakis' algoritme blijkt in de praktijk dat het algoritme zoals het origineel uitgewerkt werd toch niet zo bruikbaar is. Praktisch zijn er vaak niet genoeg redundante tuples tijdens join berekeningen om het extra werk dat verricht zou moeten worden goed te praten. Dit wil echter niet zeggen dat het algoritme onbruikbaar is. Recent werk [4] maakt gebruik van de algemene concepten van het algoritme om een join algoritme op te stellen dat wel praktisch bruikbaar is. De aanpak zelf noemt Shredded Yannakakis, het maakt gebruik van geneste datastructuren en query shredding om deze voor te stellen. Geneste data is hiërarchisch gestructureerd, dit komt omdat het hier mogelijk is om als attribuut een lijst van datapunten te gebruiken in plaats van een enkel datapunt. Om deze geneste data voor te stellen wordt de data nu opgesplitst in verschillende delen die apart als lijsten opgeslagen kunnen worden.

Een voorbeeld is te vinden in figuur 1. Aan de linkerkant is een tabel met geneste data aanwezig. Het bevat twee normale attributen x en y, en dan een genest attribuut waarin er nog een niveau dieper in genest is om $\{u, \{v\}\}$ te bekomen. De tabel bevat twee tuples maar om deze tuples te kunnen gebruiken als normale platte tuples moeten ze omgezet worden aan de hand van een unnest operator. Deze operator zal bijvoorbeeld de bovenste tuple omzetten naar twee verschillende platte tuples, namelijk $\{a_1, b_1, c_1, d_1\}$ en $\{a_1, b_1, c_2, d_2\}$. De figuur geeft nu dus ook weer hoe we deze twee geneste tuples zouden moeten voorstellen aan de hand van query

shredding. De platte attributen x en y worden normaal bewaard met hun eigen waardes. De geneste attributen worden bijgehouden aan de hand van hd, wat een pointer is naar de start van een gelinkte lijst en w, wat het aantal tuples na unnesten voorstelt. Ieder niveau van nesting heeft dan zijn eigen *store*, dit is een datastructuur die gebruikt wordt als de gelinkte lijst waar hd naar wijst en die zelf nog extra gelijkaardige data bevat indien er dieper geneste niveaus aanwezig zijn. Zo bevat in dit geval de store van $\{u, \{v\}\}$ opnieuw hd en w die verwijzen naar de store van $\{v\}$. Met deze datastructuur kan geneste data dus voorgesteld worden aan de hand van een aantal arrays.

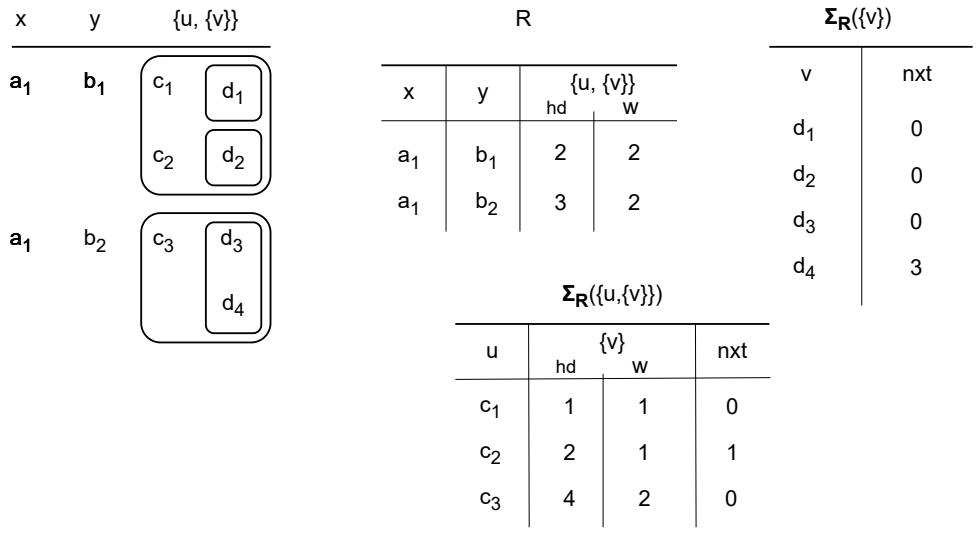


Figure 1: Voorbeeld van een shredded relation R

Query engines & Apache DataFusion

Doorheen de thesis werd er gebruik gemaakt van Apache DataFusion [1], dit is een uitbreidbare query engine met als doel om een sterke basis aan te bieden om zelf use-cases uit te bouwen. Het is een database waarin data opgeslagen wordt in kolomformaat. Dit wil zeggen dat de waarden van elke kolom fysiek bij elkaar opgeslagen worden, in tegenstelling tot het meer klassieke rij-gebaseerde formaat waar data per tuple opgeslagen wordt. Deze attribuutgebaseerde opslagmethode leent zich goed tot query shredding, waar data op een gelijkaardige manier wordt opgeslagen. Om deze reden werd er voor [4] ook een implementatie gemaakt in Apache DataFusion. Het is dus deze implementatie waarvoor parallelisme geïmplementeerd werd tijdens deze thesis.

Een algemeen concept in database engines zijn de query plans. Om SQL queries te kunnen evalueren moeten ze omgezet kunnen worden in uitvoerbare plannen die aangeven welke stappen er moeten genomen worden om de gewenste resultaten te verkrijgen. Indien we een voorbeeld database aannemen met een tabel over *Orders* en een tabel over *Products*, kan een query plan er uit zien als figuur 2. Hier zijn twee tablescans aanwezig voor de twee invoerrelaties, gevolgd door een join, een filter en een projectie. Dit is het niet-geoptimaliseerde logisch plan voor een query die data op wilt halen over orders waar het gekochte product een laptop is. Dit logisch plan kan dan verder geoptimaliseerd worden maar bevat geen specifieke operatoren, enkel het soort operator dat nodig is. Om het plan bruikbaar te maken zal de query engine de operatoren in het logisch plan om moeten zetten naar implementaties van operatoren die hij ter beschikking heeft. Hij zal bijvoorbeeld moeten kiezen welk type join het meest geschikt is voor de specifieke situatie, zoals bijvoorbeeld een hash join of en merge join. Dit doet hij aan de hand van metingen over de invoerrelaties en de algemene structuur van de query.

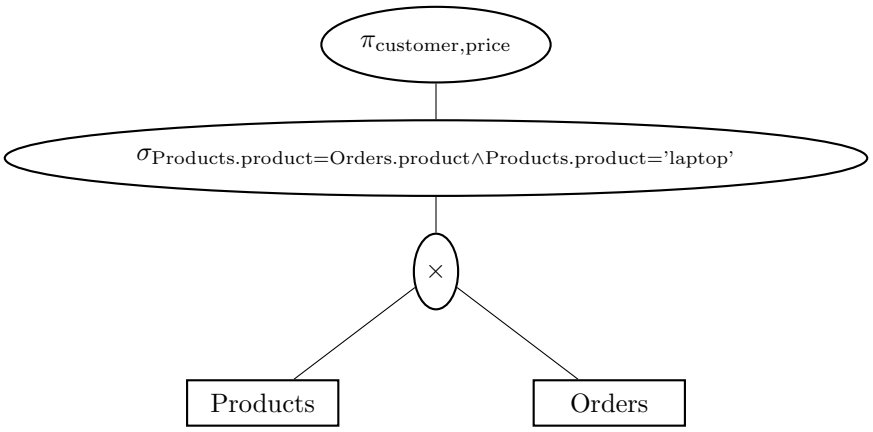


Figure 2: Logisch query plan

Volcano operator model & parallelisme

Om te kunnen werken met de DataFusion implementatie van Shredded Yannakakis moet er dan gekeken worden naar hoe database operatoren in de praktijk hun data verwerken. Een manier om dit te structureren is om het Volcano operator model paradigma te volgen. Hierbij worden er een aantal procedures voorgesteld om operatoren met elkaar te laten communiceren. Op deze manier kunnen ze steunen op een gestandardiseerde interface om met elkaar te communiceren en zo data ophalen uit operatoren. Operatoren in een query plan kunnen nu data opvragen van hun kind-operatoren van de boomstructuur om dan die data zelf te kunnen verwerken en verder door te geven aan hun voorgangers in de boom. Dit paradigma stelt ook een manier voor om aan de hand van deze interface parallelisme te implementeren in query plans. Het doet dit aan de hand van een meta-operator die in de boomstructuur geplaatst kan worden, deze operator noemt dan de exchange operator.

Om parallelisme toe te laten tijdens het evalueren van queries zal de exchange operator nieuwe werkers maken die data zullen opvragen van de operatoren onder de exchange operator. Om dit mogelijk te maken moet de data opgesplitst worden in verschillende partities. Elke werker krijgt dan een partitie toegewezen en zal onderliggende data vanuit deze partitie opvragen onafhankelijk van de andere aanwezige werkers. Een tweede taak van de exchange operator is om de data die binnenkomt door de werkers te herpartitioneren. Dit wil zeggen dat er aan de hand van een bepaald partitie schema, bijvoorbeeld een hash functie op een specifieke kolom, beslist wordt naar welke partitie iedere tuple gestuurd moet worden. Deze herpartitionering is essentieel wanneer operatoren later in het query plan data verwachten die verdeeld is op een bepaalde manier, zoals bijvoorbeeld bij een join operatie. Aangezien de implementatie van Shredded Yannakakis gebruik maakt van het Volcano operator model is het nu dus de bedoeling dat er een exchange operator wordt gemaakt die werkt met shredded data.

Shredded Yannakakis

In [4] wordt uitgelegd hoe query shredding gebruikt kan worden om joins te evalueren aan de hand van Yannakakis' algoritme. Door gebruik te maken van een multisemijoin $\rtimes$ en groupby γ operator kan het gedrag van een hash-join nagebootst worden. Na deze join operatie zullen onnodige tuples echter niet overblijven waardoor deze dus geen deel uit maken van latere operatoren en er dus rekenkracht bespaard kan worden, wat leidt tot een efficiëntere uitvoering. Om dit waar te maken moeten de query plannen die gebruik maken van deze operatoren echter een bepaalde structuur volgen. De plannen moeten zogenaamd twee-fasig zijn, dit houdt in dat alle joins eerst plaats vinden als afwisselingen van multisemijoins en groupby operatoren. Het resultaat van deze joins zal een geneste datastructuur zijn, die alvorens hij algemeen bruikbaar is, geontnest moet worden. De unnest operator μ die overeenkomt met de unnest fase bevindt

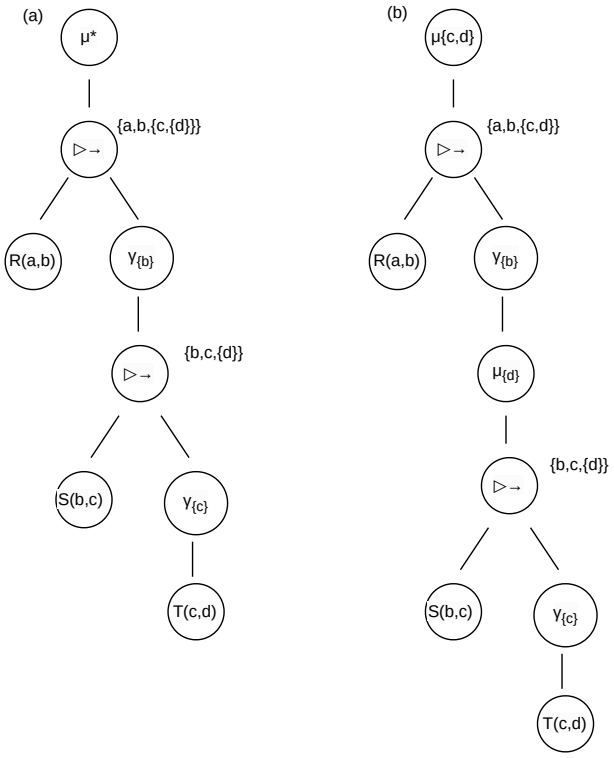


Figure 3: (a) een 2-fase NSA plan. (b) Een niet 2-fase NSA plan, met een unnest in de join phase.

zich bovenop de join fase. Het is belangrijk dat deze structuur wordt bewaard aangezien joinen na ontnesten wel mogelijks voor redundante tuples zou kunnen zorgen tijdens de berekeningen. Figuur 3 geeft een voorbeeld van een geldig 2-fase NSA plan en een ongeldig plan. Plan b is ongeldig in deze figuur omdat een unnest operator wordt gebruikt in de join phase, waardoor er redundante tuples zouden kunnen ontstaan.

Exchange operator voor Shredded Yannakakis

Om de exchange operator te implementeren voor Shredded Yannakakis met als doel de evaluatie ervan te paralleliseren, moet er dus een nieuwe operator gemaakt worden die in een 2-phase NSA plan geïntroduceerd kan worden. De uitvoer van de multisemijoin en groupby operatoren zijn verschillend dus hier zouden dan twee verschillende exchange operatoren voor moeten worden voorzien. Tijdens het uitwerken van hoe deze operator zou moeten functioneren werd het al snel duidelijk dat een van de twee voldoende zou zijn om de runtime te paralleliseren. Er werd gekozen om een exchange operator te maken die bovenop een multisemijoin zit. Deze operator moet nu dus nieuwe werkers opstarten die data uit de bijbehorende partities van de bijbehorende multisemijoin opvraagt en dan de inkomende data herpartitioneert. Dit correct partitioneren is nu van groot belang om de evaluatie van de query correct te laten verlopen. Aangezien de data nu in partities moet onderverdeeld worden zullen de verschillende operatoren de verschillende partities in parallel kunnen evalueren. Een probleem kan nu plaatsvinden indien de data komende van een invoerrelatie die richting een multisemijoin gaat, niet op dezelfde manier gepartitioneerd is als de data komende van de groupby operatie die ook richting dezelfde multisemijoin gaat. Wanneer deze verdelingen niet overeenstemmen, bestaat het risico dat tuples die samengevoegd moeten worden in de multisemijoin zich in verschillende partities

bevinden. Dit zou dan betekenen dat de werkers niet over de correcte set van relevante data beschikken om de multisemijoin correct te berekenen en dus voor een foute uitvoer zullen zorgen. Om dit op te lossen zullen we er dus altijd voor zorgen dat beide invoerrelaties van de multisemijoin op hetzelfde attribuut gepartitioneerd worden met dezelfde hashfunctie om zo ervoor te zorgen dat overeenkomende waarden die gejoined moeten worden altijd in dezelfde partitie zullen zitten.

Om de runtime te paralleliseren voegen we nu boven iedere multisemijoin een exchange operator toe die er voor zorgt dat de verschillende partities tegelijk geëvalueerd kunnen worden, en dat data naar de juiste partities wordt gestuurd. Belangrijk om op te merken is dat de exchange operator die net onder de unnest operator zou staan onnodig werk doet bij het berekenen van de hash waarden voor de tuples. Het heeft echter weinig belang dat er correct gepartioneerd wordt aan de hand van hashing vooraleer de data de unnest operator in gaat aangezien deze hier geen nood aan heeft om de uitvoer correct te houden. Dit zou leiden tot overbodig rekenwerk zonder prestatiewinst. Wat wel voor prestatiewinst kan zorgen met deze exchange operator zou een verandering zijn van partitie schema die voor een verdeelde werklading in de unnest operator leidt. We kunnen namelijk aan de hand van een normale exchange operator, er voor zorgen dat de unnest operator ook zijn verschillende partities in parallel kan evalueren. In het geval dat we de hoeveelheid werk dat verricht moet worden in deze partities kunnen balanceren, dan gaan we een bottleneck in een bepaalde partitie tegen, wat zorgt voor een versnelling van de unnest operator. In Figuur 4 zijn drie evenwaardige plannen zichtbaar die praktisch voor zouden kunnen komen in de implementatie. Plan a is de standaard seriële versie van het plan, waar er geen exchange operatoren in aanwezig zijn. Query plan b is een parallel plan, een exch() knoop duidt op een exchange operator die gemaakt is tijdens deze thesis, deze operator is gemaakt om de uitvoer van een multisemijoin op te kunnen vangen. Een rep() knoop duidt op de exchange operator zoals hij gemaakt is in Apache DataFusion, deze wordt ingeschakeld om de unnest operator te parelleliseren, aangezien deze standaard platte data als uitvoer geeft en om de invoerrelaties correct te partitioneren.

Door de exchange operator te implementeren voegen we nu horizontaal en verticaal parallelisme toe aan de implementatie. Horizontaal parallelisme verwijst naar de parallelle uitvoering van de verschillende partities binnenin dezelfde operator. Iedere werker verwerkt hierbij afzonderlijk zijn partitie, waardoor meerdere delen van de data tegelijk kunnen worden geëvalueerd. Bij verticaal parallelisme voeren meerdere operatoren uit het plan tegelijk hun berekeningen uit. Terwijl een bepaalde operator data produceert, kan een bovenliggende operator deze data onmiddellijk ontvangen en verwerken. Deze overlap aan werk draagt bij aan een verminderde uitvoeringstijd. Het moet echter wel gezegd worden dat de hoeveelheid verticaal parallelisme maar miniem is in deze implementatie. Door hoe de groupby operator werkt, moet hier alle data aanwezig zijn vooraleer deze uitvoer kan maken. Dit wil dus zeggen dat operatoren boven de groupby niet kunnen werken terwijl operatoren onder de groupby nog aan het werken zijn aangezien de groupby zelf geen data zal propageren.

Resultaten van parallelisme

Voor de evaluatie van de nieuwe operator werd er gebruik gemaakt van de STATS-CEB benchmark [5]. Deze bevat data en een reeks aan queries voor deze data die ontworpen zijn om uitvoeringstijd van join algoritmes te meten. Om deze evaluatie uit te voeren wordt er gebruik gemaakt van de uitvoeringstijd zelf, maar ook van de speedup en de efficiëntie over verschillende degrees of parallelism. De degree of parallelism verwijst naar het aantal werkers die gelijktijdig kunnen worden ingezet tijdens de uitvoering. Door de uitvoeringstijd te meten doorheen verschillende degrees of parallelism kan er onderzocht worden hoe goed het systeem schaalt, idealiter schaalt de uitvoeringstijd mooi mee met het aantal werkers die we toelaten. Speedup meet hoeveel sneller een query wordt uitgevoerd wanneer we de degree of parallelism verhogen, vergeleken met de seriële uitvoering, en de efficiëntie geeft aan hoe goed de beschikbare rekenkracht benut wordt. De degree of parallelism wordt doorheen de experimenten aangepast door het aantal partities aan te passen waarin de data wordt verdeeld. Aangezien de exchange

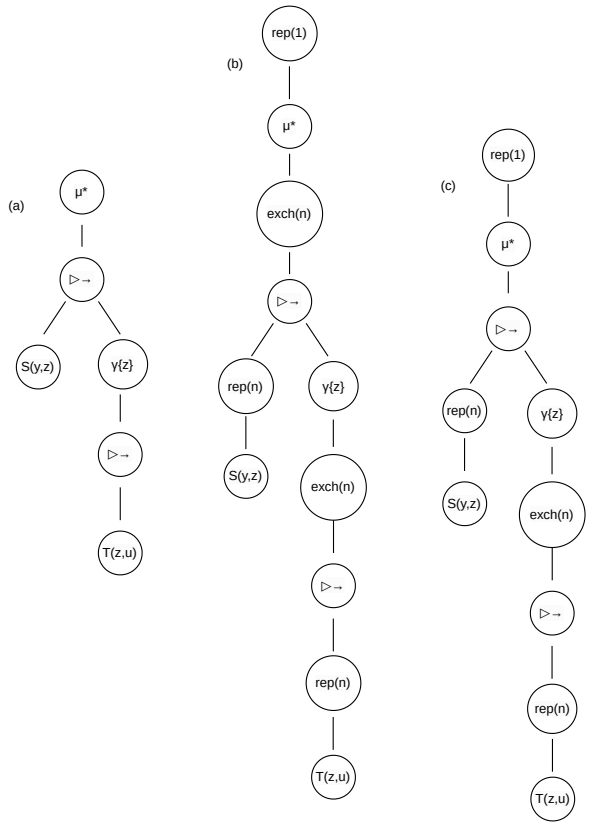


Figure 4: Verschillende equivalente plannen, serieel en parallel

operator werkers zal opstarten afhankelijk van zijn invoer partities past dit dus het aantal parallelle werkers aan, terwijl er wel toegelaten wordt dat DataFusion alle 16 CPU cores aanwezig op het toestel benut.

De metingen tonen dat er duidelijk in de grote meerderheid van queries een snelheidswinst aanwezig is. Deze winst schaalt echter wel niet super goed, waardoor de ideale speedup nooit bereikt wordt. De exacte winst in een query is sterk afhankelijk van een aantal factoren zoals de grootte van de uitvoer en de data skew die plaatsvindt na het hashen van de tuples in een exchange operator. Skew komt vooral sterk tevoorschijn in de unnest operator, waardoor het toevoegen van de extra exchange operator alvorens het unnesten zoals zichtbaar in b in figuur 4 zeker de uitvoeringstijd kan verlagen. Er is namelijk veel werk in de unnest operator indien er veel tuples in de finale uitvoer zitten, slecht verdeeld werk kan dan voor grote oneffenheden kan zorgen, waardoor er in deze operator een bottleneck gemaakt wordt. De snelheidswinst verhoogt over het algemeen wel gaande van één tot acht partities, bij zestien partities wordt er over het algemeen terug een vermindering van uitvoeringssnelheid gezien, waarbij de algemene tijd wel nog lager ligt dan de seriële uitvoeringstijd. Deze verlaging wordt toegekend aan het feit dat er dan te veel werkers opgestart worden en waardoor het wisselen tussen deze werkers te veel extra werk vereist.

Contents

Chapter 1

Introduction

For a database system to evaluate queries, join operations are essential for combining data from multiple relations. They are required in a large number of queries, making their performance an important factor in the overall efficiency of the query engine. Among the various approaches to optimize joins, Yannakakis' algorithm promises an efficient algorithm to process acyclic conjunctive queries by reducing intermediate results and thus avoiding redundant computations. Though problems arise when implementing this algorithm in practice, recent work [4] has shown that it is possible to gain practical use out of the algorithm via query shredding. This approach is called Shredded Yannakakis. By using the benefits that an in-memory column store database engine provides, the original algorithm can be adapted to perform effectively on real-world workloads.

Throughout this thesis, Apache DataFusion was used as an extensible query engine that is written in Rust. It is designed as a foundation for implementing query optimization techniques and allows for modification throughout most aspects of the engine. Due to its open-source nature, the source code of DataFusion also allowed for a practical example of how parallelism could be implemented, though the data structure created for Shredded Yannakakis brought its own challenges along with it. The implementation of Shredded Yannakakis, created alongside [4], does not take parallelism into account. As such, by extending this implementation with a structure that enables parallelism, improved performance can be obtained while still preserving its own benefits for acyclic query evaluation.

The goal of this thesis is to implement parallelism into the existing implementation of Shredded Yannakakis via a meta-operator called the exchange operator. This operator spawns new workers that evaluate parts of the query in parallel, enabling concurrent execution of independent parts of the query plan and thus improving overall execution time by utilizing multiple cores to evaluate the query concurrently. Extra care was taken throughout the implementation to keep the original code unchanged as much as possible, since the exchange operator theoretically functions as an independent operator. After achieving parallelism, a closer look will be taken at the performance of the parallel scheme and any bottlenecks that might occur, alongside a comparison with the original serial implementation.

This thesis is organized as follows: the first two chapters provide the necessary background information, one focusing on Yannakakis' algorithm and the other on query engines, with an emphasis on Apache DataFusion and its own architecture. The fourth chapter details the implementation of parallelism, discussing the challenges encountered and solutions devised. Finally, the last chapter presents an analysis of the runtime performance based on measurements obtained with the STATS-CEB [12] benchmark.

Chapter 2

The Yannakakis algorithm

This chapter contains the theoretical foundation for understanding the algorithm central to this thesis. It goes over the necessary syntax, like first-order logic and conjunctive queries, to then introduce algorithms that eventually flow into Yannakakis' algorithm.

2.1 Prerequisites for the algorithm

This section will cover the underlying concepts necessary to understand and explain Yannakakis' algorithm. An important resource for this section was the book [2], of which specific chapters were studied to learn the foundations.

2.1.1 First-order logic

With the subject of the thesis being optimal joins, we immediately turn to conjunctive queries. These queries are a subset of in **first-order logic**, a formal system with well-defined syntax and semantics that enables logical representation and inference. To refresh these concepts, chapter three from [2] and chapter eight from [20] were studied.

A database schema $\mathbf{S}$ features a finite set of relation names, with each relation having its number of attributes specified in $\mathbf{S}$. Taking a relation name R with attributes u and v, we can form an **atomic formula** such as $R(1, 2)$, which asserts that the tuple $(1, 2)$ belongs to the relation R. In this atomic formula, R is a **relation symbol**, and the values 1 and 2 are **constants**, which are also considered *terms* in first-order logic. Formulae or sentences created using *First-Order logic* are then used for evaluation on a database instance.

We may also use symbols like x and y in place of constants, writing the atomic formula $R(x, y)$. In this case, x and y are **variables**, which are also *terms*, meaning they represent unspecified values and do not assert anything concrete about the contents of R. However, if we write $R(x, x)$, this indicates that we are only interested in tuples where the values for both attributes u and v are equal.

First-order logic allows us to chain multiple atomic formulae together using operators like $\wedge, \vee, \neg, =, \exists, \forall$, creating non-atomic/complex **formulae**. An example of this would be taking the following atomic formula: $Brother(Richard, John)$, which states that Richard and John are brothers. We can extend this *atomic formula* by adding a second *atomic formula* to it, together with an `AND` symbol $\wedge$, creating $Brother(Richard, John) \wedge Brother(John, Tom)$. This formula can then be evaluated on a database, allowing us to query for specific relationships and evaluate their truth values based on the data. For instance, if the database contains the information that $Brother(Richard, John)$ and $Brother(John, Tom)$ are true, then the entire formula $Brother(Richard, John) \wedge Brother(John, Tom)$ would evaluate to true. Adding

inferencing would allow the database to infer the fact that $Brother(Richard, Tom)$ is also the case as long as the *Brother* relation is said to be transitive.

2.1.2 Conjunctive queries

Conjunctive queries are a subset of *First-Order Logic* queries used to express relational joins, an essential operation in relational databases. Since data in such databases is typically distributed across multiple tables, combining information often requires joining these tables. When a query needs data from multiple tables, the database engine performs a join operation to bundle data from different sources. This subsection relies on chapter twelve from [2] as its foundation and for its conventions.

The syntax of a conjunctive query follows as written in [2] chapter 12.

A conjunctive query over schema S is a *First-Order* query $\varphi(\bar{x})$ over S with φ being a formula of the form

$$\exists \bar{y}(R_1(\bar{u}_1) \wedge \cdots \wedge R_n(\bar{u}_n)) \tag{2.1}$$

In this formula, n ≥ 1, $R_i(\bar{u}_i)$ is a **relational atom**, and $\bar{u}_i$ is a tuple of constants and variables mentioned in $\bar{x}$ and $\bar{y}$ for every $i \in [n]$.

Example 2.1.1. An example of a conjunctive query follows: given a relational schema **S**: $Person[pid, pname, cid]$, $Profession[pid, prname]$, $City[cid, cname, country]$. With this schema in mind, a conjunctive query can be made to retrieve the list of computer scientists born in the city of Athens in Greece can be constructed.

$$\exists x \exists z(\text{Person}(x, y, z) \wedge \text{Profession}(x, \text{'computer scientist'}) \wedge \text{City}(z, \text{'Athens'}, \text{'Greece'})) \tag{2.2}$$

This conjunctive query allows us to combine the results of multiple table sources(Person, Profession & City) into a single query. An alternative syntax for these queries is called the **rule-like syntax**. Writing the previous query in rule-like syntax, we obtain the following:

$$\text{Answer}(y) :- \text{Person}(x, y, z),\ \text{Profession}(x, \text{'computer scientist'}),\ \text{City}(z, \text{'Athens'}, \text{'Greece'}). \tag{2.3}$$

Where *Answer* is a relation that does not exist in schema **S**, the y attribute of *Answer* is the value we are looking to extract from the query (in this case, the name of the relevant *persons*). $Answer(y)$ appearing to the left of :- is called the **head** of the rule, while the expressions to the right side of the :- are called the **body** of the rule. A conjunctive query is regarded as a *boolean* if it has no output variables (meaning $\bar{x}$ is empty). In the rule-like syntax example above, we change $Answer(y)$ to *Answer* to convert the conjunctive query into a boolean conjunctive query.

