VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

# Přepisovací pravidlo v Apache Calcite

Apache Calcite Rewriting Rule

Vít Maňásek

Diplomová práce

Vedoucí práce: doc. Ing. Radim Bača, Ph.D.

Ostrava, 2023

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Č.j.VSB/22/078923
vsboee22025FFD

# Zadání diplomové práce

| | |
|---|---|
| Student: | **Bc. Vít Maňásek** |
| Studijní program: | N2647 Informační a komunikační technologie |
| Studijní obor: | 2612T025 Informatika a výpočetní technika |
| Téma: | Přepisovací pravidlo v Apache Calcite<br>Apache Calcite Rewriting Rule |
| Jazyk vypracování: | čeština |

Zásady pro vypracování:

Apache Calcite je projekt, který je zaměřen na vytvoření modifikovatelného optimalizátoru pro relační databáze. Calcite umožňuje kombinovat různé datové modely a různé databázové systémy do jednoho optimalizátoru a také nabízí možnost vytvořit vlastní přepisovací pravidla. Cílem této práce je implementovat přepisovací pravidlo pro window funkce, které jsou dnes stále častěji součástí běžných SQL dotazů.

Práce bude probíhat v následujících krocích:
1. Podrobné seznámení s projektem Apache Calcite a způsobem návrhu a aplikací přepisovacích pravidel.
2. Seznámení se se syntaxí window funkcí v SQL a možnostmi jejich přepisu na klauzuli GROUP BY.
3. Návrh a implementace přepisovacího pravidla, které bude umět detekovat vhodnost přepisu a provede transformaci plánu.
4. Otestování přepisovacího pravidla na umělých i reálných datech.

Seznam doporučené odborné literatury:

[1] Begoli, E., Akidau, T., Hueske, F., Hyde, J., Knight, K., & Knowles, K. (2019, June). One SQL to rule them all-an efficient and syntactically idiomatic approach to management of streams and tables. In Proceedings of the 2019 International Conference on Management of Data (pp. 1757-1772).

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Radim Bača, Ph.D.**

Datum zadání: 01.09.2022

Datum odevzdání: 30.04.2023

Garant studijního oboru: prof. RNDr. Václav Snášel, CSc.

V IS EDISON zadáno: 07.11.2022 11:59:21

17. listopadu 2172/15
708 00 Ostrava-Poruba
Česká republika

spojovatelka: +420 597 321 111
epodatelna: epodatelna@vsb.cz
ID datové schránky: d3kj88v

IČ: 61989100
DIČ: CZ61989100

email: studijni.fei@vsb.cz
www.fei.vsb.cz

## Abstrakt

Apache Calcite je projekt, který je zaměřen na vytvoření modifikovatelného optimalizátoru pro relační databáze. Calcite umožňuje kombinovat různé datové modely a různé databázové systémy do jednoho optimalizátoru a také nabízí možnost vytvořit vlastní přepisovací pravidla. Cílem této práce je implementovat přepisovací pravidlo pro window funkce, které jsou dnes stále častěji součástí běžných SQL dotazů.



## Klíčová slova

přepisovací pravidla; Apache Calcite; Apache Lucene; HeavyDB



## Abstract

Apache Calcite is a project that aims to create a modifiable relational optimizer database. Calcite allows you to combine different data models and different database systems into one optimizer and also offers the possibility to create custom transformation rules. The aim of this work is implement a transformation rule for window functions, which are increasingly part of regular SQL today questions.



## Keywords

transformation rules; Apache Calcite; Apache Lucene; HeavyDB

## Poděkování

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

# Obsah

**Seznam použitých symbolů a zkratek** **6**

**Seznam obrázků** **7**

**1 Úvod** **12**

**2 Apache Calcite** **13**
- 2.1 Části Apache Calcite . . . . . 14
- 2.2 Logický plán . . . . . 15
- 2.3 Fyzický plán . . . . . 15
- 2.4 Přepisovací pravidla . . . . . 16
- 2.5 Calcite adaptéry . . . . . 18
- 2.6 Určování ceny . . . . . 18
- 2.7 Heavy DB . . . . . 19
- 2.8 Window funkce . . . . . 19

**3 Implementace** **31**
- 3.1 Prostředí Apache Calcite . . . . . 31
- 3.2 Tvorba schématu a dat . . . . . 31
- 3.3 Dotazový procesor . . . . . 32
- 3.4 Přepisovací pravidlo . . . . . 33
- 3.5 Testování pravidla . . . . . 43