To evaluate a conjunctive query q onto a database instance D, it is easiest to look at it in its rule-based form; as such, we proceed with formula 2.3. The body of the conjunctive query can be evaluated as a pattern that must be matched with a given database instance. This is done via an assignment η that maps the variables present in the body, (x, y, z) in this case, to constants in D. Whenever a variable gets assigned a constant in this mapping, it must be correct for all atomic formulae present in the body. If, for example, a mapping (1,'Tom', 2) is made, the following three tuples must be present in the database: (1,'Tom', 2) in the *person* table, (1,'*computerscientist*') in the *Profession* table and (2,'*Athens*','*Greece*') in the *City* table. As such, we retrieve $Answer$('Tom') from the conjunctive query q and have found an answer. As with other querying forms like SQL, the query can output multiple different values, in this case meaning that multiple valid mappings were found. When finding a valid mapping η for a database instance D, η is said to be **consistent** with D. This definition can be written as the following formula:

$$\{R_1(\eta(\bar{u}_1)), \ldots, R_n(\eta(\bar{u}_n))\} \subseteq D \tag{2.4}$$

In the case of a *boolean* conjunctive query, it evaluates to *true* if and only if a mapping can be made that is consistent with the relevant database instance. This means that the output of the query contains tuples; if it were to evaluate to false, the output would be empty.

2.1.3 Hypergraphs and Acyclicity

It is possible to view a conjunctive query as a **hypergraph**. A hypergraph is a generalization of the regular undirected graph structure where edges can create connections between more than two vertices. Formally, a hypergraph is a pair $H = (V, E)$ where V is a finite set of *nodes* and E is a set of subsets of V which are called *hyperedges*. This notion was studied using chapter eighteen from [2] and lecture notes from [13].

Example 2.1.2. An example hypergraph would be $V = \{a, b, c, d\}$ and $E = \{\{a, b, c\}, \{b, c, d\}\}$ as visualized below in figure 2.1.

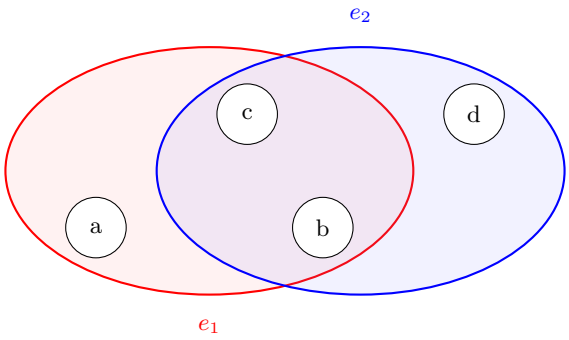


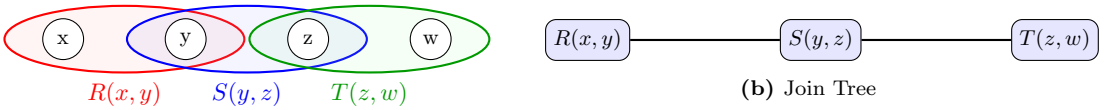
Figure 2.1: An example of a simple hypergraph.

To transform a conjunctive query q into a hypergraph $H = (V, E)$, the set of variables present in q equals the set of *vertices* V. The set of hyperedges E consists of the sets of variables appearing in the atomic formulae of q. A conjunctive query that corresponds to the hypergraph from Example 2.1.2 could be: Answer(a) :– $R(a, b, c)$, $S(b, c, d)$ in rule-like syntax.

When evaluating a conjunctive query q, there is a difference between whether the query is **cyclic** or **acyclic**. In the cyclic case, the hypergraph representing q contains at least one cycle. This cycle makes it difficult to compute query q since some joins depend on each other, forcing repeated operations that often produce large intermediate results. When q is an acyclic query, however, query evaluation becomes easier to manage, allowing for certain specialized algorithms like the **Yannakakis** algorithm described later. In practice, conjunctive queries are often *tree-shaped*; as such, we continue with handling query evaluation of **acyclic** conjunctive queries.

To formally define *acyclicity* for hypergraphs, several non-equivalent notions exist. We use the notion of **α-acyclicity** which is defined as follows. A hypergraph is said to be *acyclic* if it admits a **join tree**. This is the case when its hyperedges can be arranged to form a tree structure, while preserving the connectivity of elements that occur in different hyperedges. Formally, a *join tree* for a hypergraph $H = (V, E)$ is a tree T having the hyperedges of H as nodes such that the following condition is satisfied for every variable $x \in V$: the set of all hyperedges $e \in E$ in which x occurs forms a connected subtree of T. A connected subtree is present for an element when each occurrence of that element can form a path to each other occurrence of that element, by only following vertices containing it. A violation of this connectedness property is visible in Figure 2.3.

Example 2.1.3. To support this definition, a concrete example of an acyclic hypergraph and its join tree follows: using the hypergraph $H = (V, E)$ for the boolean conjunctive query $q :- R(x, y), S(y, z), T(z, w)$, an example taken from [13]. Its set of *vertices* is $V = \{x, y, z, w\}$ and its set of edges is $E = \{\{x, y\}, \{y, z\}, \{z, w\}\}$. When looking at this hypergraph as visible in Figure 2.2, it is intuitively visible that this is an acyclic query. The join tree of this query can be found in Figure 2.2b. The nodes of the join tree are the set of hyperedges in E and each variable v is fully connected with a subtree.



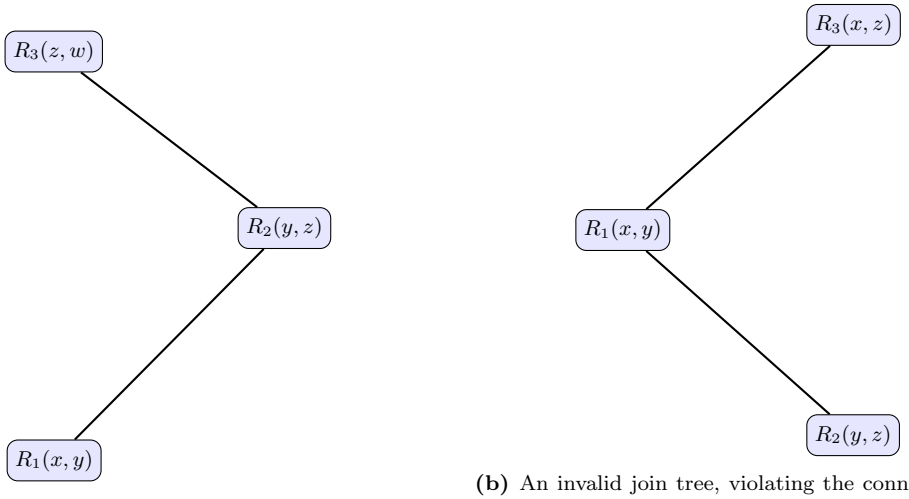
(a) Hypergraph

(b) Join Tree

Figure 2.2: A hypergraph and Join Tree for $q :- R(x, y),\ S(y, z),\ T(z, w)$

Example 2.1.4. An example of a cyclic query is the following: $q :- R(x, y), S(y, z), T(z, x)$, also taken from [13]. This hypergraph intuitively has a cycle since the variables x, y, and z are shared across the atoms in such a way that one can traverse a loop: x appears in both R and T, y appears in both R and S, and z appears in both S and T, forming the cycle $R \to S \to T \to R$.

Figure 2.3 also gives an example of a join tree where the occurrences of each element form a connected subtree and an example of when this is not the case. In Figure 2.3a, a join tree for query $q_1 :- R_1(x, y), R_2(y, z), R_3(z, w)$ is visible. The occurrences of the elements x, y, and z are all connected, and as such, the query is α-acyclic. In Figure 2.3b, an invalid join tree for query $q_2 :- R_1(x, y), R_2(y, z), R_3(x, z)$ is visible. From the tree structure, we can conclude that the query is not α-acyclic, since the occurrences of the variable z do not form a connected subtree in the join tree. Following the usage of join trees alongside given conjunctive queries, we now need a standardized way to construct join trees to prove that there is or isn't a join tree T for the query. An algorithm that can be used to do this is presented in the following subsection.



(a) A valid join tree

(b) An invalid join tree, violating the connectedness property

Figure 2.3: Join trees used to check α-acyclicity.

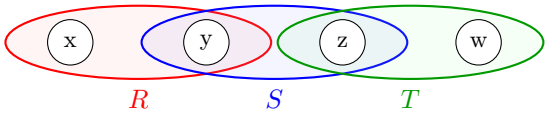
2.1.4 The GYO algorithm

To execute specialized algorithms on acyclic queries, there needs to be a way to recognize the acyclicity of a conjunctive query. To this end, the GYO algorithm is an iterative algorithm that repeatedly applies two operations until it has either processed the entire hypergraph or can't proceed any further. If the algorithm successfully processes all vertices and hyperedges from the hypergraph, the hypergraph itself is **acyclic**. If it cannot proceed at any point, while still having parts of the hypergraph remaining, the given hypergraph is **cyclic**. The information presented in this subsection was gained from [13] and chapter eighteen of [2]

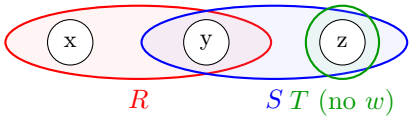
Taking a hypergraph $H = (V, E)$, the algorithm states that it is acyclic *if and only if* all of its vertices and hyperedges can be deleted by repeatedly applying the following two operations (in no particular order):

1. Delete a vertex that appears in at most one hyperedge.
2. Delete a hyperedge that is contained in another hyperedge.

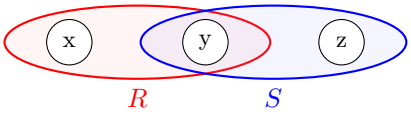
Taking the hypergraph from Figure 2.2a, the GYO algorithm's steps are visualized in Figure 2.4. After the first four steps, the algorithm then removes either the y or z node, after which it will remove the node that is left. Having processed all vertices present in H, it correctly concludes that the hypergraph is indeed **acyclic**.



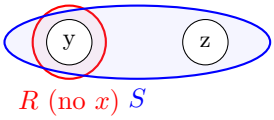
Step 0: Initial hypergraph



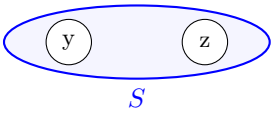
Step 1: Remove w because it only appears in T



Step 2: Remove T because it is contained within S



Step 3: Remove x because it only appears in R



Step 4: Remove R because it is contained within S

Figure 2.4: GYO Reduction Steps for $q :- R(x, y),\ S(y, z),\ T(z, w)$

In the case of $q :- R(x, y), S(y, z), T(z, x)$, the algorithm can't even take a first step since no vertex is in at most one hyperedge, nor are there hyperedges that are contained within another hyperedge. As such, the algorithm correctly concludes that this is a **cyclic** hypergraph.

This algorithm can also be extended to construct the join tree T of the given hypergraph $H = (V, E)$. Whenever a hyperedge e is removed because it is contained within a different hyperedge f, a connection between e and f is made. Following this, during the procedure shown in figure 2.4, the join tree shown in figure 2.2b is obtained.

The GYO algorithm can be implemented in linear time relative to the size of the query. Since it is possible to create a join tree for a given acyclic query during the GYO algorithm, we know that a join tree can be created in linear time as well. [13]

2.2 Efficient evaluation of acyclic queries

Using the concepts explained in section 2.1, this section contains the necessary information to evaluate acyclic conjunctive queries. First, it explains an important operator used to facilitate this, after which an algorithm using this operator is explained.

2.2.1 The semijoin operator

The following information was gained from chapter nineteen of [2] and the lecture presented in [13]. As alluded to earlier, certain algorithms only work on acyclic conjunctive queries. Now that we know how to check the acyclicity of a conjunctive query, we can begin describing these specialized algorithms. For a given acyclic conjunctive query q, an interesting optimization would be to discard tuples from all relations that appear in the body of q that will not participate in the final result of q. In doing this, the amount of work the join operations need to perform can be reduced, leading to a more efficient evaluation of q. These tuples are called **dangling tuples**. One way to remove these is via the **semijoin** operator. A semijoin operation between to relations R and S with $att(R)$ being the attributes of R, is defined as:

$$R \ltimes S = \pi_{\text{att}(R)}(R \bowtie S) \tag{2.5}$$

As such, the result of a semijoin operation features all tuples from the first relation R with at least one joining tuple in the second relation S. Tuples that share equal values in all overlapping attributes are said to be **consistent** and are present in a semijoin operation's result. Following this definition, we conclude that the semijoin operator allows us to remove *dangling tuples* that would occur in a join between R and S. This operator's result can be seen as a pruned version of the original relation that minimizes unnecessary data by eliminating tuples that would not contribute to a successful join.

The semijoin operator can create a **full reducer**. This is a sequence of semijoin operators that removes all dangling tuples from all relations that appear in the body of a conjunctive query. If an instance of a database contains no dangling tuples, it is called **globally consistent**. Suppose the full reducer is applied to the conjunctive query before evaluation. In that case, the evaluation itself will be more efficient since the subresults that would not contribute to the final results are pruned.

Example 2.2.1. A possible full reducer of the query $q :- R(x, y),\ S(y, z),\ T(z, w)$ from 2.2 would be the sequence:

$$S := S \ltimes T, S := S \ltimes R, R := R \ltimes S, T := T \ltimes S \tag{2.6}$$

To obtain a full reducer, we again turn to the GYO algorithm from Subsection 2.1.4. Whenever a hyperedge e is removed from the graph because it is contained within another hyperedge f, we compute the semijoin $f := f \ltimes e$. After the algorithm terminates, we also perform the semijoins in the reverse order.

2.2.2 Yannakakis's algorithm

Using the concept of full reducers, Yannakakis's algorithm can be used to evaluate acyclic conjunctive queries. This query uses the join tree T of a given query q that is to be evaluated. The input of this algorithm is an acyclic conjunctive query q and a database D; the output is the evaluation of query q on database D, which can be written as $q(D)$. What follows is an explanation of the algorithm as described in [8].

The algorithm is split up into two main steps, the first is a **semi-join reduction**. This reduction is done in two sweeps, one following the join-tree in a bottom-up order and the other

following it in a top-down order. In the second step, the join tree is used as a query plan to perform the actual joins. Since creating a join tree and constructing a full reducer via the GYO algorithm involve the same operations, a full reducer can be derived from the join tree.

Example 2.2.2. Taking the query $Q(y, z, p, w, x, u) :- R(y, z), S(p, w), T(x, y, z), U(z), W(y, z, u)$, its join tree T is shown in Figure 2.5:

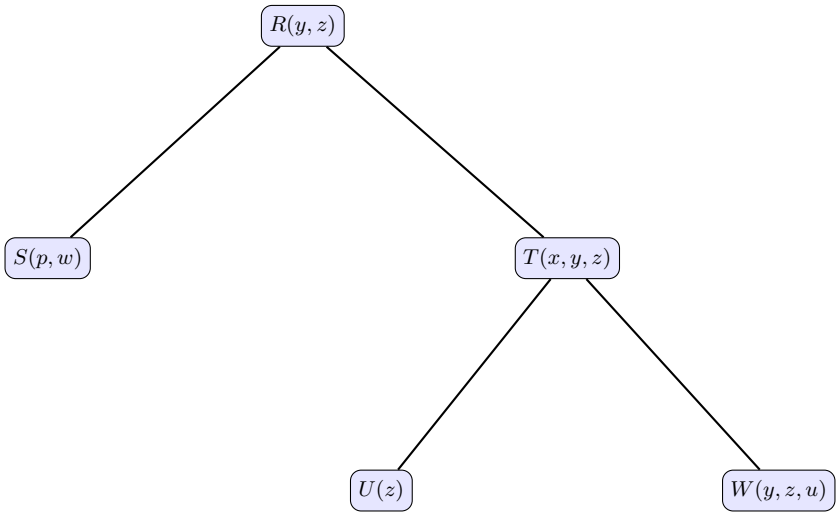


Figure 2.5: Join tree T for query q

Now both reducer passes necessary in the algorithm become clear. The reducer of the first pass performs the following sequence of semijoins: $T := T \ltimes U, T := T \ltimes W, R := R \ltimes S, R := R \ltimes T$. This sequence completely removes all dangling tuples from the root node R. If q were a boolean query, checking whether or not R contains any tuples would yield the answer to the query. Since q is not, the algorithm continues to the second sweep of top-down semijoin reductions. A possible sequence of semijoins performed during this phase is: $S := S \ltimes R, T := T \ltimes R, U := U \ltimes T, W := W \ltimes T$. With this, a full semijoin reduction has been completed, and the database is said to be globally consistent and all dangling tuples are removed. As such, the algorithm can proceed to the joining phase.

Now the algorithm can start computing the results of query q. It does this by incrementally computing the join results of connected tables in the join tree. This is possible using either a top-down or a bottom-up traversal. Choosing a top-down traversal allows us to combine the second sweep of the first phase with the computation of the joins, removing one traversal of the join tree. Since a full reducer was applied to the database before computing the joins, each join can now only increase the size of the results. There will never be any rows present in any subresult that will be removed in a later join, since this would be a dangling tuple.

Yannakakis's algorithm efficiently evaluates acyclic join queries in $O(|Input| + |Output|)$. The $O(|Input|)$ complexity corresponds to the first phase of the algorithm, with $|Input|$ being the size of the input relations, where dangling tuples are removed via semijoins in a bottom-up and top-down traversal of the join tree. This phase only processes the input relations and does not generate intermediate join results. The $O(|Output|)$ component corresponds to the second phase, with $|Output|$ being the size of the output, where the final join results are constructed by traversing the join tree and generating output tuples that satisfy all join conditions. Due to the first phase of the algorithm, no intermediate result can exceed the size of the final output, ensuring that the entire evaluation remains efficient and avoids the explosion of subresult size common in join operations called the **diamond problem** [5].

2.2.3 Practical use of Yannakakis's algorithm

While Yannakakis's algorithm provides strong theoretical promises of faster query evaluation, when implemented as-is, it lacks practical use in many real-world scenarios. When joining two tables in a real-life database system, the join attributes are often primary and foreign keys. This key-foreign key structure inherently limits the size of intermediate results as well as the final join output. As a result, the overhead of performing semijoin reductions—especially the traversal of the join tree and the multiple passes over data—can outweigh the benefits, particularly when the joins already filter out a large amount of subresults. In such cases, traditional join algorithms optimized by query planners and indexes often outperform Yannakakis's approach in practice, despite its superior worst-case guarantees.

As mentioned in [24], testing Yannakakis's algorithm in DuckDB, it does indeed perform worse than the regular query plan DuckDB would create. When removing the key constraints and duplicating the dataset multiple times, however, the duration of Yannakakis's algorithm stays the same while the duration of the regular query plan skyrockets. These measurements prove that the theoretical guarantees of the algorithm are still valid, even though they do not come to fruition in most practical cases.

All of this does not mean that Yannakakis's algorithm is unusable in practice. Works like [4] have created efficient implementations of the algorithm within existing database engines. The rest of this thesis will build upon the results of [4]

2.3 Shredded Yannakakis

This section aims to explain the method proposed in [4] to efficiently implement Yannakakis's algorithm in existing database engines. To explain this, we must explain **nested semijoin algebra** as well as the notion of **query shredding**. These concepts are important to the actual content of the thesis, since the implementation of this approach was parallelized during it.

2.3.1 Basis of the shredded approach

The basis of the shredded approach provides a different approach to processing computationally expensive joins, which can then be used to provide the same guarantees as the original algorithm without requiring additional semi-joins [5]. This method decomposes the traditional hash-join operation, which is often used in real-life database operations, into two distinct sub-operations. These are called **lookup** and **expand**. Lookup finds the first match in the hash table made for the build side, while expand iterates over the rest of the matches. These two sub-operators now serve different purposes; lookup is an operator that either keeps the number of rows present unchanged or shrinks it, while expand will do the opposite and grow the subresults. This approach can also be compared to pruning the subresults, since the lookup identifies and discards combinations that do not have a matching key in the build side's hash table. Finding an initial match filters out a large number of non-matching tuples from the probe side before the expansion phase, thus reducing the overall work and the size of intermediate results.

The shredded approach will now expand upon the Lookup & Expand approach, defining it in terms of a set of nested relational operators. This set is called the **Nested Semijoin Algebra** and allows Lookup & Expand plans to be executed in interpreted query engines.

2.3.2 Nested Relational algebra

The Nested Relational model allows for nesting objects within relations. An example would be taking a relation with schema **Department**(depId, depName) and one that uses the previous relation **Company**(name, departments). In this example, an object of type Department is a tuple with two components; it has a depId, which could be an integer denoting its unique

identifier, alongside a depName, which is a string denoting its name. The company scheme has two attributes: a depId, which could again be an integer denoting an identifier, and departments, which is a subset of tuples of an instance of the department relation. Instead of linking a department to a company using a key-foreign key structure, using the complex data object model, it is possible to link them using the attributes of the relations themselves.

In the Nested Relational model, two types of attributes are distinguished: **flat** attributes and **nested** attributes. Flat attributes hold atomic values such as integers or strings. For example, depId is a flat attribute because it is a single integer value. Nested attributes, on the other hand, refer to the schemes of nested relations. In the earlier example, the departments attribute in the Company schema is a nested attribute because it holds a set of Department tuples, conforming to the Department(depId, depName) schema.

This data model is used in [4] to create nested data structures when processing the subresults necessary for the join operations.

2.3.3 Query shredding

To represent complex data objects with flat and nested attributes, [4] proposes the use of a shredded representation. Here, a nested relation can be represented using a collection of flat relations. A columnar data format is assumed, where a flat relation $R(x_1, \ldots, x_n)$ is physically represented as a tuple $R = (R.x_1, \ldots, R.x_n)$ where each $R.x_i$ is a vector of length $|R|$. Taking one value from each $R.x_i$ at the same offset produces a tuple of R.

Now taking a nested relation R as a relation with both flat and nested attributes, its **weight** is the total number of tuples produced when flattening R. The **unnest** operation unnests a specified nested attribute in tuple t, turning it into an equivalent flat attribute in one or more new tuples. An example of this would be unnesting the tuple $\{x, \{y, \{z\}\}\}$ into tuple $\{x, y, z\}$. If we were to unnest the tuple, the resulting tuple would be $\{x, y, \{z\}\}$, which would still be nested. To obtain a completely flat tuple, we would need to either unnest it again or use the flatten operator on either this tuple or the original tuple, which completely unnests it in both cases.

Given a scheme $X = \{y_1, \ldots, y_k, Z_1, \ldots, Z_l\}$, $y_1, \ldots, y_k$ denotes the flat attributes and $Z_1, \ldots, Z_l$ denotes the nested attributes. When converting X to a shredded representation shred(X), it becomes shred$(X) = \{y_1, \ldots, y_k, hd_Z_1, \ldots, hd_Z_l, w_Z_1, \ldots, w_Z_l\}$. The flat attributes are kept as-is, while the nested attributes are kept as a pair of hd_Z_i and w_Z_i. The first attribute hd_Z_i contains a pointer to the head of a linked list that represents the content of nested attribute Z_i. A separate list called nxt is used to point to the next tuple of the linked list. The scheme $ishred(X)$ is denoted by the union of $shred(X)$ and nxt. Lastly w_Z_i stores the weight of the nested relation Z_i.

Let us continue with the shredded representation of the data present in a relation. Starting with a nested relation R over scheme X, the nested representation is $\mathcal{R} = (R, \Sigma_R, r)$. Where R is a relation with scheme $shred(X)$, Σ_R is a **store** over X, and r is a selection vector for R. Store Σ_R is a collection of relations, one $\Sigma_R(Y)$ for each nested relation Y in X, where every $\Sigma_R(Y)$ has schema $ishred(Y)$, this being the union $shred(Y) \cup nxt$.

Example 2.3.1. An example of a shredded relation can be found in Figure 2.6. Here, a relation with schema $\{x, y, \{u, \{v\}\}$ containing two nested tuples is shown, alongside its shredded representation. The nxt vectors are used to traverse the linked list to which hd points to. In this vector, a zero denotes no next value, while values higher than zero indicate the index of the next value, starting from one. Starting from hd, following the next vector allows the algorithm to produce all flat tuples during the unnesting process. Flattening this nested relation would yield four flat output tuples: $\{a_1, b_1, c_1, d_1\}, \{a_1, b_1, c_2, d_2\}, \{a_1, b_2, c_3, d_3\}$ and $\{a_1, b_2, c_3, d_4\}$.

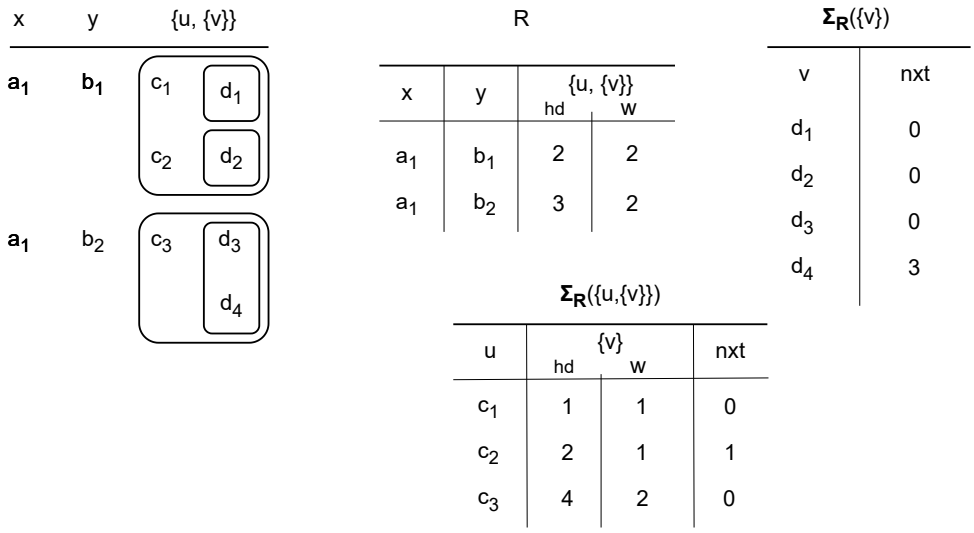


Figure 2.6: Example of a shredded relation R

2.3.4 Main operators of nested semijoin algebra

The nested semijoin algebra proposed in [4] is a set of relational operators that can operate on nested relations. At the core of this nested semijoin algebra lie four operators that can be used to perform join operations. These operators are **Group-by** (γ), **nested semijoin** ($\triangleright\!\!\rightarrow$), **unnest** ($\mu$), and **flatten** ($\mu^*$), alongside the standard relational operators, altered slightly to work on nested relations. Combining the first two operators, it is possible to emulate the evaluation of a hash-join with a nested output.

The first operator *group-by* ($\gamma_{\bar{y}}$) corresponds to creating the hash table necessary for the hash-join operation. The operator will group the tuples present in the relation it is applied to (relation R with schema X) on the attribute(s) specified in $\bar{y}$. The resulting hash map's key values are the attributes specified in $\bar{y}$, while its values are the corresponding attributes present in $X \setminus \{\bar{y}\}$. The attributes of $\bar{y}$ must all be flat attributes, since it is not possible to perform a group-by operation on a nested attribute. The important distinction to make is that after the group-by occurs, all hashmap values are nested together for each separate key. The resulting hash table made during the group-by operation is called a **dictionary**. A dictionary over $\bar{y} \rightsquigarrow Z$ maps $\bar{y}$-tuples as keys to non-empty relations with scheme Z as values.

The second operator *nested semijoin* $\triangleright\!\!\rightarrow$, is applied after the group-by operation. It takes a dictionary D: $\bar{y} \rightsquigarrow Z$ and relation R with schema X. If X is compatible with $\bar{y} \rightsquigarrow Z$, occurring when the attribute(s) of $\bar{y}$ are present in X and that $\mathcal{A}(Z) \cap \mathcal{A}(X) = \emptyset$, meaning that the flat attributes of Z and X do not overlap. This operator will then fulfill the role of probing dictionary D for each tuple t in R. If the dictionary contains t[$\bar{y}$], it will extend t with a new nested attribute Z, containing the corresponding value present in D for t[$\bar{y}$].

The unnest and flatten operators are needed to transform the eventual nested relation into a flat one with easily usable tuples. As mentioned earlier, the difference between the two operations is that the unnest operation only unnests one level of nesting, while the flatten function completely unnests the given nested relation. While the flatten operation can be achieved with a sequence of unnest operations, [4] implemented the flatten operation separately from the unnest operation to reduce overhead and increase performance.