**4 Závěr** **52**

**Literatura** **53**

**Přílohy** **54**

**A Testovací dotazy pro tabulku *zaci*** **54**

**B Testovací dotazy pro tabulku *zaci_small*** **64**

**C Testovací dotazy pro databázi *TPC-H*** **74**

**D Testovací dotazy, kdy se nemá použít přepisovací pravidlo** **80**

**E Elektronická příloha** **82**

# Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| SQL | – | Structured Query Language |
| OLAP | – | Online Analytical Processing |
| GB | – | Gigabyte |
| MHz | – | Megahertz |
| RAM | – | Random Access Memory |
| SDK | – | Software Development Kit |
| csv | – | Comma Separated Values |
| CUDA | – | Compute Unified Device Architecture |
| TPC-H | – | Transaction Processing Performance Council - Benchmark H |

# Seznam obrázků


2.1 Firmy využívající Apache Calcite a Adaptéry Apache Calcite [1] . . . . . . . . . . . 14
2.2 Logický strom [2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.3 Tabulka pro komulativní četnost . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.4 Komulativní četnost žáků podle známek . . . . . . . . . . . . . . . . . . . . . . 20
2.5 Framing [8] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.6 Tabulka pro funkci *dense rank* . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.7 Výsledek pro window funkci *dense rank* . . . . . . . . . . . . . . . . . . . . . . 24
2.8 Výsledek pro window funkci *rank* . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.9 Výsledek pro získání největšího záznamu ze skupiny . . . . . . . . . . . . . . . . 26
2.10 Testování window funkce . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.11 Testování self-joinu . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.12 SQL Server window funkce vs self-join . . . . . . . . . . . . . . . . . . . . . . 28
2.13 Postgre SQL window funkce vs self-join . . . . . . . . . . . . . . . . . . . . . . 29
2.14 Heavy DB window funkce vs self-join . . . . . . . . . . . . . . . . . . . . . . . 29

3.1 Výstup programu . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.2 Plán A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.3 Plán B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.4 Diagram přepisovacího pravidla . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.5 Diagram druhého přepisovacího pravidla . . . . . . . . . . . . . . . . . . . . . . 42
3.6 Schéma tabulek zaci a zaci_small . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.7 Schéma TPCH databáze [12] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.8 Doba trvání dotazu self-joinu s přepisem a bez . . . . . . . . . . . . . . . . . . 45
3.9 Doba trvání dotazu s použitím pravidla a bez na velké databázi . . . . . . . . . . 45
3.10 HeavyDB logický plán pro window funkci . . . . . . . . . . . . . . . . . . . . . 47
3.11 HeavyDB logický plán pro self-join . . . . . . . . . . . . . . . . . . . . . . . . 47
3.12 Příklad konfigurace pro spuštění . . . . . . . . . . . . . . . . . . . . . . . . . 48

# Seznam kódů


2.1 SQL pro logický strom . . . 15
2.2 Pravidlo komutativity . . . 16
2.3 Pravidlo asociativity . . . 17
2.4 Komulativní součet . . . 20
2.5 Komulativní součet pomocí window funkcí . . . 20
2.6 Window funkce syntaxe . . . 21
2.7 DENSE_RANK() syntaxe . . . 23
2.8 DENSE_RANK() . . . 24
2.9 Přepsání *dense ranku* . . . 25
2.10 Dense rank pro získání největšího záznamu ze skupiny . . . 26
2.11 Self-join pro získání největšího záznamu ze skupiny . . . 26
2.12 Testovací dotaz pro window funkci . . . 27
2.13 Testovací dotaz pro self-join . . . 27
3.1 Dotaz A . . . 33
3.2 Dotaz B . . . 34
3.3 Definice přepisovacího pravidla . . . 36
3.4 onMatch funkce . . . 36
3.5 Vložení operátoru LogicalFilter . . . 37
3.6 Plán po vložení operátoru LogicalFilter . . . 37
3.7 Zobrazení aktuální sloupce přepisovacího pravidla . . . 38
3.8 Vložení operátoru LogicalProject . . . 38
3.9 Plán po vložení operátoru LogicalProject . . . 38
3.10 Vložení operátoru LogicalAgregate . . . 39
3.11 Plán po vložení operátoru LogicalAggregate . . . 39
3.12 Vložení operátoru LogicalTableScan . . . 39
3.13 Plán po vložení operátoru LogicalAggregate . . . 39
3.14 Vytvoření spojovacího podmínek pro LogicalJoin . . . 40
3.15 Vložení operátoru LogicalTableScan . . . 40
3.16 Plán po vložení operátoru LogicalJoin . . . 41