2.3.5 Hash joins via nested semijoin algebra

Using the nested semijoin and group-by operators, a join between two relations can be achieved within the nested semijoin algebra. After performing both operations, a nested relation is obtained; as such, a flatten or unnest can be used to convert this to a relation containing only flat attributes. Joining two relations R and S via the nested semijoin algebra can be formalized via the following formula:

$$R(x, y) \bowtie S(y, z) \equiv \mu_{\{z\}}(R \triangleright\!\!\rightarrow \gamma_{\{y\}}(S)) \tag{2.7}$$

Example 2.3.2. Figure 2.7 shows an example of a join via group-by and nested semijoin. Two flat relations $R(a, b)$ and $S(b, c)$ are joined on their shared attribute b. A dictionary is created for relation R using the group-by operation, after which this dictionary is used in the nested semijoin operation together with flat relation S. The result is a nested operation where the join key and remaining attributes of S are flat attributes, while the remaining attributes of R are nested. When flattening the result of the nested semijoin operator, we receive $\{(b_1, c_1, a_1), (b_1, c_1, a_2), (b_1, c_2, a_1), (b_1, c_2, a_2), (b_2, c_3, a_3)\}$.

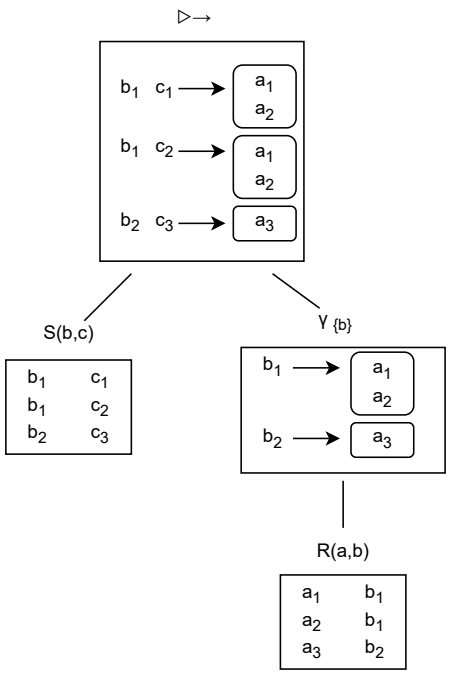


Figure 2.7: Join example between two tables using NSA operators

A structure for the query plan is specified to evaluate joins efficiently within the nested semijoin algebra. Both the nested semijoin and group-by are linear operators and, as such, generate outputs with a cardinality that is linear with their inputs. Taking the nested semijoin, each tuple in R can produce one tuple in $R \triangleright\!\!\rightarrow S$ at most, so this output cannot increase in size. As such, the nested semijoin can't create dangling tuples, just like the regular semijoin. On the other hand, the unnest and flatten operators are non-shrinking operators, meaning that they can potentially create dangling tuples. Because of this, they are always preceded by the join operations in 2-phase NSA. When they are used in this manner, they simply create their output as flat tuples and do not create any unnecessary tuples.

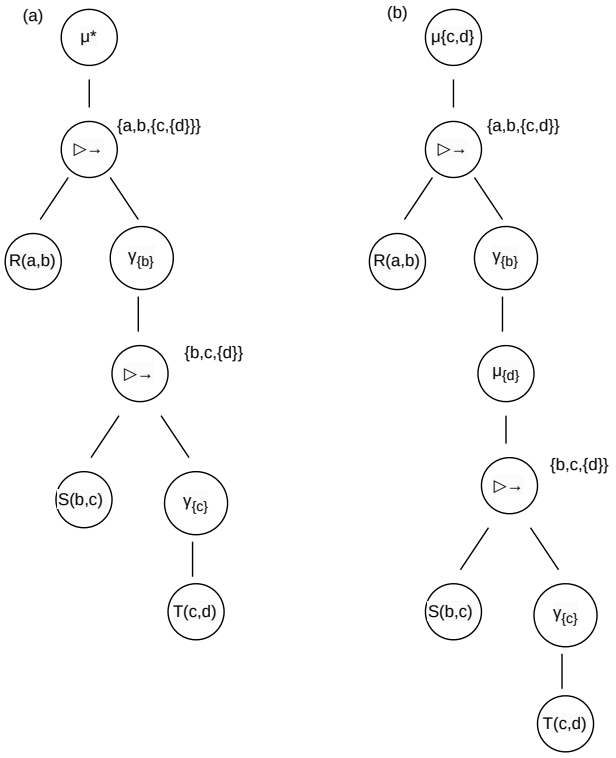


Figure 2.8: (a) A 2-phase NSA plan. (b) A non 2-phase NSA plan, with unnest applied before linear operators.

To create an efficient query plan, [4] now proposes the notion of **2-phase NSA expressions**. For an NSA expression to be 2-phase, every unnest and flatten must have only non-shrinking operators as its ancestors within the query tree. This constraint ensures that the operators capable of increasing the subresults are only applied after all operators that keep the sizes of their outputs linear to their inputs. This order avoids the risk of unnecessary subresult expansion early in the evaluation, which could degrade performance. It is then stated that a given join query Q can be evaluated by means of a 2-phase NSA join plan if and only if Q is acyclic. Shredded Yannakakis can then evaluate these 2-phase NSA expressions in time $O(In + Out)$, the same time complexity as the original Yannakakis's algorithm 2.2.2. Figure 2.8 is based on Figure 8 of [4], it shows an example of a 2-phase NSA plan (a) where the flatten operator is only called at the root of the tree. Query plan (b) also displays a non 2-phase NSA plan, where an unnest operator is placed between a nested semijoin and a group-by operator.

2.4 Usage of the algorithms within this thesis

For the practical aspects of the thesis, an implementation created for [4] was studied and extended. It implements the nested semijoin algebra operators in Apache DataFusion [1] 3.3 to measure the efficacy of the shredded approach. Alongside the operators, the shredded data structure was also implemented to be used alongside the NSA operators. Since Apache DataFusion is a column-store database system, it lends itself nicely to using the shredded approach since it allows efficient column-level access.

Chapter 3

Database systems & parallellism

Since the goal of the thesis is to add parallelism to a database implementation, basic terminology and concepts of databases are illustrated within the first section this chapter. Later sections cover a more specific approach to query evaluation, alongside a specific database implementation named Apache DataFusion, which was used throughout the thesis.

3.1 Introduction to database systems

Database systems are used as the backbone of many information systems. They allow us to store data and query it to retrieve information. This section covers basic terminology of databases, explaining how a database generally processes information. Through these explanations styling rules and conventions of the second chapter from [7] are used.

3.1.1 Structures within databases

To illustrate a possible structure of a database, an example of a relation is shown in Table 3.1. In this table, each row represents a single product in a relation featuring information about products present in a store. The schema for this relation is *Products(product, category, price, supplier)*. When this relation is inserted into a relational database, it can be interacted with using SQL (Structured Query Language). This allows managing the data present in the relations by allowing insertions, updates, or deletions and querying this data in a standardized fashion.

Product	**Category**	**Price**	**Supplier**
Laptop	Electronics	1000	TechCorp
Smartphone	Electronics	700	MobileMakers
Headphones	Accessories	200	SoundCo

Table 3.1: Example instance of the Products relation

The following explanation was written using chapter 15 and 16 of [7] as a refresher. Since database management systems allow their users to query and manage data, their interface (e.g. SQL) needs to be translated into actionable execution plans. Its so-called query processor handles this task; this component of the database management system is responsible for both query compilation and execution. The first step in translating an SQL statement into an actionable query is parsing. The query compiler parses the given SQL query into a *parse tree*.

This tree structure is then translated into relational algebra as an intermediary step between the original query statement and an actionable plan. In this stage, the query is called a **logical query plan**. It will describe the steps necessary to execute the query in a high-level, declarative manner, specifying what data needs to be retrieved and how the relations should be combined, but without committing to specific implementation details like which algorithms or indexes to use. An example for the running product example would be to translate the following query.

```
1 SELECT Orders.customer, Products.price
2 FROM Products, Orders
3 WHERE Products.product = Orders.product AND
4       Products.product = laptop
```

It retrieves data about orders and products from two tables: Products(product, category, price, supplier) and Orders(customer, product, quantity). More specifically, the query asks which customers bought a laptop at what price. The query can then be translated into the following relational algebra expression:

$$\pi_{\text{customer,price}}\,(\sigma_{\text{Products.product=Orders.product} \wedge \text{Products.product='laptop'}}(\text{Products} \times \text{Orders}))$$

The logical query plan can be visualized as a graph structure, more specifically, a tree structure which we will call an **execution tree**. The leaves of the tree structure represent the tables present in the database from which data is extracted. Other nodes in the structure represent logical operators that the data will flow through to compute the result. Examples of operators present are *Selection, projection & filter*. The earlier relational algebra expression can be translated into the following execution tree.

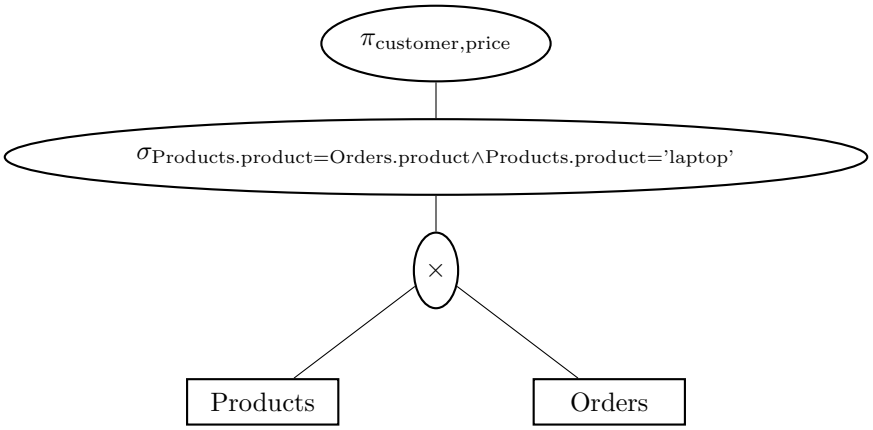


Figure 3.1: Simple query plan

Typically, the query processor will then optimize the logical query plan before translating it into later stages. This *logical plan optimization* features rewrites of the relational algebra statements in order to improve efficiency in later stages. An example would be to change the Cartesian product present in Figure 3.1 to a more efficient theta join $\bowtie$. This way, unnecessary intermediate results are avoided, reducing the overall computational cost and improving query performance. Another straightforward optimization, for example, states that the fewer nodes left from this stage onwards, the faster the eventual execution will be. A more concrete example of this would be the removal of redundant joins. In general, joins are some of the most computationally expensive operations in relational databases; hence, there is a need to remove joins that do not contribute to the result. Another optimization that is applicable to the earlier query plan would be the pushing of selections. Our query plan only has a selection for the Products table. If the query compiler added the same selection above the Orders table, fewer tuples would enter the computationally expensive join operator. As a result, the join operator

would have less work to do. Since the selection operator is computationally less expensive, in general, this query plan is more efficient than the previous plan (although on a case-by-case basis, the previous plan could be more efficient).

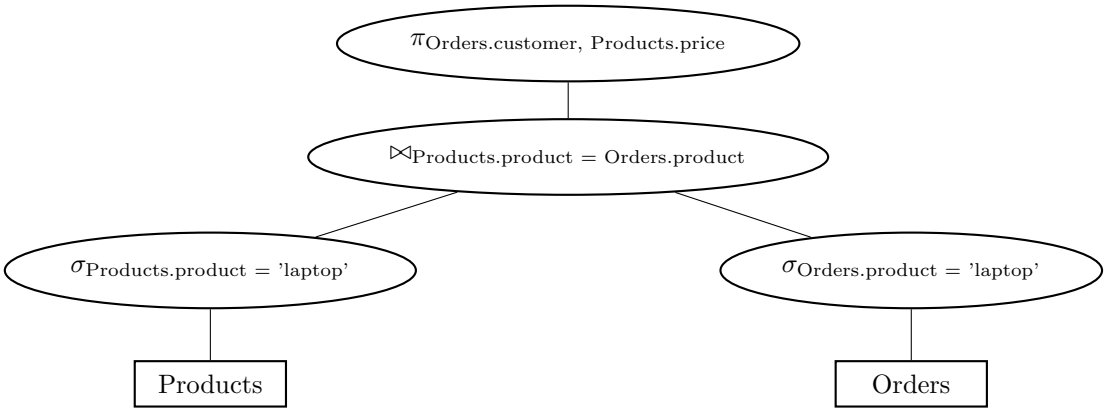


Figure 3.2: Optimized query plan

The query processor of a database management system will then translate the *optimized logical query plan* into a **physical query plan**. This physical plan comprises different, specific algorithms corresponding to different operators in the logical plan. As such, the query processor must select the most appropriate algorithms for each logical operator, as multiple physical operators can implement each logical operator, each optimized for different scenarios. This is a cost-based analysis where the cost can, for example, be expressed in the number of I/O operations the system needs to make during query execution. To make these decisions, the query optimizer will consult statistics; these include measurements like table size or data layout on disk to compute the amount of disk I/Os needed for a particular algorithm and compare it to other physical operators. This step can be seen as changing the operators in the tree structure nodes to more specific algorithms, for example, converting a join node into a hash join node. This query plan also includes helpful information during query execution, like how relations should be accessed and whether or not a relation should be sorted, since sorting relations in earlier operators can ease later computations. Once a physical query plan is realized, the query engine will be able to execute it, streaming the data present in the database through the chosen operators in the tree structure. It does so in a bottom-up fashion, starting with reading the input relations, which are present as leaves within the tree structure. The evaluation then works its way upward through the operators. Each internal node will receive data and execute its operation, sending it onward to the next operator. Eventually, reaching the root node where the results can be collected and returned to the user.

3.1.2 Data storage types

The database engine used during the realization of this thesis features data stored in a columnar format. To explain why this would be desirable over the more traditional and intuitive row format, a comparison of the two follows. Figure 3.3 is a visual representation of the difference between row and column storage. In row storage, data is stored as tuples, translating to one array for each tuple. This provides a simple and intuitive way of retrieving data, where retrieving row n retrieves the entire tuple. This approach means that whenever the data of a tuple needs to be accessed, the entire tuple needs to be read, regardless of the number of attributes needed. This can either be a positive or a drawback. If an operation inherently requires multiple attributes from each tuple, data can be accessed efficiently. The problem is that whenever an operation only requires a single attribute, the entire tuple needs to be read into memory. Since a larger amount of data is retrieved than necessary, this needs more cache memory than it ideally would, overwriting other data present in the system cache and possibly forcing a reread on that data. With the less traditional columnar storage, data is stored vertically, translating to one array for each column. This allows for more efficient data access when performing certain analytical operations [22].

An example of this would be computing the average age of the people present in the mock-up database visible in Figure 3.3. In the row-storage case, the system would need to read nine fields of data (all three tuples in their entirety). Using columnar storage, the system would only need to read the age column, meaning that only three fields of data need to be read. In this case, the columnar format would be more efficient. Columnar data storage does, however, lack when operations are write-heavy. Writing a new tuple to the database requires the data to be written into each column separately, creating a similar effect to the drawbacks of row storage.

In conclusion, both row-based and column-based storage have their strengths and weaknesses. Depending on the purpose of the database system, one may very well be preferred over the other.

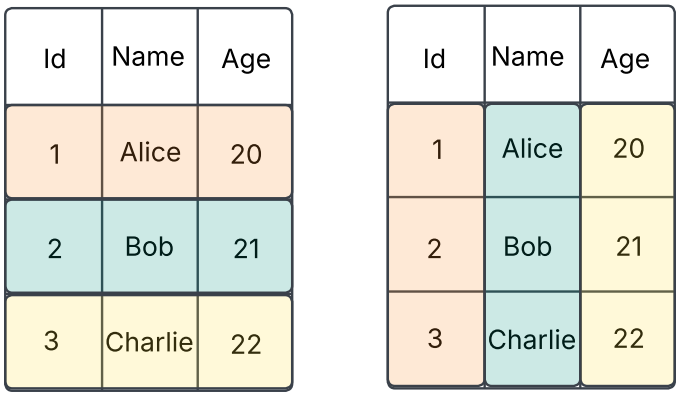


Figure 3.3: Row & column storage

3.2 The Volcano query processing system

Having created the physical query plans as explained in section 3.1, a query engine requires all of its implemented physical operators to be structured in the same way so it can slot any operator before or after any other operator. Since different queries evaluate different plans, operators can occur in several places in query plans. One way to realize this is to follow the Volcano query evaluation system [10]. It provides an interface that the operators can implement to provide standardized data flow; this way, each operator knows the structure of the data it receives and can implement it accordingly. This environment also provides a way to execute queries in parallel, implementing an operator to partition data and spawn different subtasks to enable parallelism. Volcano promises to be an extensible system, allowing users to create new operators that work alongside existing operators as long as they follow the established paradigm. Although the origins of this system are quite old, implementations of it are still being made and used, examples of which include Apache DataFusion and previous iterations of DuckDB [19]. This section aims to explain the original Volcano operator model as relevant to the thesis. Later sections explain a specific implementation of the operator model in Apache DataFusion.

3.2.1 The Volcano model as iterators

In the Volcano system, algebraic operators are implemented as iterators. This is facilitated by the operator interface mainly consisting of three different calls, namely **open, next** & **close**. Using these calls, operators can communicate and retrieve data from other descendant nodes. The *open()* procedure will serve as the initialization call for an operator. An example of this can be the materialization of the build side in a hash join in order to create the dictionary used during the join itself or simply the allocation of memory to store this dictionary. An *open()* function will typically call the *open()* function of its child nodes, propagating it throughout the query plan. In the original Volcano model, each operator has its associated state record. This data structure holds arguments for the operator (its state). To create the query results, the *next()* function of the top-most operator is repeatedly called. The purpose of this function is to retrieve a batch of data from its child operator, execute its calculations on this batch, and return the new batch. The *next()* operator, however, is not obligated to poll its child operator every time it is called. If it still has data left over that was not used in a previous procedure call, it might use this and return the result of its calculations on this leftover batch. Using the *next()* procedure calls, data flows through the operator nodes present in the query plan, starting from the tables present in the relational database, and gets modified to the eventual end result. Lastly, the *close()* function is called when the *next()* function of the root operator receives the *end-of-stream* message. Like the others, when *close()* is called on the root operator node, it propagates throughout the tree. The goal of the *close()* function is to shut down an operator, for example, freeing any memory that was allocated during the execution of this operator[10].

Using these three operations, the iterator structure becomes clear. Once the *open()* call has been propagated throughout the tree, *next()* calls begin to propagate. These calls will iteratively build the query results. As such, the promise made by the system to be extensible is fulfilled as long as new operators are implemented with this iterator structure in mind. Volcano uses *anonymous inputs* or *streams*, meaning that an operator does not need to know where its data comes from or what operations have been performed on this data. This way, the prerequisite of being able to slot any operator above or underneath any other operator is fulfilled. Since we must implement operators in a way that standardizes their output, we know the structure of this data and can implement all standard database operators with this knowledge.

A concrete implementation of this model is explained using the classic *hash join* algorithm. The algorithm itself is divided into two phases. In the first phase, known as the **build phase**, a hash table is constructed using one of the two input relations, referred to as the build side. The keys of this hash table correspond to the join attribute(s), while the values store the remaining

attributes of the tuples. This hash table is then utilized in the second phase to perform the join. In the second phase, the **probe phase**, the other input relation, referred to as the probe side, is used to probe the hash table. This will determine matches and create output tuples using the keys and values of the hash table as well as the attributes from the tuple used to probe the table. For the Volcano implementation, the build phase corresponds to the *open()* procedure, and the probe phase is included in the *next()* procedure. Every call of the *next()* procedure corresponds to probing the tuples present in the operator, retrieving tuples from child operators whenever necessary. When the child operators send the *end-of-stream* message, the join operator will join the data that it has left and send its own *end-of-stream* message afterward. When the parent operator propagates the *close()* procedure to the join operator, its own *close()* procedure gets called. This procedure can then free up the space allocated to the build table, cleaning up traces left by the operator [9].

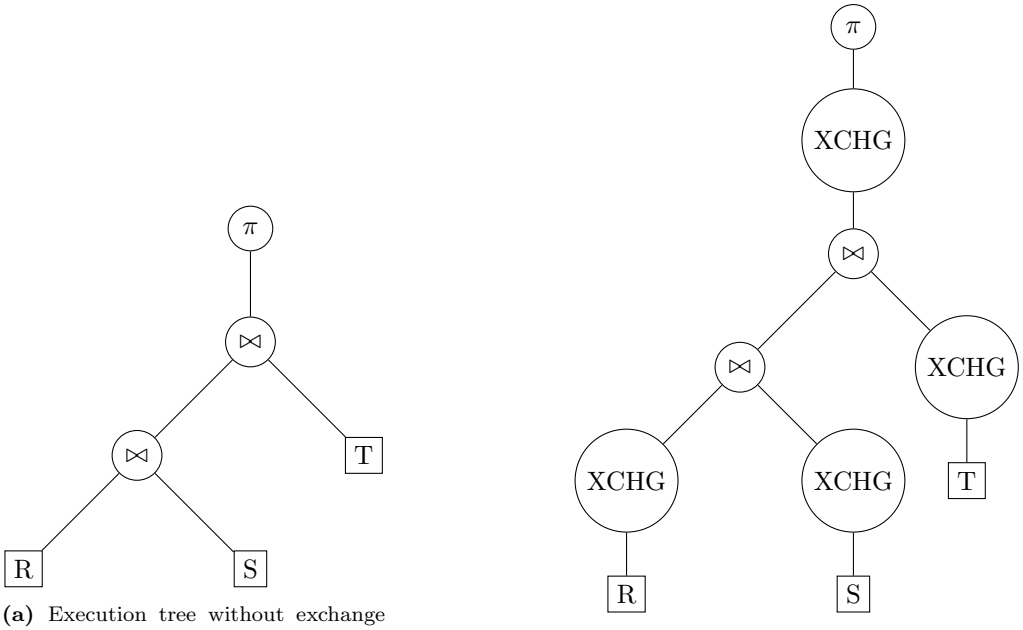
3.2.2 Parallelism in the Volcano model

The following subsection explains a new operator as it is outlined in [9]. The original Volcano model features meta-operators that embody different concepts for query processing. One of these operators is the **exchange** operator. Like the other operators, it can be slotted into a query plan tree. The exchange operator, however, does not perform a calculation on the data like the other operators. The role of the exchange operator is to allow parallelism during the query evaluation process. In general, there are two types of parallelism, **vertical** and **horizontal parallelism**. With vertical parallelism, multiple different operators run simultaneously. This way, multiple iterators can run concurrently, improving execution speed. With horizontal parallelism, a single operator can be run over multiple workers. To facilitate this, data needs to be shuffled across workers so every worker has the correct portion of data. The exchange operator aims to achieve both forms of parallelism while keeping the same interface as the other operators in the paradigm.

To enable vertical parallelism, the exchange operator is made to create a new process on which the operator can run alongside a shared memory structure to communicate between processes called a **port**. These are created during the open procedure call of the operator. The new process can be created using a UNIX fork, for example. This will create a child process that is identical to the original parent process. The exchange operator will follow a different route in the two different processes, creating a producer-consumer relationship between the child and parent. The role of the producer (child) process is to function as a normal iterator. It will pull data from its own child operators when its next procedure is called. The producer process will call the next procedure of its child operator as long as the end-of-stream tag has not been received. It does so independently of its parent operator, effectively becoming a driving force in the execution tree. This is comparable to the root node of the tree calling next procedures as long as it has not received the end-of-stream tag. When data enters the producer, it will be forwarded to the data port created by the parent. This is where the exchange operator differs from the other operators in the Volcano model. Usually, data flows through the execution tree in a demand-driven fashion, only moving data whenever the next procedure is called. The dataflow between parent and child processes, however, is data-driven, flowing from producer to consumer whenever data is present in the producer. This change was made to reduce overhead by not having to send data requests. When the parent operator of the exchange operator calls its next procedure, the consumer process will pull data from the port, waiting for new data when there is no data present.

The exchange operator also enables horizontal parallelism. Horizontal parallelism is divided into two forms, **bushy** and **inter-operator** parallelism. Bushy parallelism is when different subtrees of the execution tree run in parallel. The difference between vertical parallelism and bushy parallelism is that in vertical parallelism, the different operators are pipelined into each other. With bushy parallelism, the results of the parallelized operators eventually meet later in the execution tree (for example, in a join operator). Inter-operator parallelism occurs when multiple threads or processes run the same operator concurrently. They do this on subsets

which we will call **partitions** of the data present in the database, requiring extra care that the data has been partitioned correctly. Bushy parallelism comes prepackaged with the exchange operator workings as explained earlier, if we insert one or multiple exchange operators correctly into the execution tree. The exchange operator will execute the subtree below it, executing it in a new process, in parallel. Intra-operator parallelism requires a data partitioning mechanism to work, adding this necessity to the exchange operator. This is achieved by making changes to the port data structure. Instead of it being a simple single channel for sending data from the producer to the consumer, it must now support multiple producer and consumer processes. To realize this, the port data structure gets split into multiple channels, one for each consumer process. Each consumer process gets a channel assigned to it and is required to only pull data from said channel. The producer processes use a helper function to determine which channel to send a certain input to. The helper functions can be thought of as implementations of round-robin partitioning, hash partitioning, ...



(a) Execution tree without exchange operator(s)

(b) Execution tree with exchange operators

Figure 3.4: Different execution trees

An example of how the exchange operator should be added to existing query plans is visible in Figure 3.4, based on Figure 5 from [9]. Figure 3.4a shows the standard, single-threaded version of the plan. In figure 3.4b exchange operators were added to this plan. It reads three different tables from the database and joins them via two different binary join operators. The role of the bottom three added exchange operators is to independently read the input relations and partition them. These operators should be attuned to each other to ensure the correct (equal) partitioning scheme. The role of the top exchange operator is to start one or more new processes to compute the two joins underneath it. It will become the driving force that calls the next procedures of these joins and ensures that the original process only handles the projection root node and the consumer side of the exchange operator.

To illustrate the need for correct partitioning, an explanation of the **grace hash join** algorithm follows, based on the presentation from [21]. This is a commonly used algorithm for performing joins on parallel and/or distributed systems. The grace hash join algorithm starts with **declustering** as it is called in [21], although, in this text, we will call it **partitioning**. As visible in 3.5, during the partitioning stage, input relation R is partitioned into N buckets by performing a hash on the join attribute(s). Afterwards, the same is done to input relation S, hashing it with the same hash function on the join attribute(s). Using the same hash function

for both relations is vital to the correctness of the algorithm since this is how we ensure tuples that are to be joined end up in the same bucket. After partitioning is completed, the algorithm will join matching buckets from both relations to build the resulting relation. In a parallel context, each pair of buckets can be joined in a different thread or process, allowing horizontal, intra-operator parallelism. The partitioning phase of the algorithm can, for example, be implemented during the file scan where the input table is originally read. This way there is no need for an extra exchange operator in the execution tree. The output this operator produces is, in turn, partitioned on the original join key. In order to combine the outputs of the different partitions, there is a need for a new meta-operator that is capable of combining data from multiple threads.

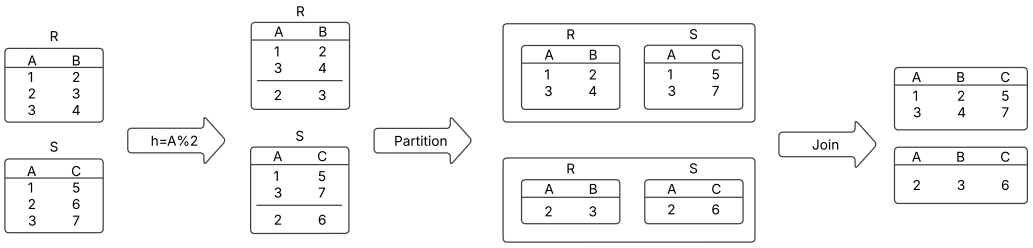


Figure 3.5: Grace Hash Join Algorithm

3.3 Apache DataFusion

During the realization of this thesis, Apache DataFusion was used. It is an extensible query engine written in Rust that uses a columnar data format. DataFusion promises high performance while still being manually extensible to allow its users to tune it to their preferences or to add more functionality or algorithms to the query engine. The engine uses Apache Arrow as its memory model; this makes DataFusion an in-memory query engine featuring columnar data storage. The main goal of DataFusion is to provide a reusable system that can be used as a baseline for other systems, implementing the optimal strategies for the basic elements of a query engine. To write this section, a paper provided by the developers of DataFusion [17], architecture talks explaining the goals and inner workings of Apache DataFusion [14] [15] [16], and the source code [1] were used.

3.3.1 Core concepts in DataFusion

To understand what DataFusion was made for, understanding the concept of *deconstructed databases* is vital. Database systems have been extensively researched throughout history, resulting in a wide array of developed and comparatively analyzed techniques. Most of these techniques, however, are only implemented in strict, tightly integrated databases that limit their reuse for the average user. Since implementing a database system is costly, many systems prefer to focus on a "one size fits all" approach during development, ensuring their wide usage. DataFusion aims to shift this approach towards "fit for purpose" systems, implementing an efficient baseline and a multitude of extension APIs. This makes creating a database system less costly since developers can lean on DataFusion for the basics of their system, skipping implementing aspects that have been implemented numerous times throughout history [17].

Apache Arrow represents data in memory during computations using columnar layouts. It provides a standardized way to represent data and was created with concepts such as cache efficiency in mind. Much like the idea behind DataFusion, Arrow allows its users to avoid

re-implementing basic features needed for database systems. The usage of Arrow enables and significantly enhances DataFusion's columnar processing capabilities. DataFusion allows input data from multiple types of source files. A simple example of this is CSV files. DataFusion allows data sources to be CSV files. A different, preferred data source, however, is **Apache Parquet**. Parquet is a column-oriented data file format that works nicely with Arrow. It was also made with performance in mind, providing structures like indexes and bloom filters for fast data access. It features efficient data compression and embedded schemas, allowing the schema of the data type to be added to the data file itself, making the file "self-describing". Having these technologies strongly implemented in DataFusion is part of what makes it so successful and efficient [17].

To use DataFusion or implement extensions, one needs to adhere to its basic structure. It works as a general high-performance query engine without any manual changes. Adding it to a project is done by simply adding the correct crate (Rust terminology for package) to the project. Running a simple SQL query requires adding the input files (e.g., CSV or parquet files) as tables within a context, specifying a query, and running the SQL query in the context. DataFusion will then create a logical and physical execution plan and execute the query whenever the result is required. The **sessioncontext** is an important part of DataFusion; it allows query execution and maintains a state for an instance of the query engine. Adding data sources is done via this sessioncontext by stating the file to be used and a table name for the data read from this file.

3.3.2 Core structures in DataFusion

A query engine comprises different subsystems that, when used together, allow its users to execute a query in its entirety. DataFusion allows developers to build on top of the existing subsystems it provides in order to customize its behavior. Figure 3.6 displays Figure 2 of [17] outlining DataFusion's architecture; this subsection will explain the different parts of this architecture and the role they play in query execution as summarized in [17].

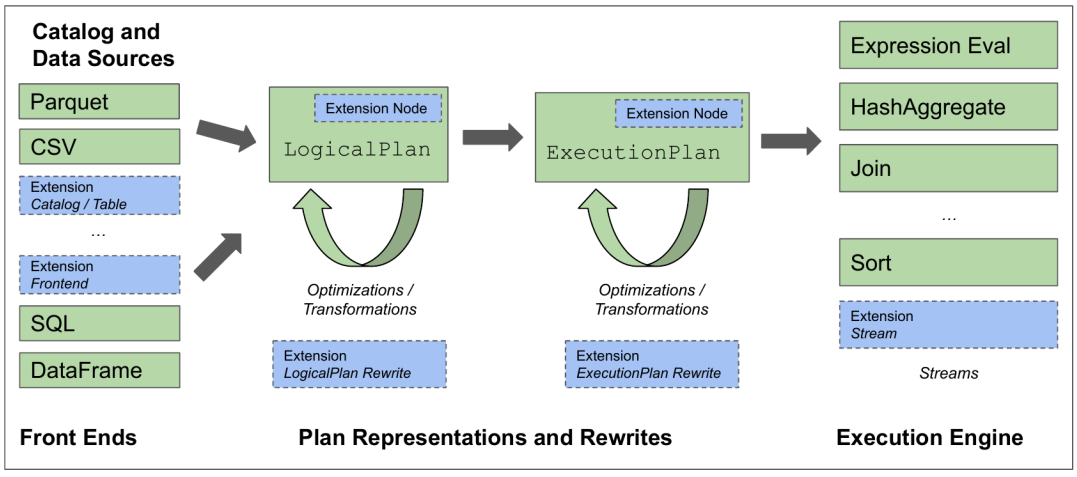


Figure 3.6: DataFusion system architecture

Starting with the input **data sources**, as mentioned earlier, DataFusion supports multiple file formats. A possible extension to this would be to support a file structure that is not yet supported. Existing file formats are supported using the same interface that can be used to create custom file formats for inputting data. The implementation of parquet files makes use of Apache Arrow, increasing their efficiency. **Catalogs** are needed to provide metadata about tables and columns present in the database (sessioncontext). They also provide the data types

that are present in the different tables alongside statistics relevant to the optimization of query execution.

The front end of the query engine consists of three different elements. The **data types** supported by DataFusion are implemented using Apache Arrow's type system. The **SQL planner** is tasked with parsing SQL text and forming a logical plan. An extension to this system would increase the size of the SQL subset the engine supports. A different extension that would be associated with the front end of the engine would be the addition of a new or different query language. Lastly, DataFusion also supports **DataFrames and LogicalPlanBuilders** to offer an alternative to building query plans via SQL. This is also extensible, allowing developers to create their own logical plan builders to further customize DataFusion's behavior.

Next up are the **logical plans** and their **optimizer**. It is possible to create customized logical plans using custom logical expressions. The optimizer that rewrites the logical plan is also customizable, rewrites are executed using *passes* over the logical plan. These passes are extensible and usually feature operations like type coercing and introduction of sort and redistribution operations.

The last and most important for this thesis is the execution engine itself. The operators present follow the Volcano operator model. Operators are implemented using streams, allowing them to produce output incrementally. The rust stream trait allows for the creation of a standardized polling structure [3]. Implementing a stream requires the developer to implement a *poll_next* function that returns the current state of the stream; if there is a data point present in the operator, the function will return Poll::ready(Some(x)), indicating that output was created. If the operator currently has no output that is ready to be propagated, Poll::Pending(None) will be returned, indicating that the caller of the function should try again at a later time. Lastly, Poll::Ready(None) is returned to indicate that the stream has been completed and no new data will be propagated from this stream. Certain operators, like a full sort operator, necessitate a full materialization of the subresults. These operators are called pipeline-breaking operators. With the Volcano operator model, the output of the query is built up incrementally, with data flowing in a **demand-driven** fashion only when the *next()* operator is propagated through the entire query tree. Whenever an operator requires its complete input to produce even its first output tuple, it acts as a pipeline-breaking operator. This is because the next() call cannot be satisfied until the operator has processed all the data from its children in the query tree.

Data flows in the form of **RecordBatches**, bundling tuples into a larger structure, allowing for more efficient computation. The size of a RecordBatch is variable, and it is important that a suitable size is chosen. Large RecordBatches allow us to call fewer iterations of *next()*, possibly allowing for a faster query runtime. However, this must be balanced against the fact that a large RecordBatch carries a longer individual processing time per batch per operator, also pressuring memory consumption at runtime. Parallelism is achieved using an *exchange* operator that works as explained in section 3.2.2. This allows a runtime to use multiple instances of the same operator in order to run them in parallel. This would mean that a single operator is split up into multiple streams, each handling their own partition of data that flows throughout the query tree. To schedule the parallel threads, the **Tokio** runtime is used. It is a library present in Rust, designed for asynchronous network I/O and renowned for its efficiency. Memory is managed using a shared memory pool, operators record their current memory consumption via function calls on the shared pool. Extensions to the execution engine include new operators or a different memory management scheme.

With these different sections of the query engine being extensible, DataFusion offers a highly customizable programming environment, allowing many different specialized database systems to be made from a shared baseline.

3.4 Parallelism in DataFusion

Understanding the implementation of query parallelism in DataFusion was an essential part of the thesis. The implementation explained later on is based on DataFusion's implementation since it was made to parallelize custom DataFusion operators. The goal of this section is to provide details on the exchange operator as implemented within the original DataFusion source code.

3.4.1 DataFusion operator structure

When talking about **operators**, we specifically refer to implementations of physical operators present in DataFusion. These operators are used as nodes in physical plans made by DataFusion. To qualify as an operator to be used in a DataFusion runtime, the programming object must implement the **ExecutionPlan** trait. A trait is the Rust equivalent of an interface in other programming languages, like Java, or an abstract class in C++. A trait defines a set of methods that a type must implement, without specifying how those methods are implemented, allowing different types to share common behavior. This trait features the all-important **execute()** function, which functions as an initialization function for the operator. This also allows the operator to call the **execute()** of its child relations, propagating the start of the query execution through the tree structure. When creating a custom operator, the role of the execute function is to create the stream of data that will be used to pull it in its parent operator. As such, when we propagate the *execute()* throughout the query tree, parent operators will receive the relevant data streams from their child operators, allowing them to poll for RecordBatches. Other functions from this trait include *getters* for information like the operator name and schema, and also functions to create new instances of the operator.

The simplest example of the *execute()* function is the one found in the **filter** operator. The only thing that happens is the creation of a metrics object to track and report performance and resource usage during query execution, alongside the *execute()* function call on its child operator. Afterwards, it creates a *FilterExecStream* object that can be used to pull data from the operator itself, meant to go to its parent operator. A more advanced implementation of the execute function can be found in the **hash_join** operator. For this operator, we require a full materialization of the build side and the creation of the hash table. After creating the build table necessary for the operator, the *execute()* function of the probe side operator is called, and the **HashJoinStream** object is created.

As mentioned in the previous section 3.3.2, operator output is generated via *streams*. A stream object in DataFusion is implemented via the **RecordBatchStream** trait, functioning like a regular Rust stream with a corresponding schema that specifies the schema of RecordBatches created by this stream. Each operator implements its own stream, where it pulls a RecordBatch from its child relation, performs its operation on the data, and indicates it is ready to pass data onto its parent operator. To return to the filter operator for a simple example, the file where the implementation of the operator is located also houses the implementation of its stream, the *FilterExecStream*. The following stream explanation is visualised in Figure 3.7. A stream defines an item that passes through the stream, which is a *RecordBatch* in this case and a **poll_next()** function. This function can be thought of as the *next()* function for the Volcano operator model. It should first poll its child operator for a *RecordBatch*, the filter operator then filters the *RecordBatch* using a helper function called *filter_and_project()*. Afterwards, it will indicate that it is ready to send this batch further by setting its *poll* parameter to *Poll::Ready*, the *poll_next()* function will then return the processed *RecordBatch*.

For a more advanced example, we again turn to the *hash_join* operator. In this operator, its *next()* is implemented as a finite state machine, with states indicating what the operator is currently doing with its input data as visible in Figure 3.8. As such, this Figure 3.8 varies from Figure 3.7 in that in shows the inner workings of the operator, without taking the streaming states into account. The operator starts in the *WaitBuildSide* state, which indicates that it is still busy with fully materializing the build state. This materialization is necessary to create

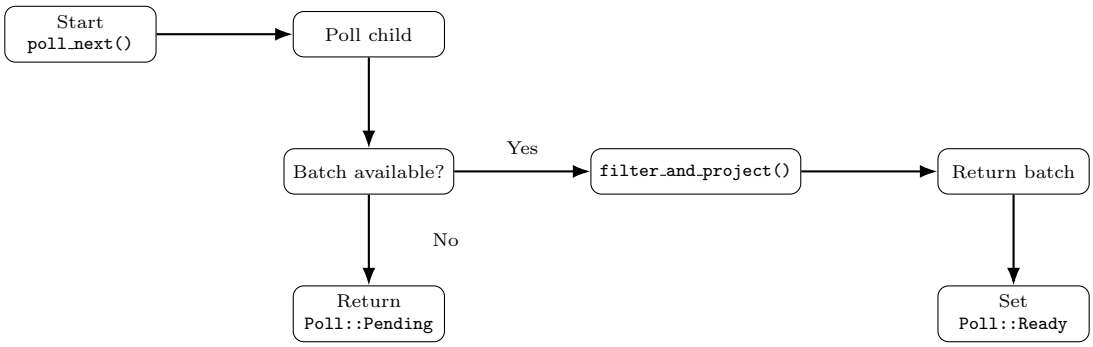


Figure 3.7: flow inside the filter operator

the hash table needed for the algorithm to function. After the hash table has been built, the stream transitions to the *FetchProbeBatch state*. A call to the *poll_next()* function in this state leads to a call to the *poll_next()* of the probe side child operator. If this child yields a new *RecordBatch*, the state of the hash join operator transitions to the *ProcessProbeBatch* state, where the fetched batch is joined with the build side and an output batch is produced. Should the child operator not yield a batch, the stream transitions to the *ExhaustedProbeSide* phase, during this phase, the stream processes unmatched build side rows. This could be necessary in an outer join operation, for example, where it is necessary to also output all build-side rows that did not find a match on the probe side. When the stream reaches the *Complete* phase, it indicates that it is ready by returning Poll::Ready(None) whenever *poll_next()* is called. This information was gained from the DataFusion source code [1].

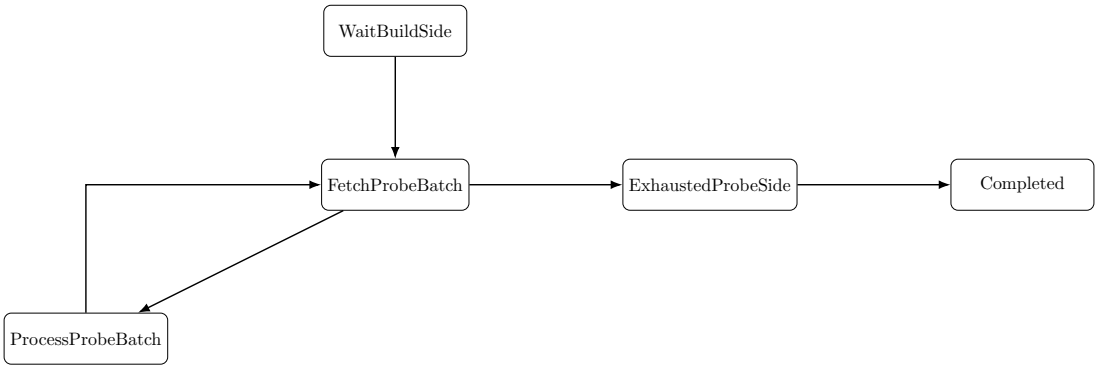


Figure 3.8: State machine implementation of DataFusion's hash join poll_next

3.4.2 The exchange operator in DataFusion

"Since DataFusion implements the Volcano model, parallelism can be enabled by implementing an exchange operator. This operator is called the **repartition** operator in DataFusion's source code and workflow. This subsection will explain the flow of this operator within DataFusion since the exchange operator implemented during this thesis was modeled after it. This subsection was written using DataFusion's source code [1] as a base.

To start explaining how this operator works, a description of the data structure used within the operator follows. Since this operator will receive data from multiple child operator streams and needs to support multiple parent operator streams, a data structure called a **distributor channel** is used. This structure can be thought of as a multi/demultiplexer, turning an amount of inputs into a possibly different amount of outputs. A visualisation is shown in Figure 3.9.

The operator workflow starts with its *execute()* function, which acts as the *open()* function for

the Volcano operator. In this initialization function, a given number of workers will be spawned, each responsible for driving a partition/stream of the child operator to completion. As part of its task, a worker performs a full materialization of the relevant child stream. It will poll the corresponding child stream batch-per-batch for each incoming RecordBatch and will then look at each incoming row within the batch separately. For each row, the worker looks at the correct attribute(s) to decide which output partition/stream it has to be sent to. This process is comparable to the first step in figure 3.5, and is done via a keyed hash function from the *ahash* Rust crate. An important note to make when hashing in this context, since a **random state** is used within the hashing algorithm, it is imperative that this random state is the same for each operator/stream present within the query tree. As explained in Subsection 3.2.2, every operator needs to partition its data using the same hash function in order to have a correctly functioning partitioning scheme. If, for example, a join operator has two child operators that hash the same values to different partitions, the operator would not be able to correctly join the data and would then provide incorrect results. DataFusion simply seeds the random state with a static value to circumvent this.

For each outgoing partition, a new RecordBatch will be created containing the rows headed for this partition. It is not possible that this does not fit into a RecordBatch since the operator splits one batch into one or more batches. Each outgoing batch will then be sent to the correct output buffer to wait for the corresponding parent stream to retrieve it.

Since the *open()* function contains all of the interaction needed with the child operators, alongside the preparation of the output data. This operator's Volcano *next()* function simply retrieves a RecordBatch from the corresponding output buffer whenever it is called. The only extra work it performs is checking whether or not its corresponding output buffer has been exhausted and incrementing the number of exhausted output buffers by one whenever this is the case.

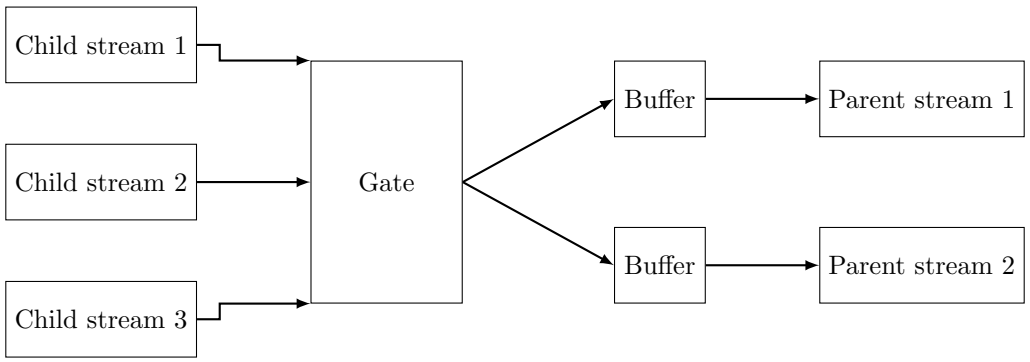


Figure 3.9: Distributor channel diagram

The *exchange* operator is automatically inserted into query trees whenever DataFusion compiles an SQL query. It is possible to print out the physical plans that execute the given query to verify this. Suppose that we execute a join between `table1(a,b)` and `table2(a,b)` on column `a`, hereby selecting only the rows where `table1.b > 10` and `table2.b < 10`. This is done using the following SQL query:

```
SELECT * FROM table1 JOIN table2 ON table1.a = table2.a WHERE table1.b > 10 AND
table2.b < 10.
```

Figure 3.10 shows the constructed logical plan for this query. It shows the filters found in the WHERE clause alongside the table reads and joining of the two used data sources. At the top of the logical plan is a projection that does not remove any attributes. The query compiler can derive various physical plans from this initial logical representation. Some of these optimizations occur at the logical plan level, like pushing the filter operations down towards the table scans. This early filtering reduces the amount of intermediate results, leading to a more efficient join operation. To facilitate parallelism, the *exchange* operator also needs to be inserted into the

plan, preferably at multiple locations.

Figure 3.11 displays two different physical plans for the same query. Figure 3.11a is created when parallelism is disabled in DataFusion; this plan lacks the *exchange* operator. There are two main differences between the initial unoptimized logical plan and the physical plans in this case. First off, the query compiler has decided to split the filter operator into two filter operators and to push them down to the table scan (ParquetExec), an optimization done in the logical plan stage. Second is that a new operator named *CoalesceBatchesExec* is added; this operator is linked to the size of RecordBatches, which has an impact on the runtime as explained in 3.3.2. Since the filter operator decreases the size of the batches that leave it, the query compiler places a BatchCoalescer after it to get the average RecordBatch size back up to a larger amount.

Figure 3.11b is created when running the query normally, allowing DataFusion to make a parallel runtime environment. The **RepartitionExec** operator is the *exchange* operator implementation, it supports multiple partitioning schemes as visible in the plan. The *RepartitionExec* operator comes in pairs in this query tree. The first of the two uses a round-robin partitioning scheme. This means that data will get shuffled across multiple partitions equally. The parameters of this operator also contain the number of input partitions that enter this operator; this is set to one, meaning that the reading of the entire file and its filtering all happen in a single stream/task. The parameter of the round-robin partitioning itself is set to four, which means that the data will be split evenly across four different partitions. As such, this repartitioner will split one stream of data into four different streams of roughly equal size.

An important sidenote to make is that the repartition operator does not always come in pairs. If the previous operators are already working in multiple streams, there is obviously no need for a round-robin partitioner. There is also the case when an input file is sufficiently large, and the query compiler will decide to read this in parallel into multiple streams. In this case there is also no need for the round-robin repartitioner. The hashing *RepartitionExec* has four input partitions, these are the four different streams that the round-robin repartitioner created. This repartition operator will map four input streams to four output streams, correctly partitioning them based on the join key of the following join operator. As mentioned earlier, this operator will split up *RecordBatches* into multiple smaller batches; as such, the query compiler places another *BatchCoalescer* above the repartition operators to increase the average batch size back up to the standard 8192. The **HashJoinExec** operator is set to *partitioned* in the parallel case. This means that it has to take things into account, like the number of output partitions each of its children has; if, for example, the left child has three output streams while the right child has four, it would abort the query execution. In the single-threaded version, there is no need for checks like these, and as such, it operates in a different mode. When the HashJoinExec operator is set to a non-partitioned mode, it is also possible for the build side relation to be read in multiple streams while the probe side only contains one stream. In this case, the operator will automatically coalesce all partitions of the build side into a single stream.

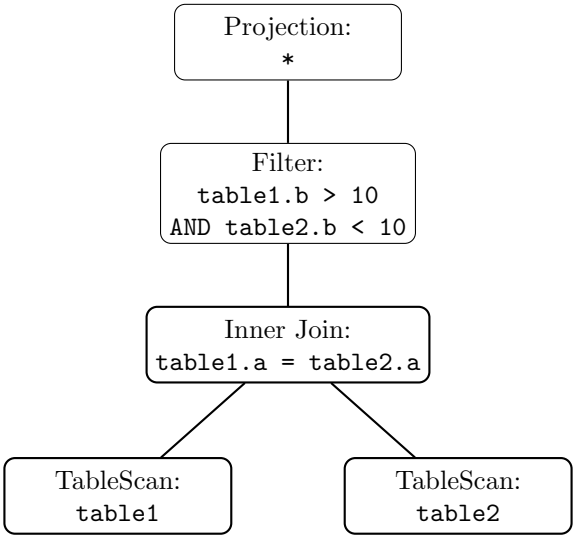
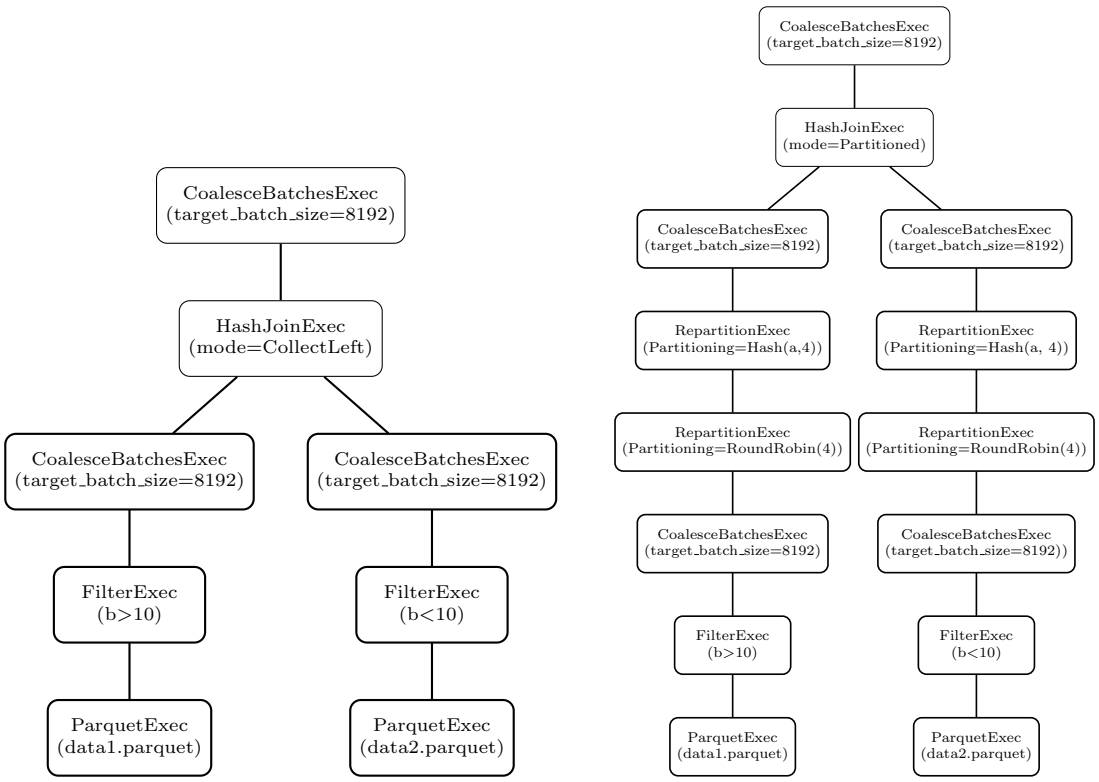


Figure 3.10: Logical plan tree for a join with filter



(a) Serial physical execution plan **(b)** Parallel physical execution plan

Figure 3.11: Physical plans with parallelism disabled and enabled

Chapter 4

Integrating parallellism

This chapter contains an outline of the implementation of Shredded Yannakakis made by [4] in DataFusion. It then describes how parallelism was added to this implementation, the issues that arose, and the solutions for those issues.

4.1 Overview of the Existing Implementation

This section provides a detailed explanation of the Rust implementation of Shredded Yannakakis presented in [4]. It describes the core data structures and the various operators' functionality. The subsequent section will then outline the steps required to introduce parallelism into this implementation correctly.

4.1.1 Data structures

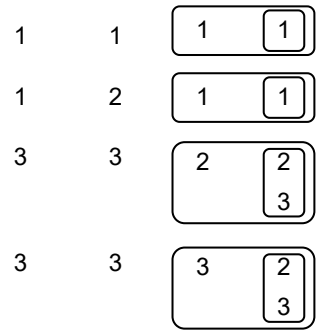
At the core of the implementation lies the **NestedBatch** data structure. This object is described as *a batch of records that have at least one nested column.* This is comparable to the RecordBatch present in Apache DataFusion, since it manages a collection of records. A NestedBatch contains both flat and nested data, with the flat data being held as-is and the nested data being split into several arrays using a shredded representation (see subsection 2.3.3). These arrays represent the different parts of the shredded approach, like the head-of-list and next vectors. The nested data is also recursive, allowing for flexibility between different levels of nesting.

Figure 4.1 shows the most intuitive way I found to visualize a NestedBatch. Figure 4.1a is an example of a nested table with Figure 4.1b being the same data, visualised as they are presented in a NestedBatch. The flat attributes are on the left-hand side of both figures, since the nested batch stores these as-is. The flat attributes have arrows pointing to their corresponding nested values. The arrows themselves correspond to the head-of-list vector present in this level of nesting. Each rectangle indicates one level of nesting, and the dotted arrows correspond to the next vector for this level of nesting. When no dotted line is present, the *next* value is zero, indicating there is no next element.

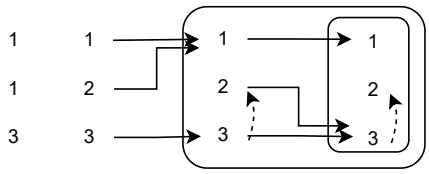
With this data structure, tuples with nested data can be collected as records in NestedBatches, ready to be used within the implementation of the nested semijoin algebra.

4.1.2 Operators

The two main operators required for joining two input relations are called **multisemijoin** and **group-by**. The multisemijoin operator differs slightly from the nested semijoin as explained in Section 2.3, while the group-by operator is the same.



(a) Conventional nested representation



(b) Example of shredded representation in code

Figure 4.1: Comparison between shredded and nested representations

The multisemijoin operator was implemented as a DataFusion operator with its corresponding stream. It takes a guard relation, which is a regular table read, alongside one or more child group-by operators. It will then perform the semijoin operation as presented in Figure 2.7, the guard relation would be relation S, while the group-by operation is the child relation. The multisemijoin has its own stream structure, with its own data type running through it. A **SemiJoinResultBatch** is the output of the multisemijoin operator and the input of the group-by operator. It can either be a flat RecordBatch or a nested NestedBatch; this is necessary since the multisemijoin is also used to introduce table scans to the 2NSA plans. A 2NSA plan is slightly different in practice as visible in Figure 4.2, since an extra semijoin is used as input to the lowest group-by. The output of the multisemijoin that reads relation T are flat RecordBatches, that enter the group-by $\gamma_{\{a\}}$ to be nested on attribute a.

The group-by operator, also implemented as a DataFusion operator with its corresponding stream, differs from the multisemijoin in fully materializing its input stream. A group-by operator will exhaust its child multisemijoin relation in its *execute()* function, collecting metrics like the number of input batches and rows as it does so. After this, it will group all batches using its specified group-by attribute. It will then return a **GroupedRel** object, similar to a NestedBatch but with specialized implementations depending on its content. The important difference between the multisemijoin and the group-by operators is that the multisemijoin has a corresponding stream structure, allowing it to function like a regular DataFusion operator via execute() and poll_next() function calls. In contrast, the group-by operator immediately returns the value of its computation.

4.2 Adding parallelism to the existing implementation

This section will explain the idea behind the parallel implementation created for the existing shredded Yannakakis implementation [4]. It will review the architectural decisions and describe the strategies used to facilitate parallelism.

4.2.1 The exchange operator for shredded Yannakakis

To add parallelism to Shredded Yannakakis, the overarching strategy was to introduce an exchange operator that works with nested relational algebra. As explained in Subsection 3.2.2, the exchange operator is a meta-operator that can be inserted into existing query plans. Once

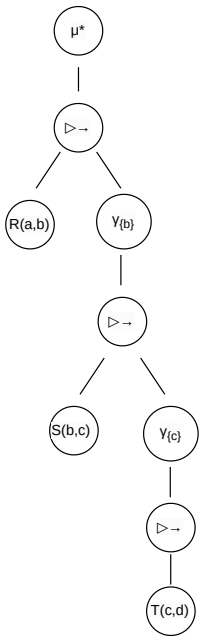


Figure 4.2: A realistic 2-phase NSA plan

in place, it spins up new threads to drive the execution of different subtrees to completion in parallel. Apache DataFusion already includes an exchange operator for standard RecordBatch data, but adapting it to support nested batches required some adjustments to ensure the query results remained correct. We must now focus on creating an exchange operator that can be inserted into 2NSA plans. The original operator in DataFusion will be referred to as the **repartition** operator, while the newly implemented operator designed to handle nested data will be referred to as the **exchange** operator.

When thinking of ways to approach this problem, it quickly became clear that implementing the exchange operator would be different for either the multisemijoin or the group-by operators. Since it goes on top of an existing operator, and both operators produce their output in other ways with different resulting data objects, ensuring correctness for both would mean that two different operators had to be made. Following a manual calculation of what should happen for the data to flow through the operators concurrently, it became apparent that implementing the exchange operator for either operator would prove sufficient. Doing this for the multisemijoin seemed to be the most intuitive approach. The manual calculation also made clear how the partitioning would go and why this would be correct.

When implementing the exchange operator above multisemijoin operators using the general idea explained in Subsection 3.2.2 and shown in Figure 3.5, it's crucial to partition the data correctly as it passes through the operator to ensure correctness. This involves selecting an attribute on which to partition the data, where the value of that attribute determines the outgoing partition for each tuple. This attribute, referred to as the **partition key**, must align with the grouping attribute used by the group-by operator higher up in the query plan to maintain correctness and avoid missing join tuples. When partitioning in this manner, we ensure we obtain a reproducible scheme. This is important because we are performing the semijoin on two different relations together; as such, we must ensure that equal join key values end up in the same partitions.

Example 4.2.1. To illustrate how this could go wrong, using Figure 4.2, data from table T and table S are joined in the second multisemijoin. If, for example, numerical data is split into

two partitions, where data coming from T is sent to partition 0 if its partition key is even and sent to partition 1 if it is odd. Data read from table S now also needs to follow this partitioning scheme, for if it is not followed, tuples that should join will not be present in the same partition and as such won't be recognized as joining tuples. This leads to the same issue that arose in Figure 3.5.

Since there is no repartitioning after a group-by operation, its output is still partitioned as it was when leaving the multisemijoin exchange operator. We also know that the group-by operator output will have one flat attribute. Consequently, this is the attribute on which the data is partitioned and will be present in the guard stream of the higher multisemijoin. All that needs to happen now is to partition the guard stream on this join key using a regular DataFusion repartition operator. Since we use different operators to partition the data, both partitioning schemes must utilize the same hashing function. Otherwise, equal values for the join key might still end up in different partitions. If we are able to align the hashing that occurs after a multisemijoin with the hashing that occurs when reading the guard relation, we have ensured correct partitioning.

Example 4.2.2. A walkthrough of the partitioning needed in Figure 4.3 follows. Starting with the lowest table scan for table T, it would need to be partitioned on its Z attribute, since this is what its following group-by operator will group the relation on. Leaving the group-by $\gamma_{\{Z\}}$, the output would still be partitioned on the Z attribute, since the group-by will perform its operation on the different partitions separately without any data shuffling. This would mean that the table scan for table S which is the guard relation for the multisemijoin also needs to be partitioned on its Z attribute; this way, equal values for Z will always be present in the same partition, and as such, the multisemijoin operator can join the two relations correctly. Leaving this multisemijoin operator, the output is still partitioned on the Z attribute. The problem now is that the next join operation will take place using the Y attribute as the join key, as such, there is a need for a repartitioning of the data, now using Y as the partition key.

The exchange operator must now be implemented to repartition the nested data structure to achieve this partitioning. It must look at every output tuple of its underlying multisemijoin and send it to the correct output streams. Just like the regular exchange operator, it will function like a distributor channel as seen in Figure 3.9. Having repartitioned the data on attribute Y in between the multisemijoin and group-by $\gamma_{\{Y\}}$, the group-by operator will again group the incoming tuples. The table scan of guard relation R will then be partitioned on its Y attribute, before being sent to the multisemijoin that will join the two inputs. It now becomes clear that every time the join key changes along the query plan, a repartitioning step must be introduced to ensure the data is correctly aligned for the next operator.

Since the exchange operator promises to introduce parallelism into query runtimes without changing existing operators, the goal of the implementation was to make changes only inside the newly implemented operator. The implemented operator largely succeeds in doing so; however, there was one change that had to be made to the group-by operator. The group-by operator contains an array of child multisemijoin streams, which it uses to materialize their subresults. This array had to be converted to a two-dimensional array, where instead of only containing the children, it contained each partition of each child. Since the original implementation only worked with a single partition, the operator could not work over parallel partitions without this change.

4.2.2 Shuffling nested data

The main hurdle in implementing the exchange operator is implementing how data gets shuffled across partitions. To look at every tuple and decide where to send it, a full materialization of the subresults is necessary. Each tuple will be looked at separately while calculating a hash value for its partition key. Since the partition key is always a flat attribute in the nested batch case, we can extract this column to calculate the hash values, as happens for regular RecordBatches. In

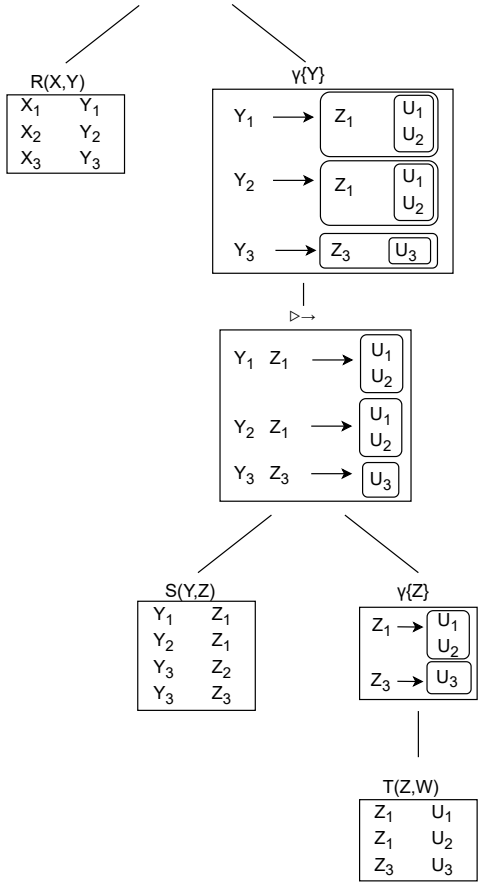


Figure 4.3: Possible Shredded Yannakakis runtime

the flat RecordBatch case, a RecordBatch gets split into smaller RecordBatches, one for each output channel. After the whole incoming RecordBatch has been processed, each outgoing batch will be pushed into its corresponding output channel. During the implementation of the operator, issues arose with shuffling the nested data columns. This subsection will explain the problems that arose, while Subsection 4.2.3 will explain the solution to this problem.

The group-by operator, as implemented, fully materializes its input data before performing its operation. Since the data is fully materialized, it contains the whole subresult and performs its operation on it in its entirety. This leads to any group-by operator always returning one groupedRel (which can be considered as a RecordBatch), which contains its nested data in one NestedColumn alongside its flat attributes. The multisemijoin operator will take this single resulting groupedRel batch and join it with its guard relation, resulting in multiple batches leaving the multisemijoin operator. In practice, when creating the outgoing groupedRel in the group-by operator, the implementation will simply take the first batch from the materialized subresult and use its nested data since it knows that the nested data is the same in each batch present in the group-by. An important note is that the nested data is equal in all batches; the head-of-list vector that links flat attributes to nested attributes may differ for each batch. To illustrate this by using Figure 4.1b, the numbers inside the nested columns are the same $1, 2$ and 3, the next vector is the same, visualized by the dotted arrows, but the arrows pointing from the flat attributes to the nested attributes are different.

When working with multiple streams concurrently, each stream of data/partition that flows through the group-by operator may result in a different groupedRel with different data. A problem arises when data shuffling occurs, when, for example, we send a nested batch from partition 0 to partition 1, its nested columns will contain data created in partition 0. The other batches present in partition 1 will contain the data that originates from this partition, meaning that when we simply work with the nested data of any given batch during the group-by operator, this data is not necessarily representative of all batches present in the partition. This leads to an incorrect query execution, possibly creating index out of bounds exceptions since the head-of-list vector remains unchanged, and crashing the query runtime.

To illustrate this problem, we turn to Figure 4.4. Each separate part of the image shows a batch that passes through the operators; the batches themselves come from inside the exchange operator. Starting with batch *a*, since it only contains one row, the repartitioning in the exchange operator will calculate the hash value for the partition key and send that row (now batch *c*) to the correct output partition, which is partition 0 in this case. Batch *b* has two rows, so hashes will be calculated for each row. Because of the values for the partition key hashing to different buckets, the batch will need to be split into two separate batches. Batch *d* will be sent to partition 0, meaning that it has moved partitions, while batch *e* stays in partition 1.

The problem now becomes clear when looking at batches *f* and *g*. They are what becomes of the batches *c,d* and *e* after passing through a group-by and a multisemijoin operation. Since partition 0 now has two batches with differing nested column data, $\{1, 3\}$ and $\{3\}$, and the group-by operator simply looks at the first batch it retrieves to obtain the nested column data. We obtain a **race condition**, where depending on which partition was the first to push its data through the exchange operator, the results of the following group-by operator change. It is clear that both batches *f* and *g* do not contain the same results, neither of them being correct. This means that the output of the query execution would be incorrect. When running this scenario, though, the pointer in batch *g* that points to the second value of the deepest nested column points outside the array, ending the execution with an error.

We can note that this problem does not take place in partition 1, since there is only one instance of nested column data. This would mean that if data shuffling never takes place and data never changes partitions, this issue does not take place, and the runtime can correctly return. This would practically be the case when all joins in the query take place on the same attribute. In that case, the input relations always get partitioned into the different streams and keep on being processed within the same streams, avoiding the need for reshuffling. This is not always the case, obviously, so a solution for this problem had to be devised.

4.2.3 Combining nested data

The solution implemented for the problem outlined in the previous Subsection 4.2.2 is to create a new structure, responsible for combining multiple nested data structures into one. By combining the nested data of each partition into one overarching NestedColumn, we again ensure that the nested data is equal throughout all batches passing through the group-by operators, leading to correct query outputs.

Practically, a **NestedCombiner** object was added to the functionality of the nested exchange operator. It requires one batch from each incoming partition and extracts its nested columns. Afterwards, it will concatenate all nested columns from each partition into one overarching nested column, keeping an array of offsets so the head-of-list and next pointers can be incremented by the corresponding offset to maintain correctness. This nested column will then be inserted into the batches that are headed to the output channels of the exchange operator.

The fact that the exchange operator now requires one batch from each partition before it can start producing its output does bring some inefficiency with it. Each exchange operator now requires a synchronization point where all partitions must have made enough progress to send at least one batch to their input channel of the relevant exchange operator. To implement this,

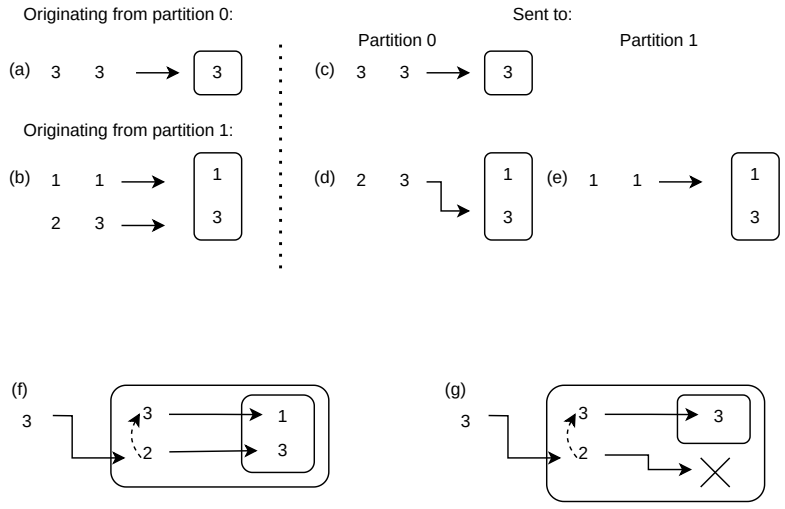


Figure 4.4: Batches passing through the exchange operator

a **barrier** from the Tokio runtime was used. The barrier knows the number of partitions that must call its *wait()* function and will halt any progress that partitions want to make as long as any still have to execute the function call. Once all partitions have progressed up to the barrier, one partition will perform the work needed to combine the nested columns present in the combiner object, while the others wait behind a second barrier until this work is finished. Afterwards, the different partitions all get a copy of the large combined nested column alongside the offsets that the combiner calculated, and start partitioning the batches to send them to their corresponding output partitions. Now, when splitting a batch into multiple smaller batches, the offset corresponding to the partition the batch originated from is added to the head-of-list vector, and the data is swapped with the data that was created in the nested combiner. This ensures that flattening the batch still results in the same flat data that the batch originally would flatten to, while also removing the issue of incorrect nested columns after shuffling the data.

Example 4.2.3. We now turn to Figure 4.5 for a concrete example of the combiner in action. It features the same input relations and query as in Figure 4.4 but now with the nested combiner implemented. The batches that get sent to the output channels now feature different nested column data than they did in 4.4, since the data has been created by the nested combiner. The operator will wait for the nested data from both partitions to be present before calling the nested combiner to combine the two. It will then combine the two arrays $\{3\}$ and $\{1, 3\}$ together into array $\{3, 1, 3\}$, while also changing the weights of the nested column to the correct values. After the combiner is done with its operation, batches *c*, *d*, and *e* can be made using this data. The head of list pointers then get summed up with the relevant offsets. Batches made in partition zero have an offset of 0, while batches made in partition one will have an offset of the length of the nested data present in partition zero. Since the nested data is $\{3\}$ in this case, the offset will be one. When working with more than two partitions, all previous nested data sizes must be summed by taking the previous offset and adding the length of the newly appended nested data to it. The head-of-list in batch *c* now points to the first value in the nested data. In contrast, batch *d*, whose tuple previously had a head-of-list pointer value of 2, now has a value of 3.

The ripple effect of the newly changed nested data can be seen in batch *f*. This batch is an input batch for an exchange operator higher up in the query plan. It has the concatenated nested data, and as such, the head-of-list pointers can no longer point outside of the array. Alongside

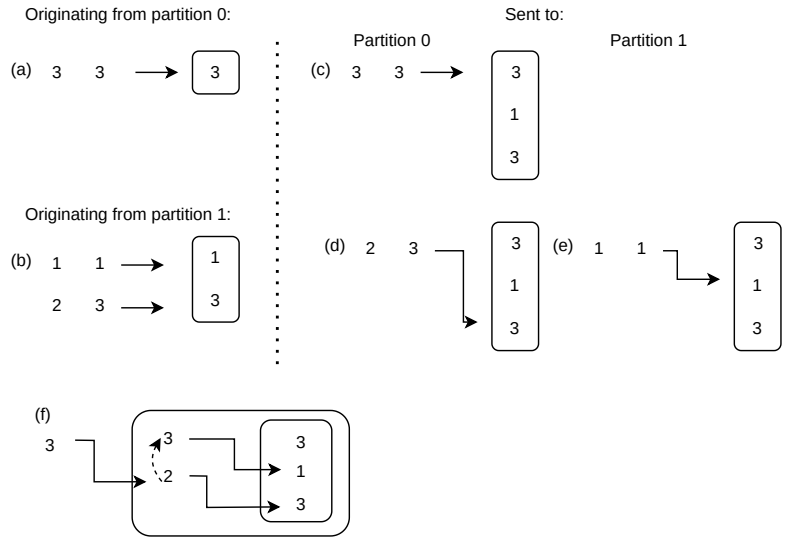


Figure 4.5: Batches passing through the exchange operator with the combiner active

correcting the output, this approach also fixes the race condition present without it, since now each partition has the same values for the nested data.

4.2.4 Editing existing plans

The original implementation uses a JSON file format to pass query plan data to DataFusion. The file consists of a tree structure that denotes the different operators and their parameters. To give an example of this, the parameters of a multisemijoin include its join keys, its guard relation, which is usually a table scan operator, and its child relations. The child relations are a variable number of group-by operators that are used as input to the multisemijoin operator; this can be either zero, one, or multiple operators.

To add the exchange operator to this structure, extra nodes were added to the conversion logic from JSON to the implemented operators. First off, the regular DataFusion exchange operator was added. Its parameters include the number of output partitions and the partitioning scheme (hashing or round-robin). This operator was necessary to partition the RecordBatches read during the table scan operation into multiple streams. Secondly, two new attributes were added to the multisemijoin node. The first attribute **partitioned** indicates whether or not a multisemijoin exchange operator is to be added above this multisemijoin node or not and the second denotes the partitioning scheme of this exchange operator.

In Figure 4.6, three different query trees are visible. The tree itself displays a join between tables T and S, and the operators in the tree occur as they would in a runtime of the implementation. Tree (a) features the regular serial query plan. It first reads table T as the guard relation of its first multisemijoin, after which it will perform the multisemijoin operation, alongside the group-by operation. Now it will pass a second multisemijoin operation where table S is read as the guard relation. After the top multisemijoin, a flatten operator is used to transform the nested data into flat data. Query tree (b) features the same plan, now ready for parallelism. Above the two table scans, a DataFusion repartition operator is inserted with n output partitions. Above the two multisemijoin operators, an exchange operator was inserted, specific to the output of this operator. It automatically uses the same number of output partitions as the DataFusion exchange operator, since the partition count is automatically propagated. This plan now calculates its output across multiple workers. Each exchange and repartition operator spins up workers according to its input partitioning count; the repartition operator for table

scans only spins up one worker since the table scan produces its output in a single partition. Afterwards, this repartition operator will split that data into n partitions, allowing parallel data flow. As such, the exchange operator meant for the multisemijoin starts n workers that each pull batches from their respective partition in parallel.

Above the flatten operator, another standard DataFusion exchange operator was added with one output partition. It receives n input partitions from its child unnest node and coalesces them into 1 output partition. If this operator is not added to the query plan, the output made by the runtime will be partitioned. An example would be a query that retrieves the number of output types using $Count(*)$. Partitioned output would feature n outputs, corresponding to one number for each partition, that together sum up to the same number that the serial runtime would return. By repartitioning everything into a single partition, the counting aggregate can perform its operation on the entire output and return the same value as it would in the serial case.

As visible in tree (c), the exchange operator between the unnest and the top-most multisemijoin can be left out of the query tree while still retaining the same, correct output. This is because the exchange operator does nothing more than shuffle the data across partitions, alongside spinning up worker threads. The DataFusion repartition operator above the unnest will see n input partitions, and as such spin up n workers that take care of the unnest, top-most multisemijoin, and group-by operators. The extra data shuffling cost that this operator incurs does nothing to the correctness of the output and only increases the measured runtime.

Another possibility would be to keep the exchange operator for the highest multisemijoin operator, but to have it use round-robin partitioning instead of hash partitioning. This greatly decreases the complexity of the work that this operator performs, since there will be no need for calculating hashes and checking all tuples in a RecordBatch separately. The upside of this is that we perform load-balancing before sending batches to the unnest operator. If the hash partitioning scheme gives us a skewed distribution of data, the unnest operator would have more work in certain partitions and less in others, increasing runtime when compared to a more balanced workload. In the ideal case, the exchange operator would take an approximation of the amount of work that the unnest operator performs for each outgoing batch and send them accordingly, which would allow even better load balancing.

Another potential upside of keeping the exchange operator underneath the unnest operator is that this will then allow a higher degree of vertical parallelism. This exchange operator will allow the unnest operator that is being run by its own workers to unnest batches concurrently with its child multisemijoin that creates its input.

4.3 Type(s) of implemented parallelism

As explained in Subsection 3.2.2, we distinguish between different types of parallelism, and the exchange operator is designed to support multiple forms of it simultaneously. Starting with vertical parallelism, the implemented operator theoretically supports the concurrent evaluation of multiple operators. In practice, however, this is not entirely achieved in the implementation of the nested semijoin algebra. The degree of vertical parallelism is limited by the fact that the group-by operator requires full materialization of its input before it can begin producing output. As a result, operators higher in the query tree cannot receive any input batches until the preceding group-by operator has received and processed all necessary data. This effectively limits vertical parallelism to groups consisting of a multisemijoin operator, a group-by operator, and an exchange operator.

For horizontal parallelism, inter-operator parallelism is clearly enabled. The implemented exchange operator partitions the data into multiple streams and allows those streams to be processed concurrently. However, the materialization required by the group-by operator also limits bushy parallelism, as it prevents different subtrees of the query plan from being evaluated concurrently.

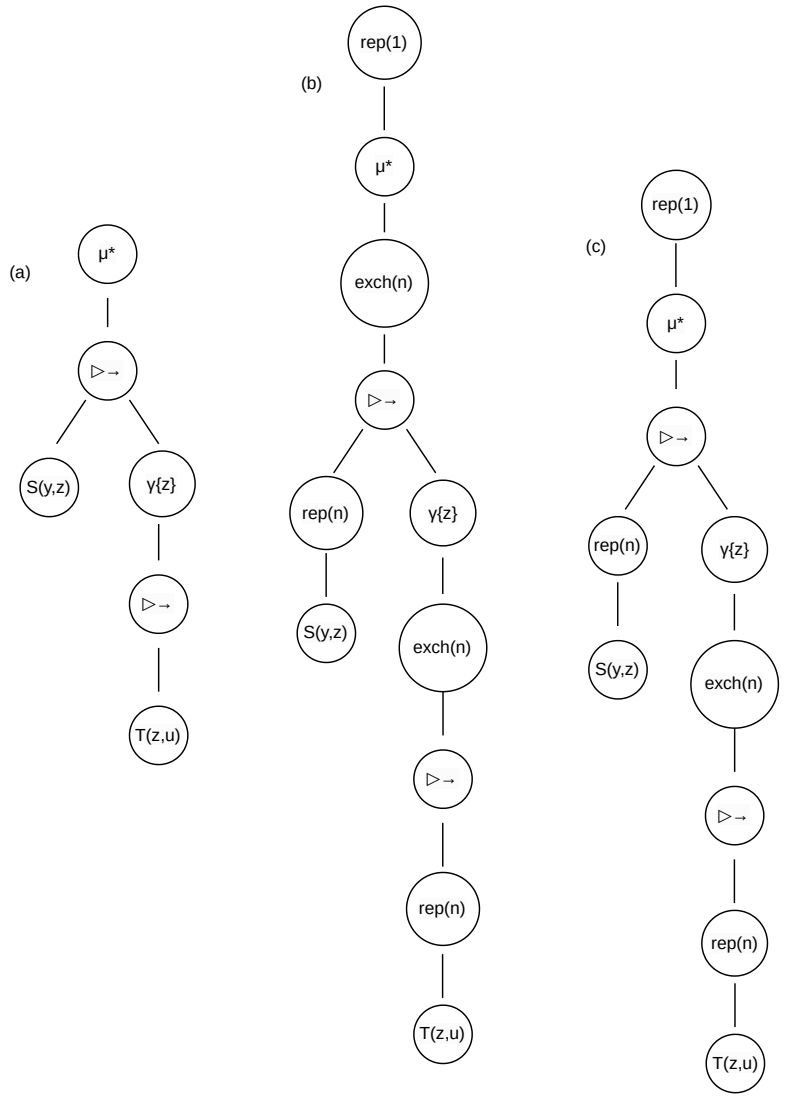


Figure 4.6: Different equivalent plans, serial and parallel

As practical proof, a query was evaluated while timestamping any batches that passed through any operator. It clearly shows that a group-by operator acts as a boundary between two multi-semijoin operators evaluating their batches. As such, vertical and bushy horizontal parallelism is practically limited.

4.4 Shortcoming of the implementation

The implementation of the exchange operator was created with the goal of being able to support any 2-phase NSA plan. There is, however, one class of queries that can be evaluated with the serial implementation that cannot be evaluated in parallel. This is due to the fact that the multisemijoin operator supports multiple group-by children relations. Each child relation will be joined with the guard relation of the multisemijoin. This, however, leads to the possibility that several attributes of the guard relation can be used as join keys. In this case, since the

guard relation is always partitioned on its join key attribute using an exchange operator, the guard relation would have to be partitioned on multiple join keys at the same time to ensure correct partitioning. A new exchange operator to facilitate this behaviour was not created throughout the thesis, so queries that exhibit this were filtered out to maintain correctness and validity during testing. What follows is how an exchange operator could be implemented to facilitate this.

We now need to utilize the standard DataFusion hash function, but hash our values in such a way that we get a multi-dimensional hashing scheme. Now, instead of assigning each tuple to one bucket, we assign each tuple to N buckets, where N corresponds to the number of attributes that will be used as a join key within the multisemijoin operator. We hash each tuple N times using the standard hash function, once for each join key, and place it in the bucket corresponding to the join key. To put it formally, take P partitions, let K be the join key attributes used in the multisemijoin $K = k_1, k_2, ..., k_n$. Now, for each tuple, we compute $hash(T(k_1))$, $hash(T(k_2))$, ..., $hash(T(k_n))$ with $T(k_n)$ denoting the value for the k_n attribute of the tuple. We then map each hash value to a partition via the modulo operation. Tuples can now appear in multiple partitions, allowing them to be joined correctly since they will now be present in the correct partitions.

Chapter 5

Experimental Evaluation

This chapter will explain concepts used to evaluate parallelism and go over the results of the parallelism added to the shredded Yannakakis implementation. Different aspects such as total runtime, independent operator runtime, scalability via efficiency and speedup will be covered. Comparisons will be made to the original serial runtime, with all measurements coming from the same machine.

5.1 Concepts of parallel evaluation

When evaluating parallel implementations, the **degree of parallelism (DOP)** plays a critical role in determining performance. This metric specifies how many operations can be executed concurrently and typically corresponds to the number of worker threads or tasks active at any given time. In the implemented system, the exchange operator introduces parallelism by spawning multiple workers, each responsible for processing a portion of the input. These workers are scheduled and managed by the **Tokio runtime** [23], which assigns tasks to available processor cores. As a result, the DOP directly influences how effectively system resources, particularly CPU cores, are utilized during query execution.

To compare the runtime duration of varying degrees of parallelism, **speedup** and **efficiency** are used as metrics. Speedup is defined as the ratio of time required by the sequential algorithm to solve a problem, denoted by $T(1)$, to the time required by the parallel algorithm using p processors to solve the same problem $T(p)$. As such, speedup $S(p)$ for p processors is defined using the following formula as stated in [18].

$$S(p) := \frac{T(1)}{T(p)} \tag{5.1}$$

There is a concept called the **ideal speedup**, which serves as a theoretical upper bound to the performance gain achievable through parallelism. In this case, the speedup is the same as the number of cores used for parallelism, thus scaling linearly. This means that if a task takes T_1 time to complete in the serial case, it should ideally take T_1/p time on p cores, resulting in speedup $S(p) = p$. In practice, due to factors such as load imbalance, synchronization delays, and function call overhead, the measured speedup often falls short of this ideal speedup. Nonetheless, the ideal speedup can be seen as a curve that is useful as a benchmark for evaluating the efficiency of parallel implementations. Figure 5.1 shows the ideal speedup curve as a linear function of the number of cores used.

Speedup is also limited by a phenomenon known as **Amdahl's law**, alongside overhead such as thread scheduling and the additional overhead brought by the fact that parallel implementations usually feature more/longer code than their serial counterparts. Amdahl's law states that the

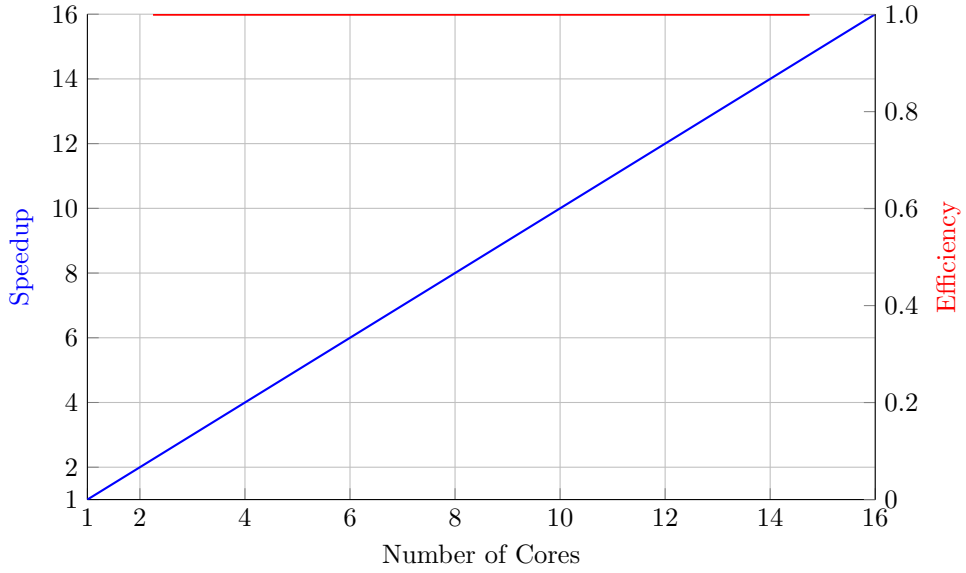


Figure 5.1: Ideal speedup and efficiency as a function of the number of cores.

speedup of a parallel algorithm is limited by the number of operations it must still perform sequentially. This means that the theoretical maximum speedup of a program as a result of parallelization is based on the proportion of the program that can be/is parallelized. It alters the speedup formula to formula 5.2, where f is now the fraction of the workload that is parallelized, meaning that $1 - f$ denotes the fraction of the workload that is serial. This formula was also extracted from [18]:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}} \tag{5.2}$$

A second useful concept for evaluating parallelism is **efficiency**. Since an ideal system with p processors has a speedup equal to p, and this is not the case in practice, a processor cannot use 100% of its time for the computation. The processor also has tasks other than pure computation such as scheduling threads and interprocess communication. The formula for efficiency in Equation 5.3 can hence be interpreted as a measure of the percentage of time the processor is utilized to effectively perform "useful" computations. In the ideal case, efficiency equals one, but in practice, it is between zero and one, since we know that the ideal speedup is not usually feasible. Figure 5.1 also displays the ideal efficiency for different degrees of parallelism [18].

$$E(p) := \frac{T(1)/p}{T(p)} \tag{5.3}$$

Two different types of parallelism can be differentiated from each other. When talking about **strong scaling**, the problem size is fixed and the speedup and efficiency are evaluated for a different number of cores. This strong scaling is often associated with *Amdahl's law*. The opposite would then be **weak scaling**, where the problem size varies while the number of processing cores used stays fixed. Since *Amdahl's law* proposes a theoretical upper limit to speedup, Gustafson noted that this view does not reflect parallel systems in practice. **Gustafson's law** proposes that as more processors are added to the computation, the problem size increases to take advantage of this extra computational power. This law, as described in [11], provides a more optimistic approach to parallelism. The implementation of the thesis considers strong

scaling, since it looks to improve the runtime of existing queries by allowing them to run their calculations in parallel.

5.2 Experimental setup

To evaluate the scalability of the implementation, queries need to be evaluated and timed. To this end, the original shredded Yannakakis implementation in DataFusion will be compared to the parallel implementation. This means that for the same query there will be query plans where the exchange operator is absent, which are compared to the query plans that do feature the exchange operator above multisemijoin operators to integrate parallelism. We compare both plans on their execution time, which is measured starting after the query plan has been generated, until the output is fully created. We always report an average of multiple runs, since a phenomenon called **warmup runs** was noticed throughout testing. When calling a query multiple times, the first execution is typically much slower than later runs, possibly due to initialization overhead. Whenever an average value was used, the number of repeated runs will be mentioned in its discussion.

The benchmark used throughout the analysis of the results is the STATS-CEB benchmark [12]. Due to the aforementioned limitations of the implemented exchange operator in Section 4.4, of the 143 queries used in [4], 125 queries remain that can be correctly evaluated by the implementation. These queries vary in the number of records that pass through the joins, as well as the number of joins they compute. The tests were run on my own laptop, featuring an AMD® Ryzen 7 5800HS processor with 16 cores and 16 Gigabytes of RAM.

In order to get more detailed insight into the total runtimes, metrics objects were implemented for the various operators. These allow us to measure specific aspects/subtasks of the operators to find possible bottlenecks and are implemented in the DataFusion query engine. Examples of metrics for the exchange operator are: the time it takes to pull the first batch from its child multisemijoin, the time spent waiting for all partitions to arrive, and the time spent combining the nested column data of the different partitions as described in Subsection 4.2.3. Using these metrics, we can also measure the idle time spent waiting at the barrier for synchronization between the different partitions.

Measurements are mostly taken with partition counts in exponents of two, meaning that we measure the time it takes for the serial implementation to evaluate a query, alongside one, two, four, eight, or sixteen partitions using the parallel implementation. We can then calculate the speedup and efficiency of these measurements, alongside looking at the more specific metrics. The calculations for speedup use Formula 5.1, and the calculations for efficiency use Formula 5.3.

5.3 Results for the STATS-CEB benchmark

This section discusses the results of the execution speed when evaluating the queries from the STATS-CEB benchmark [12]. It takes all 125 usable queries into account (see Subsection 4.4) and evaluates the approach in its entirety. The section that follows will then provide more in-depth looks at some queries from the benchmark. The queries themselves focus on join operations; they are designed to simulate analytical workloads by joining data from multiple sizeable input tables together. Most queries include filtering, and all queries end in a count aggregation. Measurements for this section are averaged out from ten runs.

5.3.1 Results without specifying core count

This subsection features results from measurements where the number of cores usable by the Tokio runtime was not specified, allowing it to use all 16 cores available. The only difference between the runtimes is the number of partitions that the data gets split into. This then leads

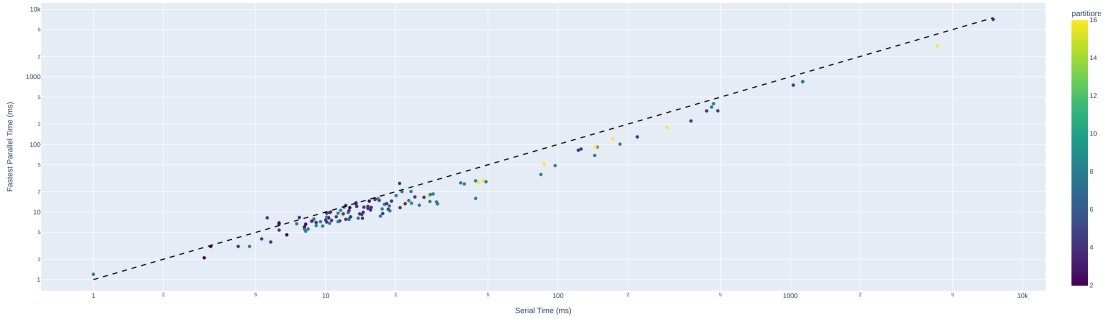


Figure 5.2: Log-log scatterplot for STATS-CEB benchmark

to a different number of workers for each operator, changing the degree of parallelism while keeping the amount of usable cores stable.

To evaluate the benchmark, scripts were created to add the new necessary information to the query plans originally created for the serial implementation. After filtering out the usable query plans from the benchmark, we are left with 125 of the original 143 queries. They were then tested for correctness on multiple partition counts, to ensure the validity of the implementation with the limitation set (Section 4.4). The benchmark queries vary greatly in complexity and input size. In particular, filter operators often filter out large subsets of the data, changing the amount of work that later operators need to perform. To illustrate, a query from the benchmark follows.

```
1 SELECT COUNT(*) FROM c, b
2 WHERE c.UserId = b.UserId AND c.Score=0
3       AND b.Date<='2014-09-11␣14:33:06'::timestamp;
```

By filtering the data in the WHERE clause, fewer records pass through the join operators, leading to less work for these operators and a smaller output size.

Queries are identified by a number, ranging from 1-146. To simplify the following explanations, an approximation can be made that as the identifier increases, the queries become more complex. Figure 5.2 shows a log-log scatterplot comparing the serial execution times to the parallel execution times. A logarithmic scale was chosen since the query execution times often vary greatly. Each dot corresponds to a specific query, with the colors of the dots corresponding to the partition count of the measurement. For this plot, the fastest measured partition count was chosen. If a dot were to be placed on the diagonal line of the plot, its serial and parallel runtimes would be equal. All dots underneath the line have gained a boost in performance from being run in parallel, while the dots above the line have slowed down due to parallelism (on all partition counts). The figure explained in this paragraph is reproduced in A.1 for improved readability.

Table 5.1 displays the number of times the given number of partitions was faster than the serial implementation. In the fastest queries column, it shows the number of times that number of partitions has the fastest average runtime among all parallel partition counts, indicating it is the most suitable partition count for those queries. Starting with one partition, this effectively increases the workload by adding the exchange operators to the query plan, without the upside of inter-operator parallelism. Theoretically, a performance gain from vertical parallelism is still possible, but as explained in Section 4.3, there is little vertical parallelism present in the implementation. There are still 33 queries that gain enough benefit from this vertical parallelism to gain a small boost in performance, though these gains are very small. A large jump is visible when comparing it to the two partition measurements. In this case, data is split into two streams so operators can process the different streams concurrently, allowing for inter-operator parallelism. A smaller jump is seen between two and four partitions, and the number starts to drop between four and eight. When looking at the number of times a certain number of

Partition Count	# Faster queries compared to Serial	# Fastest Queries
1 partition	33	4
2 partitions	106	9
4 partitions	115	49
8 partitions	113	51
16 partitions	100	12

Table 5.1: Metrics for different partition counts

Partition Count	Average(ms)	Median(ms)	Average without outliers(ms)
Serial	162.70	15.10	73.04
1 partition	215.54	17.00	75.37
2 partitions	155.11	13.30	58.28
4 partitions	132.57	11.80	51.92
8 partitions	136.36	11.90	51.82
16 partitions	134.01	12.90	53.08

Table 5.2: Table with average and mean for different partition counts

partitions was the fastest out of the parallel runs, eight partitions are often the fastest, followed closely by four. This metric indicates the number of times a dot on Figure 5.2 has the color of that partition. There is a large drop in fastest queries when comparing eight partitions to sixteen, this is due to the overhead that the increased number of partitions brings with them.

For a more specific breakdown of runtimes across the different partition counts, we turn to Table 5.2. This table presents the average and median execution times for ten runs for different partition counts. It is evident that the average execution times for one partition are higher than those of the serial implementation. However, starting from two partitions, the average time is lower, indicating improved performance for the majority of queries. Queries 126 and 135 stand out as outliers, showing significantly longer runtimes than the other queries. Query 126 displays a significant decrease in performance when run with one partition when compared to the serial runtime. Its inclusion skews the average runtime for the one-partition case, making it appear worse than it actually is. When these outliers are excluded, the average execution time for one partition drops considerably. The majority of queries already show performance gains starting from two partitions, indicating that parallel execution is generally effective even at this lower level of partitioning.

The lowest overall execution times are achieved with four partitions, suggesting this configuration offers the best balance between parallelism and overhead. There is, however, a case to be made that the eight partition measurements have a slightly lower average when omitting the two outliers. Fewer partitions (one or two) do not leverage parallelism as effectively, while higher partition counts (eight or sixteen) can cause additional overhead that negates some of the performance gains.

Figure 5.3 displays the query runtimes from the STATS-CEB benchmark. Queries around the median are gathered into a boxplot, while outliers are presented with a scatterplot. The majority of queries are contained within the boxplot, while the same 22 queries out of the 125 total queries always occur as outliers. Median values inside of the boxplots slightly differ from those in Table 5.2 since they only account for the non-outliers. Important to note is that the outliers in this graph do not correspond to the outliers mentioned in table 5.2. Outliers mentioned in the table are those that have an incredibly large impact on the average measurements, with their runtimes being above 1500 ms, while the queries in the graph count as outliers when they fall outside a certain range from the median. Figure 5.3 is reproduced in A.2 for better readability.

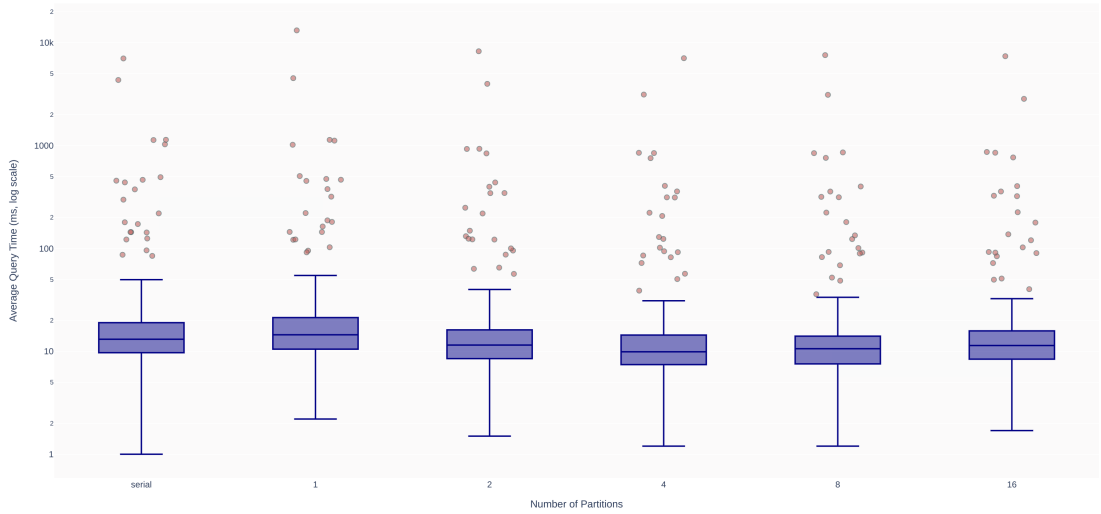


Figure 5.3: Scatter- boxplot for runtimes of different partition counts

From the figure, an increase in average runtimes is visible when comparing the serial case to the parallel configuration with one partition. An especially large performance decrease is visible in the longest-running query of the benchmark (query 126), explaining the large difference in average runtime with and without outliers in Table 5.2. Overall performance gains are observed from one to eight partitions, while a decrease in performance is visible at sixteen partitions. This degradation in performance can be attributed to the overhead contributed by the large degree of parallelism. Since a large number of workers will be created throughout these evaluations, context switches occur often, alongside an increase in scheduling for the different workers. The sixteen cores present on the machine on which these tests were performed can't coordinate these large numbers of workers efficiently.

5.3.2 Results with specifying core counts

For this subsection, all runtimes were measured again, but this time, instead of letting the Tokio runtime use as many cores as possible (16 in my case), the number of used cores equaled the number of partitions that the data is divided into. This limits the degree of parallelism and should slow down the runtimes, since horizontal and vertical parallelism cannot be utilized at the same time. An upside to this could be a decreased amount of scheduling needed, since fewer threads may lead to less context switching and simpler task coordination.

Table 5.3 contains some information from these measurements. The two outlier queries, again, have a large impact on the measurements taken. Since they have larger deviations, the averages where they are present differ from the averages in Table 5.2, while the averages where they are omitted are more in line with those from the earlier table. This phenomenon is especially visible when comparing the average runtime of two partitions between the two tables. The difference in average runtime for two partitions can be attributed to two cores not being enough for the large query 126, since, with an average of ∼9700ms for these measurements and an average of ∼8250ms for the previous measurements, it varies greatly between the two tests. The other average results from these measurements mostly overlap with those from the previous subsection.

What is interesting, however, is that with these measurements, instead of one partition being faster in 33 cases, it is only faster in four cases. This is because we still increase the workload, but since we only allow our runtime one core, any upside gained by vertical parallelism cannot be achieved since there is no way of completing tasks concurrently. There are also some other changes visible when comparing this column to the same column in Table 5.2, but these can

Core Count	Average (ms)	Average without outliers (ms)	# Faster queries compared to serial
Serial	168.66	73.22	X
1 core/partition	219.53	77.27	4
2 cores/partitions	176.70	58.30	111
4 cores/partitions	136.92	52.07	118
8 cores/partitions	138.21	51.70	112
16 cores/partitions	138.89	52.93	102

Table 5.3: Table with average and mean for different partition counts with varying core counts

be attributed to varying runtimes, even though an average of runtimes was taken for each query.

5.3.3 Conclusions for this section

From the graph in Figure 5.3, it is clearly visible that the linear speedup from Figure 5.1 is not reached. From the tables present in the subsection, a small increase in performance can be seen, though not linearly. This was to be expected, since the required full materialization in the group-by operator heavily limits parallelism, alongside the necessary synchronization of partitions in the newly implemented exchange operator. There is also a general decrease in performance when running the queries with sixteen partitions. This can be attributed to the machine used for the timings only having 16 cores, and not all cores can be used for the calculations at once, since other processes run concurrently. In Figure 3.11, DataFusion had also decided to make use of four partitions to enable parallelism via their repartitioner, indicating that this partitioning count should indeed be optimal in their case.

Because most queries are found in the boxplots, the few outlier queries have a disproportionate impact on the average measurements, making those measurements less reliable. The average where they are left out, however, paints a slightly different picture, especially when comparing the serial implementation to the 1 partition runtimes. I initially expected more of a difference between the results with and without specifying core counts; the main difference here is with one partition/core count. When using sixteen cores and one partition, vertical parallelism is still possible since the workers created by the exchange operator(s) can wait for their input concurrently, limiting the amount of context switching necessary. When only allowing one processing core, there is a need for context switching between the tasks started by the exchange operator(s), increasing runtime duration. Measurements like speedup and efficiency will be calculated in the next Section 5.4 for separate queries, alongside independent operator timings. This will allow us to get a better grasp on what is exactly going on during the different runtimes and where any bottlenecks may appear.

5.3.4 Results for the STATS-CEB benchmark in DuckDB

To frame the results that were measured using the newly implemented exchange operator, the same 125 queries were measured in DuckDB [6]. In DuckDB, it is possible to specify the number of threads that the runtime can use. By altering this number, the degree of parallelism changes, and the total runtime changes along with it. A sidenote to make is that since only the thread counts change, there is no separate one-thread parallel runtime. Both figures relevant for this explanation (5.4 and 5.5) are reproduced in A.3 and A.4 for improved readability.

Visible in Figure 5.4 is the scatterplot also used in Figure 5.2 but with measurements taken from DuckDB. Each dot again corresponds to a query runtime, comparing the serial runtime to the fastest parallel runtime. A dot being under the diagonal line corresponds to a query that received a performance increase from parallelism. In Figure 5.2 there are three dots that are distinctly visible above the diagonal line, indicating a degradation of performance. In the case of DuckDB, however, this is never the case. Though there are still six queries that perform better serially than they do in parallel, the differences are much smaller.

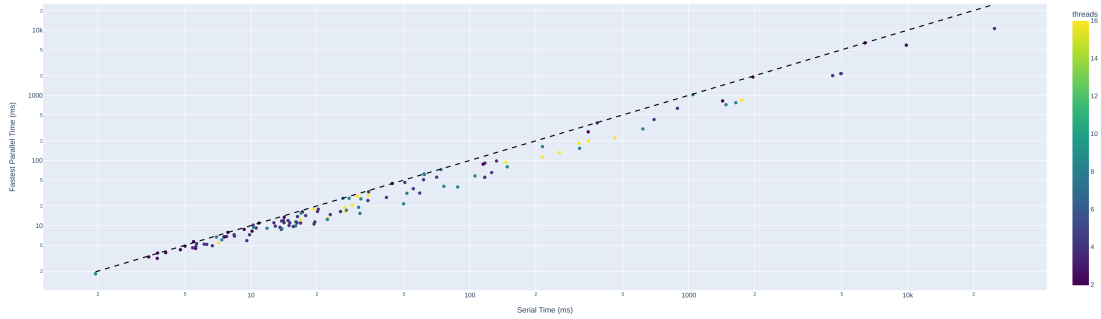


Figure 5.4: Log-log scatterplot for STATS-CEB benchmark in DuckDB

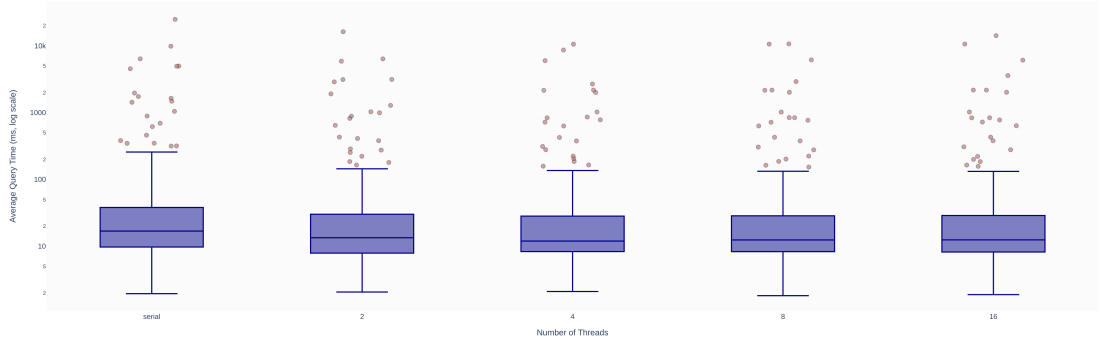


Figure 5.5: Scatter- boxplot for runtimes of different thread counts in DuckDB

Figure 5.5 displays the separate query runtimes for the benchmark. Queries around the median are again gathered into a boxplot, while outliers are present as dots in a scatterplot. The majority of queries are again contained within the boxplot. The overall runtimes lie a lot lower using the Shredded Yannakakis approach, the distribution of queries also differs a bit. Query 126 is still the largest outlier, taking ∼25 seconds to complete serially, where the median of the benchmark lies at 17ms. It does greatly benefit from parallelism up to four threads. As with the measurements of Shredded Yannakakis, eight and sixteen core counts do not really offer up much of a performance increase; this can again be attributed to scheduling overhead. Remarkable in these measurements is the presence of query 38 as the second-longest running query of the benchmark. This query did not stand out in the measurements for Shredded Yannakakis, while for these measurements its runtime is greatly increased, and when comparing its high-thread runtimes, it takes 14193 ms on sixteen threads compared to 10593 ms on eight threads, indicating potential overhead or other issues and becoming the longest-running query at sixteen threads.

Table 5.4 presents some more specific runtime measurements. Just like in table 5.2, the 4-thread/partition runtime proves to be the fastest, offering a balance of scheduling overhead and effective parallelism. Large queries again skew the average by quite a bit, which is why an average without the very large outliers is also present in the table. From this table, we can see that DuckDB also does not approach the ideal speedup. While the total measurements do point to better parallelism, this is not the case for every single query. Pointing to queries like query 38 again, their performance degrades greatly when run on more threads, while a degradation of this scale has not been measured in the Shredded Yannakakis approach.

5.4 Results for separate STATS-CEB queries

In this section, a more in-depth analysis will be made for several individual queries from the STATS-CEB [12] benchmark. We will take a closer look at runtimes per partition count, op-

Thread Count	Average(s)	Median(s)	Average without outliers(s)
Serial	0.585	0.0225	0.261
2 threads	0.405	0.0190	0.181
4 threads	0.351	0.0164	0.152
8 threads	0.370	0.0167	0.154
16 threads	0.404	0.0168	0.160

Table 5.4: Table with average and mean for different thread counts for DuckDB

erator metrics, and the difference between the presence of an extra exchange operator (see Subsection 4.2.4) between the joining and unnesting phase, where this makes a notable difference. This extra operator better balances the workload in the unnest operator, alongside enabling vertical parallelism between the unnest operator and the uppermost multisemijoin operator in the query tree. The measurements from this section are averages of 30 runs.

Figure 5.6 displays the query plans of the three queries that will be further analyzed in this Section. All three queries share a similar main structure, with the join phases containing repetitions of multisemijoin, group-by, and scan groups, eventually feeding into the unnest operator. The queries from the benchmark all return a count aggregate; as such, this aggregate is the final operator in the serial plan, from which the output is retrieved. To parallelize these serial plans, the conversions visible in Figure 4.6 are applied. The numbers next to certain operators in the plans of queries 31 and 133 are identifiers used in figures 5.9 and 5.14.

5.4.1 Query 31

Query 31 of the benchmark is a right-deep plan with four multisemijoin operators and three group-by operators. Its average runtime when compared to the rest of the benchmark lands it around the median of the boxplot in Figure 5.3. Its output size of 6 672 465 places it above the benchmark median of 1 356 723, though they share their order of magnitude. Its output size, alongside its average serial runtime makes it an average query for the benchmark.

Comparing serial to parallel

From Figure 5.7 we observe that there is a clear difference between the runtimes with usable parallelism and those without. The serial and one partition runtimes both lie around the same average, with the parallel one partition runtime lying slightly lower. We attribute this to variations in hardware performance. When we go higher than one partition, we do see improvements. Two, four, and eight partitions all see an improvement in performance when compared to their previous measurements; the jump from four to eight partitions is a small one, however. When running with sixteen partitions, performance decreases as expected from the overarching benchmark results.

For speedup and efficiency in Figure 5.8, we immediately notice that we do not approach the ideal speedup nor an efficiency value of one. The speedup does increase throughout one to eight partitions, mirroring the increased performance visible in the boxplot 5.7. This does, however, not scale linearly with the number of extra partitions. Similar to the average time, there is only a slight improvement when increasing from four to eight partitions. However, speedup decreases when moving from eight to sixteen partitions, likely due to scheduling overhead and the fact that the machine's sixteen cores cannot all be effectively utilized simultaneously for the computations. The efficiency keeps on decreasing as the partitions increase; this is to be expected when looking at the chart plotting the average runtimes for each partition count. This indicates that the cores are not used efficiently when increasing the number of partitions into which the data is split.

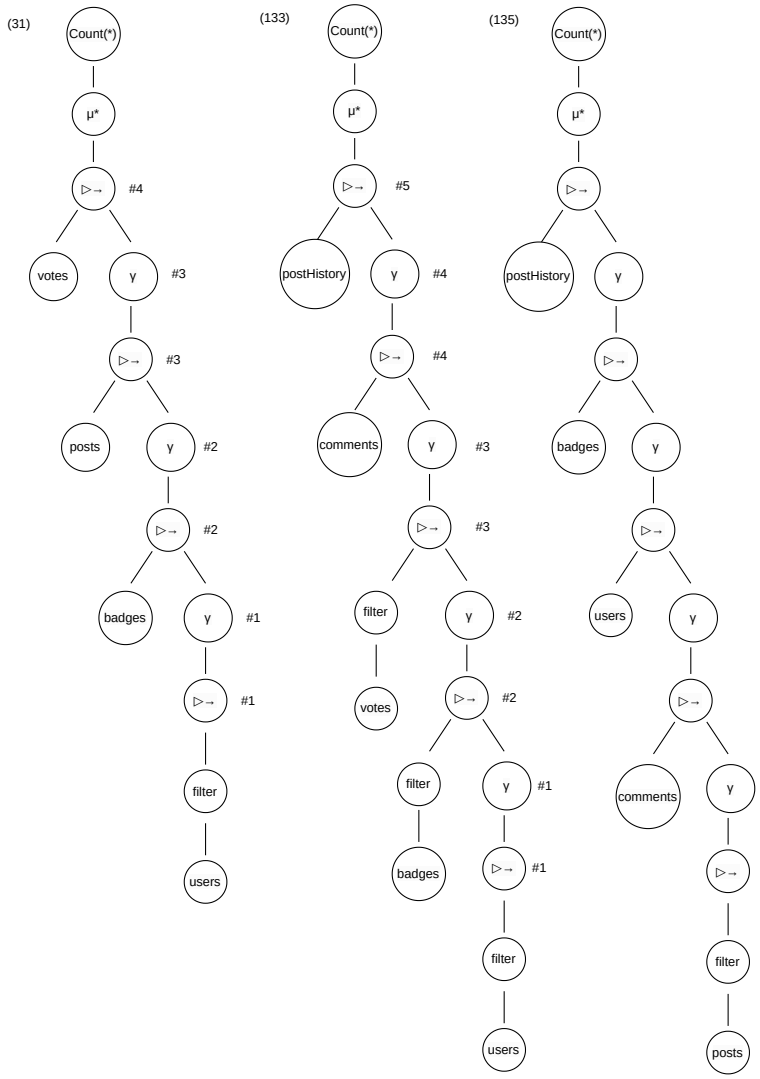


Figure 5.6: Query plans of analyzed queries

Comparing operators

For a closer inspection of independent operators, each operator holds several metrics. The exchange operator, for example, measures the time it takes waiting for the nestedcombiner (described in Section 4.2.3) to combine its columns and the amount of time it spends partitioning batches, among others. An overarching metric for several operators is called its **elapsed compute**. This metric measures the amount of CPU time an operator occupies until it has processed its last batch. When a CPU core waits for input in an operator, that wait will add to the elapsed compute of that operator. Notably, the multisemijoin operator differs from exchange and groupby in how it receives data: it does not consume input through a stream. As a result, its elapsed compute time excludes time spent waiting for input. In contrast, both the exchange and groupby operators poll their input as a stream. When they begin polling their child operators, their elapsed compute timers start, thereby including the time it takes for all descendants to produce the first batch. This can significantly inflate their elapsed compute, especially since the groupby operator requires a full input materialization before producing

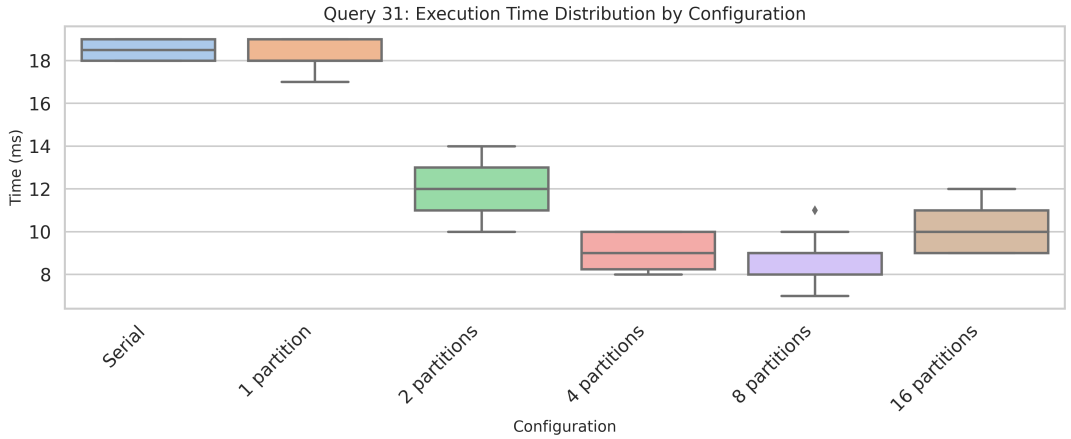


Figure 5.7: Boxplot of runtime measurements for query 31

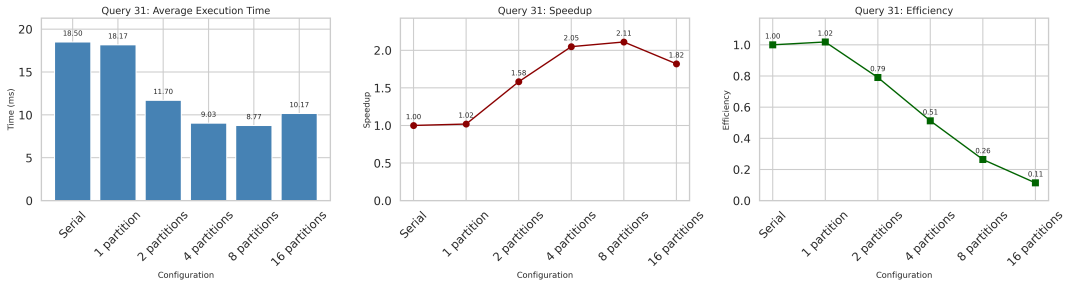


Figure 5.8: Various metrics graphs for query 31

any output. Consequently, the elapsed compute times of groupby and exchange operators will include the full compute time of their downstream operators, leading to the ever-increasing elapsed compute timers of exchange and groupby operators.

The gradual increase of elapsed compute time is visible in Figure 5.9. Each of the three graphs show a different type of runtime for query 31, the first plot displays the serial runtime. The plot on the right shows a parallel runtime without an exchange operator between the joining and unnesting phases; the plot at the bottom features this extra operator set to use a round-robin partitioning scheme. Both parallel runs only feature two partitions in order to keep the graphs orderly. Since all graphs have the same scale for their y-axis, we can make comparisons across graphs. The general throughline is that the average time each operator has to spend is slightly lower in the parallel case than it is in the serial case. Hence, an overall performance increase is gained. A noticeable change lies between the lowest multisemijoins, when comparing the serial case to the parallel cases, the elapsed compute of the parallel cases lies about 1000x lower than that of the serial case. The unnest time of the unnesting operator is unbalanced between the two partitions in the normal parallel case, an issue that can be solved by adding the extra exchange operator. An issue with this operator, however, is that its own runtime is extraordinarily long. The cause of this is the time it takes for the operator to send all its batches to the unnest operator. This metric is called its *send time*, and takes up more time in this case than it does in the normal cases. This behavior is also observed in the standard DataFusion exchange operator. Figure A.5 is reproduced in A.5 for better readability.

Further observations can be made regarding each group of exchange, multisemijoin, and group-by operators. Since group-by operators must wait for all input data before proceeding, their runtime is significantly higher than that of multisemijoin operators. When examining group-

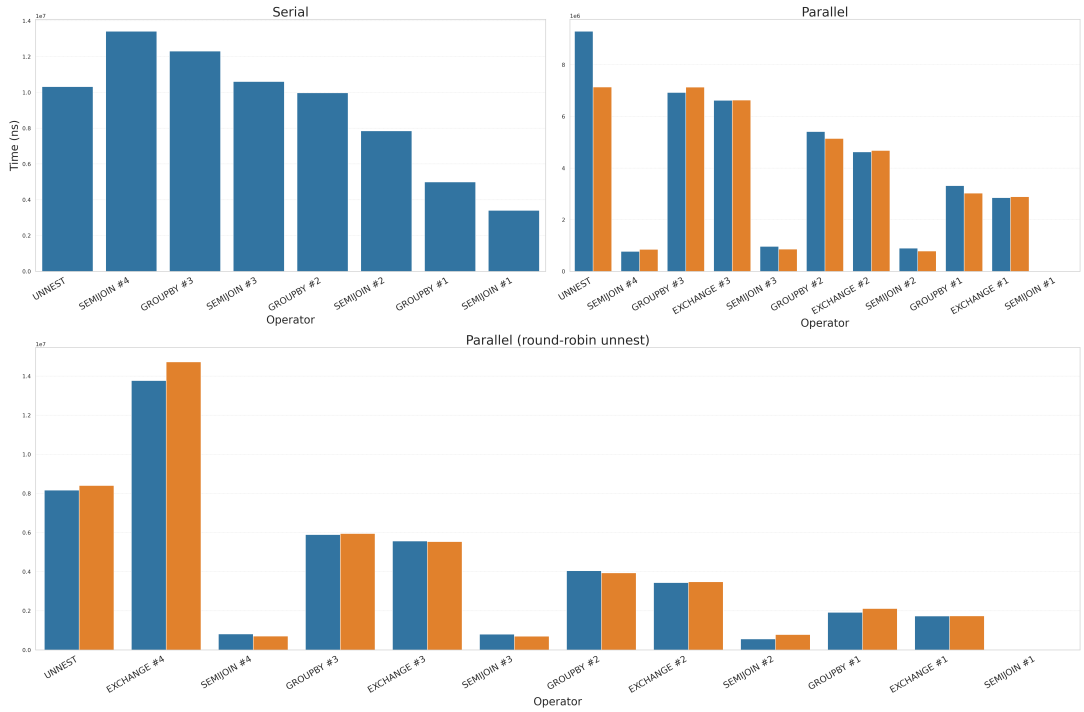


Figure 5.9: Operator compute durations for three different runtimes for query 31

by operators in isolation, a large portion of their elapsed compute time is spent waiting for their child multisemijoin operators to finish. This is due to the full materialization required before the group-by can begin processing. Conversely, a multisemijoin operator that follows a group-by has a much shorter elapsed compute time because its timer only increments when the preceding group-by has completed, and it can actually perform its computations. When we sum the elapsed compute times of a multisemijoin operator and its child group-by operator, we obtain an approximation of the elapsed compute time of the subsequent exchange operator, whose elapsed compute is also inflated due to busy waiting behaviour, just like in the group-by operator. When taking a closer look at an exchange operator, we see that most of its time is spent waiting for data to arrive, more precisely, waiting for its first batch to arrive. It then possibly waits a short amount of time for all other partitions to receive their first batch, and then proceeds to the nested combiner. This necessary synchronization phase is a small fraction of the total time spent in this operator.

The specific breakdown of the exchange operator that slots between multisemijoin and groupby #3 in Figure 5.6 from the parallel case without round-robin is visible in Table 5.5. From this table, it immediately becomes clear that most of the operator's time is spent waiting for the first batch to arrive. The repartition_time also takes a relatively long time when compared to the other metrics. This is because it constitutes all the work that the operator is supposed to do. This metric is made up of pulling a batch from its corresponding input channel, calculating hashes for tuples, partitioning the tuples based on the hashes, and sending them to the relevant output channels. The reason why combine_timer is zero in partition 1 is that this partition is not responsible for performing this operation.

5.4.2 Query 133

Query 133 is another query that finds itself in the boxplot in Figure 5.3. This query is presented more in-depth due to the partitioning of data being skewed when evaluating the query with a high number of partitions. Its output size of 13 971 410 places it above the benchmark median of 1 356 723 rows.

Metric	Partition 0	Partition 1
fetch_time	41 989	217 078
first_batch_time	5 456 488	5 201 409
barrier_time_1	541	96 441
barrier_time_2	7 214	160
combine_timer	23 003	0
lock_time	511	351
repartition_time	1 089 178	1 111 620
send_time	2 193	7 025
elapsed_compute	6 622 951	6 628 942

Table 5.5: Operator Metrics for the newly implemented exchange operator in ns

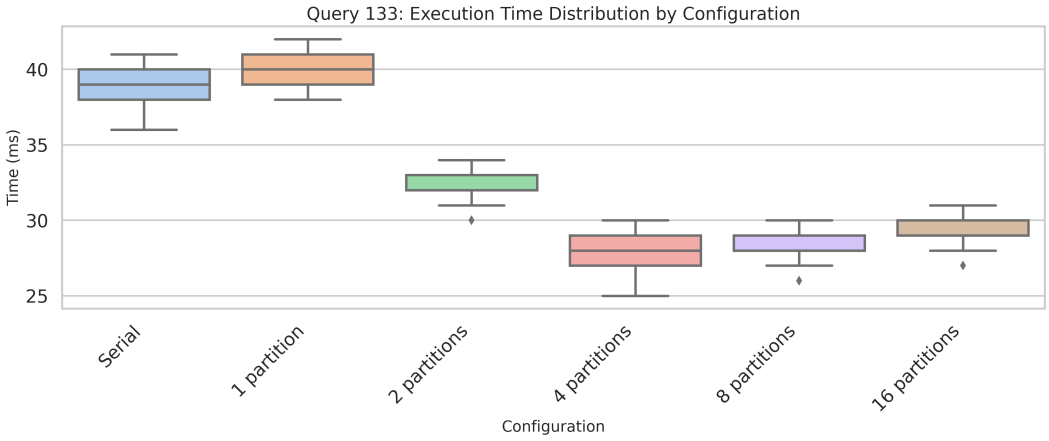


Figure 5.10: Boxplot of runtime measurements for query 133

Comparing serial to parallel

The query shows a by-now expected degradation of performance for one partition when compared to the serial runtime. As with query 31 from Subsection 5.4.1, a boost in performance is visible when inter-operator parallelism is possible with two partitions, and again when the degree of parallelism is increased with four partitions. This query, however, already starts degrading at eight partitions, with further performance loss at sixteen partitions. The reason for this performance degradation becomes apparent when comparing the specific operators' times. The unnesting operator accounts for a large amount of the total query runtime, with poor balancing of work in this operator, we do not stand to gain much performance increase from parallelism.

As expected from the average measurements, the speedup calculated for this query lies lower than that of the previously analyzed query. A similar efficiency graph to the one in Figure 5.8 is also present.

Adding the round-robin exchange operator

The amount of output rows a partition ends up with is a good indicator of its workload it has to perform in the unnest operator. If this number is compared to the total number of output rows the query produces, data skew across partitions can be analyzed. Recall that query 133 produces 13 971 410 output rows. In Table 5.6, metrics for the unnest operator of this query are visible. For each configuration, the first column shows the maximum number of input rows the unnest operator gets in a single partition. The second and third column

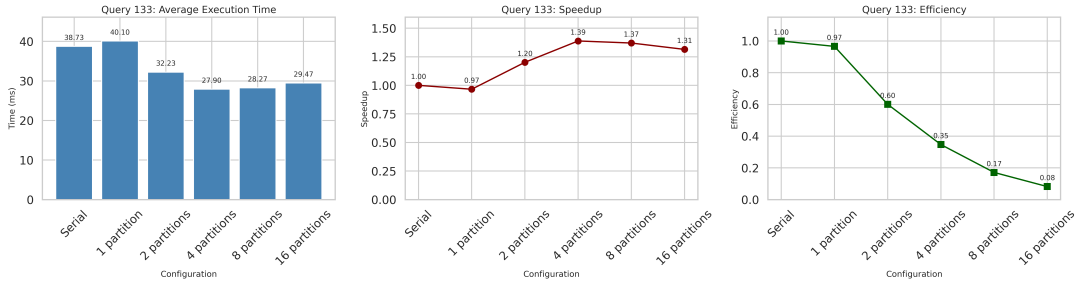


Figure 5.11: Various metrics graphs for query 133

Configuration	Max Input Rows	Min Output Rows	Max Output Rows
Serial	9 197	13 971 410	13 971 410
4 partitions	2 708	696 351	9 673 345
4 partitions + round-robin	2 358	2 562 392	4 736 971
8 partitions	1 388	288 557	9 153 273
8 partitions + round-robin	1 173	816 362	3 020 560

Table 5.6: Detailed metrics for the unnest operator of query 133

show the minimum and maximum number of output rows that the unnest operator produces in a single partition. A skewed workload is visible when looking at the regular four and eight-partition configurations. In both cases, a large majority of the output rows ($\sim$ 65-70% in both cases) are produced in a single partition, leading to a strongly unbalanced workload and a disproportionately long runtime in a single partition. This longer partition leads to a longer overarching runtime since the query can only end after this partition has fully unnested its output. Adding a load-balancing operator via an extra exchange operator decreases this skew considerably. While the workload is certainly still skewed, the maximum number of output rows more than halves in both cases, leading to more balanced workloads and theoretically decreasing the overall runtime.

This more balanced workload will theoretically prevent any single partition from becoming a bottleneck due to having a disproportionately large share of data. In practice, due to the elementary nature of round-robin partitioning, this could still happen. Ideally, a new partition scheme would be made that takes the amount of work necessary to unnest a batch into account.

In the two figures that follow, average measurements for the round-robin configuration can be found. The results of this configuration lead to a higher performance gain than with the configuration described previously. There is still a performance loss from eight partitions onwards. This is to be expected, since the data skew occurs in more operators than just the unnest operator. The increased vertical parallelism gained by adding the exchange operator now leads to a faster 1 partition runtime.

When observing the speedup for this configuration plotted in Figure 5.13, it can be concluded that this configuration is a better fit for query 133 than the configuration without the extra round-robin partitioning. This is due to the utilized hash function leading to a data skew that greatly influences the amount of time the unnest operator takes to use.

Looking at the independent operator durations in Figure 5.14, the large skew in unnest time becomes clear. When the round-robin exchange operator is not inserted, the unnest time dominates the runtime, since as mentioned earlier, the workload for this operator is distributed poorly. Just like for query 31, the durations of the groupby and exchange operators take longer due to them including downstream wait times. Because of them being forced to wait for downstream operators to finish, a logical follow-up would be to suggest that a general trend is

Figure 5.12: Boxplot for measurements of query 133 with an extra exchange operator

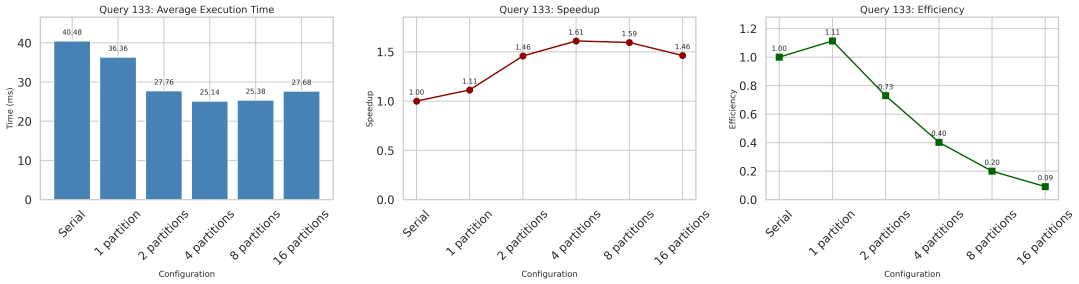


Figure 5.13: Various metrics graphs for query 133 with an extra exchange operator

that deeper plans will generally take longer than plans with fewer joins. Since more joins lead to more groupby operators, the runtime needs to wait for more full materializations, increasing the elapsed compute values of later operators and increasing the runtime duration as a whole. The figure itself is reproduced in A.6 for readability.

Comparing the standard two configurations

With the new configuration adding the round-robin repartitioning, query 133 sees lower average runtime measurements in the higher partition counts, resulting in a better speedup and efficiency. We can conclude that the large skew in data across partitions can be an important bottleneck in many queries. When this skew is not present or less pronounced, performing the extra work necessary to rebalance the data before unnesting it can be useless, increasing the runtime instead of shortening it. Creating a new partitioning scheme specifically to feed into the unnest operator would also lead to more work in this operator since this partitioning is more complex than the current round-robin partitioning.

5.4.3 Query 135

The last query that will be investigated in more detail is query 135. This is one of the two outliers omitted in certain calculations of Section 5.3 due to its large runtime. This large runtime is attributed to the fact that most queries in the benchmark have filter operations that reduce the amount of data to be processed. This query, however, features comparatively little filtering and will process the large amount of data in its entirety. Query 135 is a join between five tables, resulting in 2 263 957 167 outputs, the second-largest output size of the benchmark.

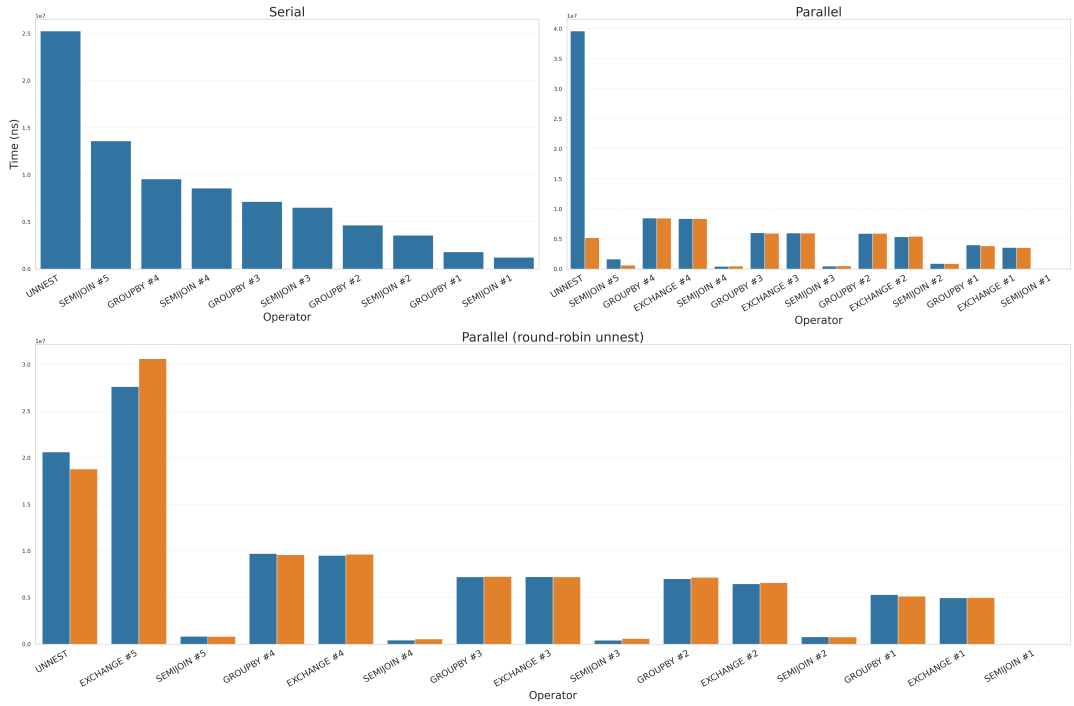


Figure 5.14: Operator compute durations for three different runtimes for query 133

Comparing serial to parallel

The large output size of this query is caused by an explosion of rows in the unnest operator. In total, 43 135 input rows enter the unnest operator, with 2 263 957 167 output tuples leaving it. This explosion of rows brings a large workload for the unnest operator along with it. As such, a large majority of the runtime is spent in the unnesting phase, rather than actually joining the data. For this query, balancing the partitioning of data headed towards the unnest operator is once again a major factor in speeding up the runtime.

Figure 5.15 displays some boxplots for measurements of query 135 for a standard configuration of parallelism. When comparing the serial runtime to the parallel runtime with one partition, we see a slight improvement when going parallel, though these two are comparable, except for the large outlier in the parallel case. There is an improvement visible in increasing the degree of parallelism by splitting the data into two partitions, though only slightly. This points to a skewed distribution of the workload that is to be performed in the unnest operator, since most of the possible performance gain is located in this operator. A larger increase is visible when splitting into four partitions, indicating a better distribution of the workload. Unlike the previous two queries, there is no degradation in performance visible when working with sixteen partitions, though there is also little to no performance increase when comparing it to eight partitions. Since the distribution of data in the unnest operator is crucial to increasing performance, adding an extra round-robin repartitioning before this operator could increase performance.

The metrics shown in Figure 5.16 show that the serial and one-partition runtimes both lie around the same duration. Because of this, the speedup for the 1-partition runtime is 1.00. The speedup then slightly increases with two partitions, while a larger jump is visible at four. Notable is that the speedup for both the eight- and sixteen-partition runtimes is 1.65, also indicating that they approximate each other's average runtimes.

Figure 5.15: Boxplot of runtime measurements for query 135

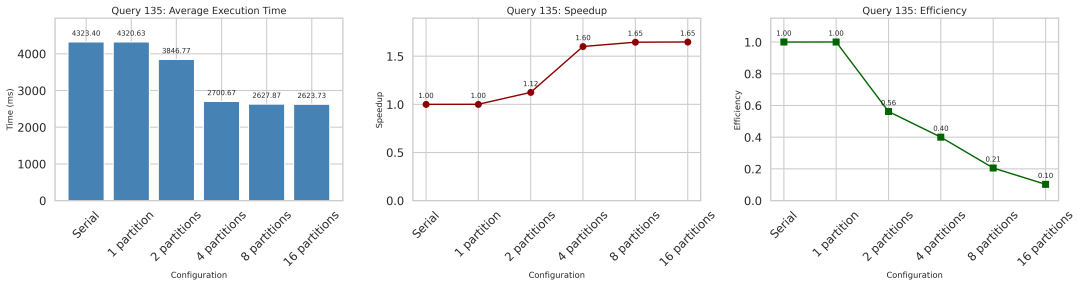


Figure 5.16: Various metrics graphs for query 135

Figure 5.17: Boxplot for measurements of query 135 with an extra exchange operator

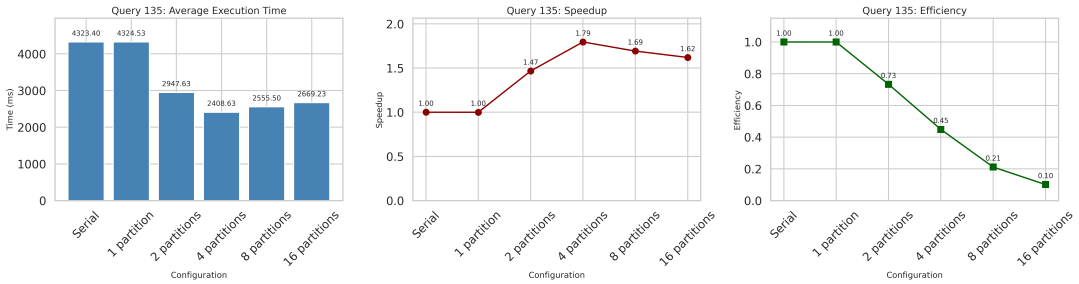


Figure 5.18: Various metrics graphs for query 135 with an extra exchange operator

Adding the round-robin exchange operator

Figure 5.17 displays the boxplots for 30 measurements of query 135 where an extra round-robin exchange operator was inserted between the join and the unnest phases of the query evaluation. In contrast with the measurements from Figure 5.15, there is a decrease in performance when evaluating with eight and sixteen partitions when compared to the four-partition runtime. From this boxplot, it becomes clear that splitting the data into four partitions gives the fastest overall runtime, just like most other queries in the benchmark.

Figure 5.18 now shows the average, speedup and efficiency metrics for this configuration. A slightly larger maximum speedup is visible when compared to Figure 5.16, going from 1.65 to 1.79. This indicates that the addition of the exchange operator proved beneficial to the performance. This configuration does see a peak at four partitions, while the previous configuration still benefited from eight and sixteen partitions. This can be attributed to the fact that the extra exchange operator will spawn additional workers, leading to sixteen concurrent workers for the eight-partition case and 32 concurrent workers for the runtimes with sixteen partitions, half of each responsible for pulling data from the multisemijoin operator and half responsible for pulling data from the unnest operator. The sixteen cores present on the machine that ran these tests will not be able to actually run all these workers in parallel, forcing context switches that degrade performance, potentially degrading it so far that it takes longer to perform with the extra operator than it would without it. For this query we omit the graph displaying independent operator runtimes, since the time spent during this runtime is so skewed towards the unnest operator that the other operators are not visible in the graph.

Configuration	Average runtime (ms)	Max Input Rows	Min Output Rows	Max Output Rows
Serial	4 323	43 135	2 263 957 167	2 263 957 167
2 partitions	3 846	26 337	277 095 707	1 986 861 460
2 partitions + round-robin	2 947	22 113	1 038 370 724	1 225 586 443
4 partitions	2 700	14 802	18 217 437	1 086 628 254
4 partitions + round-robin	2 408	11 735	444 395 746	704 935 813
8 partitions	2 627	11 796	4 600 632	1 048 588 666
8 partitions + round-robin	2 550	5 965	172 810 156	402 569 437
16 partitions	2 623	7 778	201 604	847 978 287
16 partitions + round-robin	2 669	3 704	70 523 645	310 489 591

Table 5.7: Table with unnest metrics for query 135

Comparing the two configurations

To further illustrate the performance differences and their links to the unnest workloads, Table 5.7 displays the average runtimes for the different configurations, alongside unnest metrics that correspond to the workload to be performed in the operator. An important metric for determining the influence that the distribution of data will have on the runtime is the maximum output rows in the unnest operator. If a majority of the output rows stem from a single partition, that partition will have a disproportionately large workload and will take considerably more time unnesting its input than the other partitions would, acting as a bottleneck that increases the total query runtime. If the maximum number of output rows is lower, the partition with the highest workload has less work to perform, decreasing its runtime and decreasing the bottleneck effect it has on the runtime. There is a significant difference in maximum output rows when comparing the regular runtimes to those with the added round-robin exchange operator. In the lower partition counts (two, four, and eight), this leads to a performance increase due to better balancing and an increase in vertical parallelism. With eight partitions, however, the increase in performance is a lot smaller than those of the previous two partition counts, and the large number of workers starts to cause overhead with context switches and scheduling. Finally, when compared to the runtime with sixteen partitions, while the unnest operator still benefits from a better balancing of workload, the large number of workers starts to bring too much overhead with them, causing a decrease in performance when compared to eight partitions and an even further decrease when adding the extra exchange operator.

5.4.4 Conclusions for this section

While the ideal speedup and efficiency are not reached with the added exchange operator, there is most definitely a performance gain visible by allowing the queries to run in parallel. The speedup measured is dependent on the structure of the query plan that is to be evaluated and the amount of data that flows through it. Even though the implementation of parallelism mostly focuses on the join phase, the impact that the unnest phase has on the overall runtime is not to be underestimated. The exchange operator does, however, also improve the performance of the unnest operator by allowing it to work concurrently across several partitions. It also promotes vertical parallelism between the join and unnest phases by inserting it between the two phases, with the added benefit of a possible rebalancing of data skew. A new partitioning scheme that takes the workload of the unnest operator could be desirable, granted that these calculations aren't too complex, since these computations could then lead to significant processing time themselves. The rebalancing of data before unnesting via an extra exchange operator can significantly speed up the runtime, though the extra work this brings might sometimes outweigh its benefit.

While the overall speedup of DuckDB is higher than that of the implementation presented in this thesis, it is not too far off. DuckDB also does not come close to the ideal speedup and efficiency, and just like with parallel shredded Yannakakis, the measured speedup when increasing the degree of parallelism is also dependent on the query itself.

Chapter 6

Conclusions

Throughout this thesis, practical experience was gained with query engines and their inner workings. The Volcano operator model served as a baseline on how operators that process data should be implemented and provided a clear structure to do so. The actual implementation of this model in Apache DataFusion was then studied to understand how this could be applied in practice. Going through the source code of Apache DataFusion also allowed me to learn how to approach open-source projects and eventually extend them, alongside learning the ins and outs of the Rust programming language.

Educating myself on the more theoretical aspects of the thesis present in Chapter 2 took some time due to the mixture of my having to refresh my knowledge on some concepts I had learned earlier, alongside learning new concepts like Yannakakis' algorithm itself. After creating the foundation necessary to understand the concepts of Shredded Yannakakis [4], I had to take some time to completely understand the approach itself, to then start analyzing the code implementation of Shredded Yannakakis. An important part of understanding the code implementation was to gain an understanding of how the data structures work. The data structures themselves I found to be fairly complex, with various levels of recursive nesting, and with the data being scattered throughout the objects due to the nature of query shredding itself.

During the implementation of the exchange operator within Shredded Yannakakis, theorizing about how it should be implemented proved to be much easier than actually implementing the operator itself. The issue presented in Subsection 4.2.2 proved to be fairly complex in practice due to the nature of the data structures. Crashes and errors quickly revealed that something was going wrong, and due to their nondeterministic nature, they pointed towards a race condition. Finding what exactly went wrong took quite some time and forced me to find a way to clearly visualize the data structures, since simply printing them out proved insufficient to finding the root cause of the issue.

Now, for the results themselves, even before solving the data shuffling issue, performance gains were visible when comparing the serial implementation to the one where parallelism was added. Although they do not approach the ideal speedup, this was to be expected given the context of the implementation. There are still many synchronization points present during the query evaluation, halting the different workers and slowing down the process. Increasing the degree of parallelism also improves efficiency for the new implementation, up until a certain point. This occurs due to scheduling overhead and the fact that every exchange operator spawns new workers, eventually leading to more workers than there are cores present on the system that carries out the query evaluation. Data skew caused by the hash function can greatly impact the performance increase that partitioning the data allows. When evaluating total query runtime, the larger the data skew, the more work there is to do in a single partition. Total query runtime is strongly linked to how long it takes to finish the longest-running partition, since it needs to be processed completely before the final result can be returned. The hash function used can

still lead to some data skew, possibly leading to performance losses. In general, though, most of the measured queries gained a performance boost from the newly implemented parallelization scheme.

Further work includes implementing the new repartition operator for table scans, which would allow the multisemijoin operator to actually join multiple groupby results at once in a partitioned context. Another possibility would be to implement an exchange operator that functions on top of the groupby operator to see if this results in better results. The impact that the unnesting phase has on performance is also not to be neglected. Though the current implementation of parallelism effectively distributes the work in this operator, a new partitioning scheme could be created to better balance this workload.

Bibliography

[1] *Apache Arrow DataFusion*. URL: https://datafusion.apache.org/.

[2] Marcelo Arenas et al. *Database Theory*. Preliminary version. Aug. 2022. URL: https://github.com/pdm-book/community.

[3] *Asynchronous Programming in Rust*. URL: https://rust-lang.github.io/async-book/05_streams/01_chapter.html.

[4] Liese Bekkers et al. "Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores". en. In: ().

[5] Altan Birler, Alfons Kemper, and Thomas Neumann. "Robust Join Processing with Diamond Hardened Joins". en. In: *Proceedings of the VLDB Endowment* 17.11 (July 2024), pp. 3215–3228. ISSN: 2150-8097. DOI: 10.14778/3681954.3681995. URL: https://dl.acm.org/doi/10.14778/3681954.3681995 (visited on 04/23/2025).

[6] *DuckDB: An in-process SQL OLAP database management system*. en. URL: https://duckdb.org/ (visited on 06/13/2025).

[7] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems: the complete book*. en. Second edition, Pearson new international edition. Always learning. Harlow: Pearson, 2014. ISBN: 978-1-292-02447-9 978-1-292-03730-1.

[8] Wolfgang Gatterbauer. "Topic 3: Efficient query evaluation Unit 1: Acyclic query evaluation Lecture 18". en. In: ().

[9] G. Graefe. "Volcano-an extensible and parallel query evaluation system". en. In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (Feb. 1994), pp. 120–135. ISSN: 10414347. DOI: 10.1109/69.273032. URL: http://ieeexplore.ieee.org/document/273032/ (visited on 04/15/2025).

[10] Goetz Graefe. "Encapsulation of parallelism in the Volcano query processing system". In: *SIGMOD Rec.* 19.2 (May 1990), pp. 102–111. ISSN: 0163-5808. DOI: 10.1145/93605.98720. URL: https://dl.acm.org/doi/10.1145/93605.98720 (visited on 04/15/2025).

[11] John L. Gustafson. "Reevaluating Amdahl's law". en. In: *Communications of the ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/42411.42415. URL: https://dl.acm.org/doi/10.1145/42411.42415 (visited on 05/13/2025).

[12] Yuxing Han et al. "Cardinality estimation in DBMS: a comprehensive benchmark evaluation". In: *Proc. VLDB Endow.* 15.4 (Dec. 2021), pp. 752–765. ISSN: 2150-8097. DOI: 10.14778/3503585.3503586. URL: https://doi.org/10.14778/3503585.3503586 (visited on 05/13/2025).

[13] Paris Koutris. *Lecture 4: Acyclic Conjunctive Queries*.

[14] Andrew Lamb. *Apache Arrow DataFusion Architecture Part 1*. Mar. 2023.

[15] Andrew Lamb. *Apache Arrow DataFusion Architecture Part 2*. Apr. 2023.

[16] Andrew Lamb. *Apache Arrow DataFusion Architecture Part 3*. Apr. 2023.

[17] Andrew Lamb et al. "Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine". en. In: *Companion of the 2024 International Conference on Management of Data*. Santiago AA Chile: ACM, June 2024, pp. 5–17. ISBN: 979-8-4007-0422-2. DOI: 10.1145/3626246.3653368. URL: https://dl.acm.org/doi/10.1145/3626246.3653368 (visited on 04/15/2025).

[18] Cetin Nurhan. *Speed-Up and Efficiency*. May 2001. URL: https://svn.vsp.tu-berlin.de/repos/public-svn/publications/kn-old/strc/html/node9.html (visited on 05/12/2025).

[19] Mark Raasveldt. *Push-Based Execution in DuckDB*. Nov. 2021.

[20] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Fourth edition. Pearson series in artificial intelligence. Hoboken: Pearson, 2021. ISBN: 978-0-13-461099-3.

[21] E. Schikuta. "Performance modeling of the Grace Hash Join on cluster architectures". en. In: *Proceedings International Parallel and Distributed Processing Symposium*. Nice, France: IEEE Comput. Soc, 2003, p. 7. ISBN: 978-0-7695-1926-5. DOI: 10.1109/IPDPS.2003.1213496. URL: http://ieeexplore.ieee.org/document/1213496/ (visited on 04/15/2025).

[22] Kevin Sterjo. "Row-Store / Column-Store / Hybrid-Store". en. In: ().

[23] *Tokio - An asynchronous Rust runtime*. URL: https://tokio.rs/ (visited on 05/12/2025).

[24] Qichen Wang et al. *Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees*. en. arXiv:2504.03279 [cs]. Apr. 2025. DOI: 10.48550/arXiv.2504.03279. URL: http://arxiv.org/abs/2504.03279 (visited on 04/22/2025).

Appendix A

Reproduced images for chapter 5

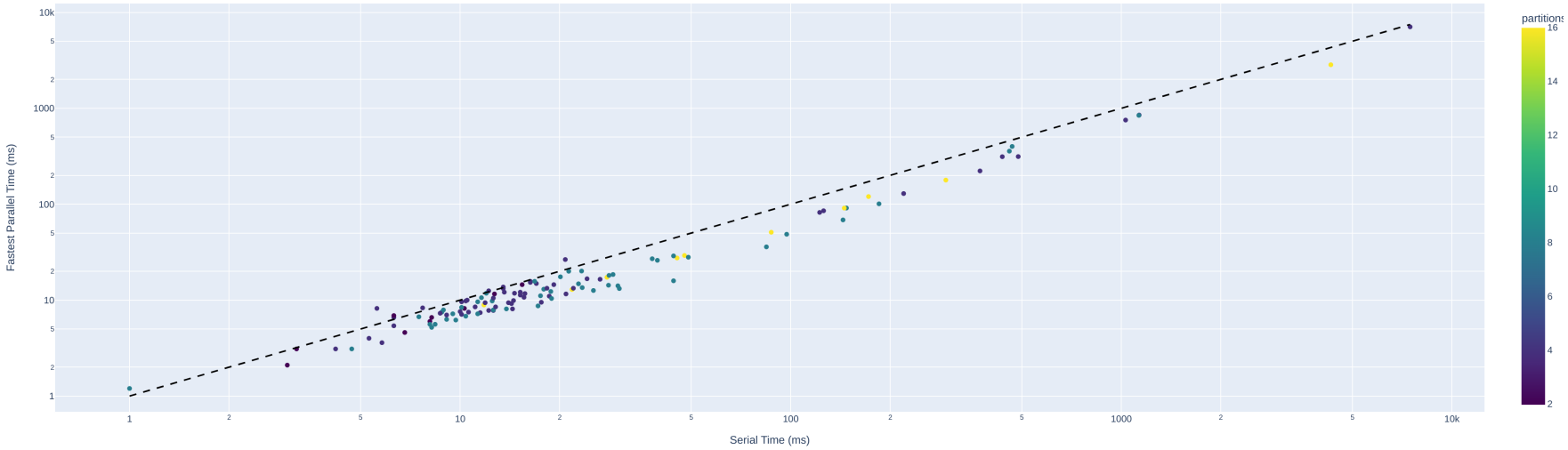


Figure A.1: Log-log scatterplot for STATS-CEB benchmark (Figure 5.2)

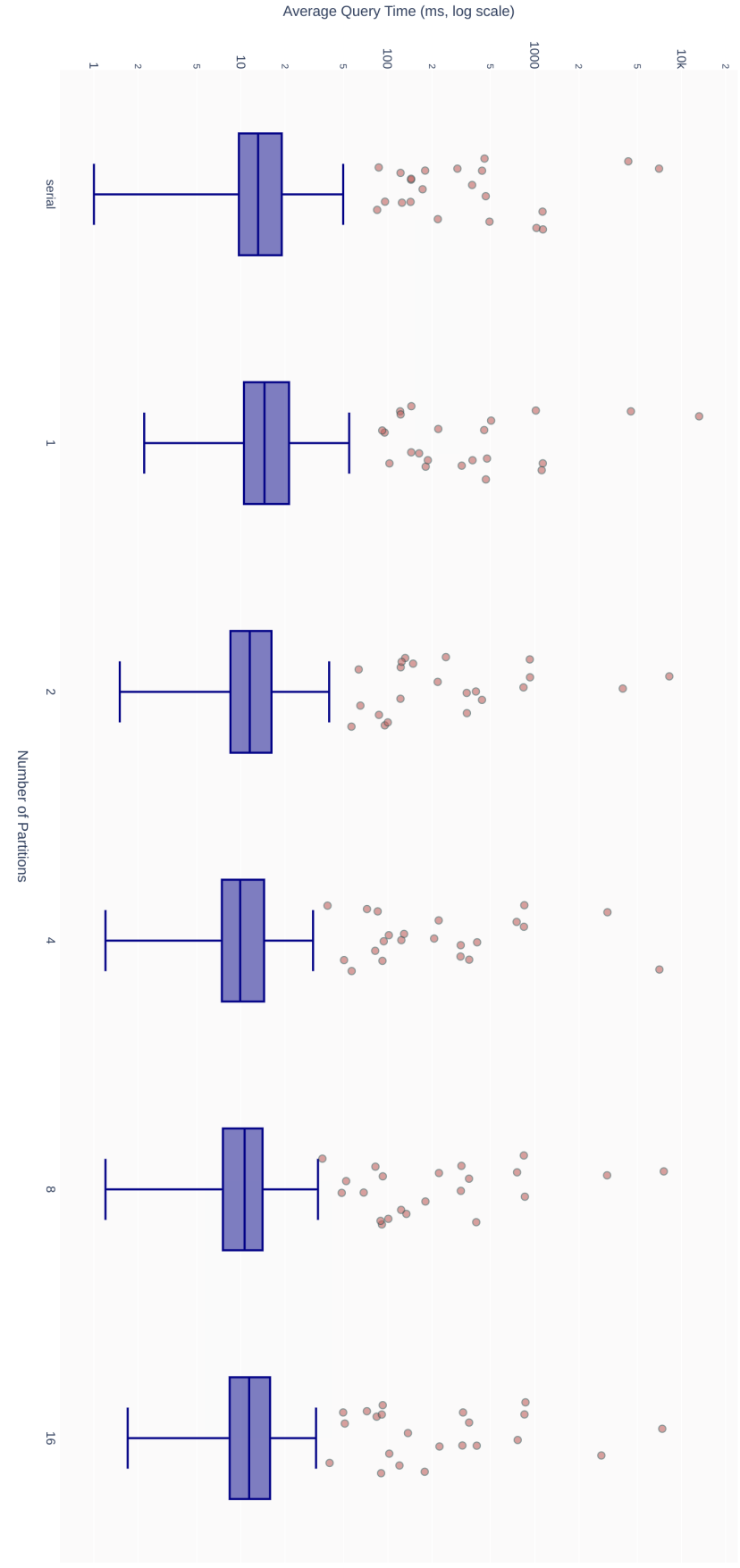


Figure A.2: Scatter- boxplot for runtimes of different thread counts (Figure 5.3)

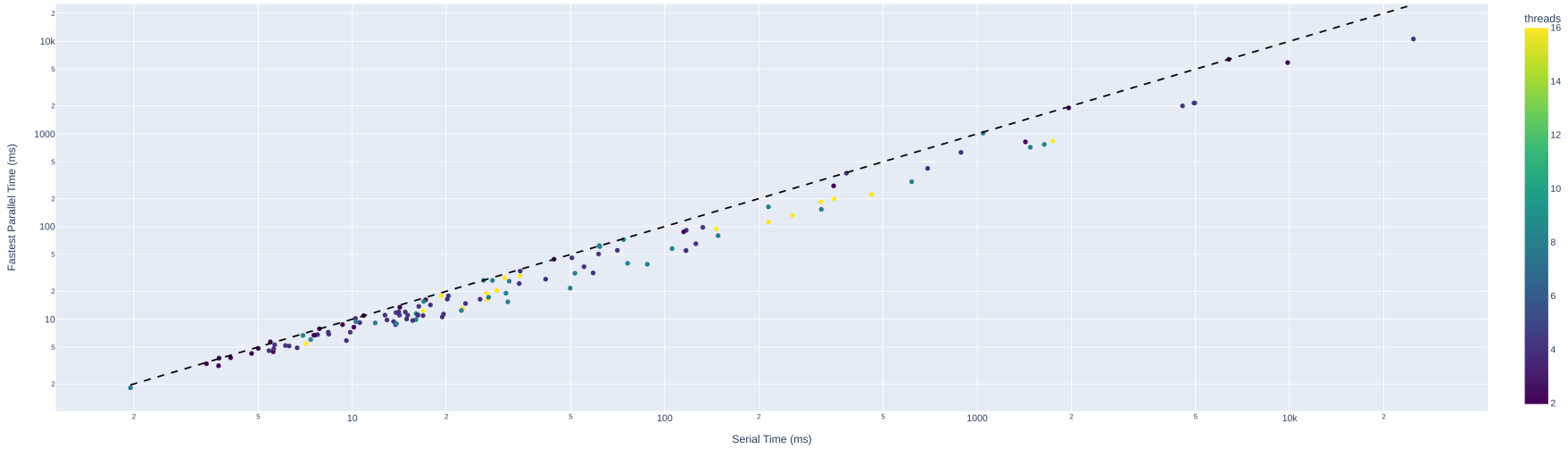


Figure A.3: Log-log scatterplot for STATS-CEB benchmark in DuckDB (Figure 5.4)

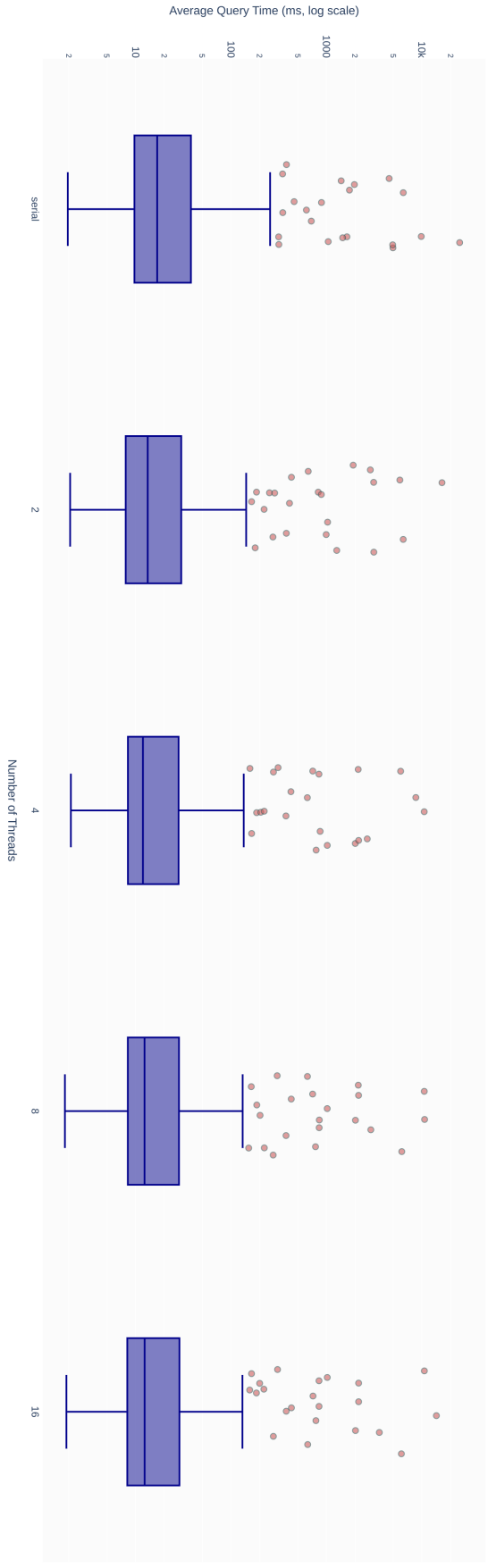


Figure A.4: Scatter- boxplot for runtimes of different thread counts in DuckDB (Figure 5.5)

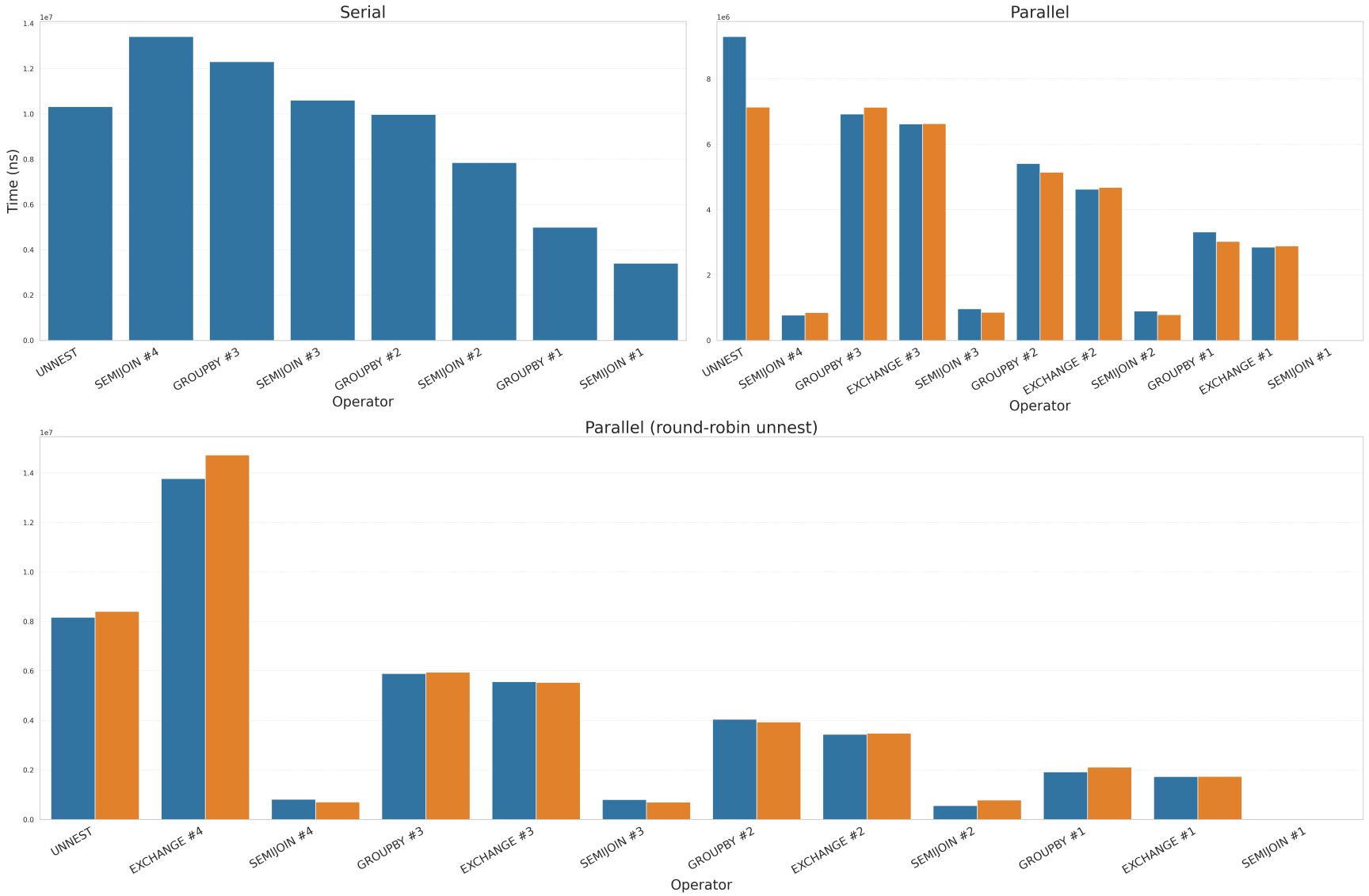


Figure A.5: Operator compute durations for three different runtimes for query 31 (Figure 5.9)

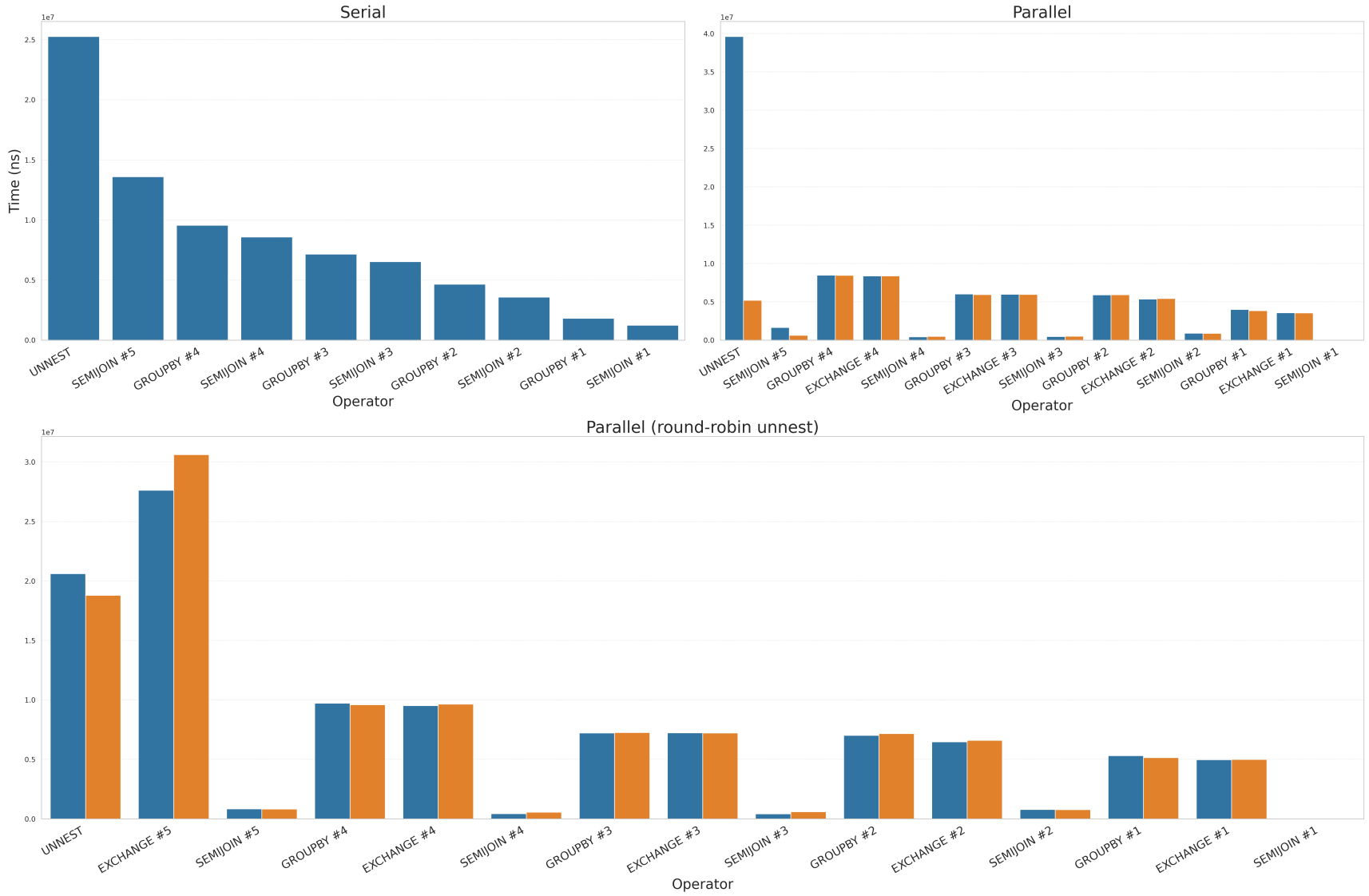


Figure A.6: Operator compute durations for three different runtimes for query 133 (Figure 5.14)