3.17 Vložení druhého operátoru LogicalProject . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.18 Plán po vložení druhého operátoru LogicalProject . . . . . . . . . . . . . . . . . . . . 41
3.19 Transformace původního plánu na nový . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.20 Vynucení přepisovacího pravidla . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.21 Testovací dotaz pro testování výkonnosti . . . . . . . . . . . . . . . . . . . . . . . . 45
3.22 HeavyDB inicializace přepisovacích pravidel . . . . . . . . . . . . . . . . . . . . . . 47
A.1 *zaci* Q1_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
A.2 *zaci* Q1_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
A.3 *zaci* Q2_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
A.4 *zaci* Q2_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
A.5 *zaci* Q3_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
A.6 *zaci* Q3_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
A.7 *zaci* Q4_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
A.8 *zaci* Q4_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
A.9 *zaci* Q5_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
A.10 *zaci* Q5_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
A.11 *zaci* Q6_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
A.12 *zaci* Q6_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
A.13 *zaci* Q7_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
A.14 *zaci* Q7_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
A.15 *zaci* Q8_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
A.16 *zaci* Q8_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
A.17 *zaci* Q9_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
A.18 *zaci* Q9_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
A.19 *zaci* Q10_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
A.20 *zaci* Q10_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
A.21 *zaci* Q11_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
A.22 *zaci* Q11_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
A.23 *zaci* Q12_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
A.24 *zaci* Q12_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
A.25 *zaci* Q13_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
A.26 *zaci* Q13_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
A.27 *zaci* Q14_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
A.28 *zaci* Q14_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
A.29 *zaci* Q15_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
A.30 *zaci* Q15_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
A.31 *zaci* Q16_A . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
A.32 *zaci* Q16_B . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

A.33 *zaci* Q17_A . . . 62
A.34 *zaci* Q17_B . . . 62
A.35 *zaci* Q18_A . . . 62
A.36 *zaci* Q18_B . . . 63
B.1 *zaci_small* Q1_A . . . 64
B.2 *zaci_small* Q1_B . . . 64
B.3 *zaci_small* Q2_A . . . 64
B.4 *zaci_small* Q2_B . . . 65
B.5 *zaci_small* Q3_A . . . 65
B.6 *zaci_small* Q3_B . . . 65
B.7 *zaci_small* Q4_A . . . 65
B.8 *zaci_small* Q4_B . . . 66
B.9 *zaci_small* Q5_A . . . 66
B.10 *zaci_small* Q5_B . . . 66
B.11 *zaci_small* Q6_A . . . 66
B.12 *zaci_small* Q6_B . . . 67
B.13 *zaci_small* Q7_A . . . 67
B.14 *zaci_small* Q7_B . . . 67
B.15 *zaci_small* Q8_A . . . 67
B.16 *zaci_small* Q8_B . . . 68
B.17 *zaci_small* Q9_A . . . 68
B.18 *zaci_small* Q9_B . . . 68
B.19 *zaci_small* Q10_A . . . 68
B.20 *zaci_small* Q10_B . . . 69
B.21 *zaci_small* Q11_A . . . 69
B.22 *zaci_small* Q11_B . . . 69
B.23 *zaci_small* Q12_A . . . 69
B.24 *zaci_small* Q12_B . . . 70
B.25 *zaci_small* Q13_A . . . 70
B.26 *zaci_small* Q13_B . . . 70
B.27 *zaci_small* Q14_A . . . 70
B.28 *zaci_small* Q14_B . . . 71
B.29 *zaci_small* Q15_A . . . 71
B.30 *zaci_small* Q15_B . . . 71
B.31 *zaci_small* Q16_A . . . 71
B.32 *zaci_small* Q16_B . . . 72
B.33 *zaci_small* Q17_A . . . 72
B.34 *zaci_small* Q17_B . . . 72

B.35 *zaci_small* Q18_A . . . . . 72
B.36 *zaci_small* Q18_B . . . . . 73
C.1 *TPC-H* Q1_A . . . . . 74
C.2 *TPC-H* Q1_B . . . . . 74
C.3 *TPC-H* Q2_A . . . . . 75
C.4 *TPC-H* Q2_B . . . . . 75
C.5 *TPC-H* Q3_A . . . . . 75
C.6 *TPC-H* Q3_B . . . . . 76
C.7 *TPC-H* Q4_A . . . . . 76
C.8 *TPC-H* Q4_B . . . . . 76
C.9 *TPC-H* Q5_A . . . . . 77
C.10 *TPC-H* Q5_B . . . . . 77
C.11 *TPC-H* Q6_A . . . . . 77
C.12 *TPC-H* Q6_B . . . . . 78
C.13 *TPC-H* Q7_A . . . . . 78
C.14 *TPC-H* Q7_B . . . . . 78
C.15 *TPC-H* Q8_A . . . . . 79
C.16 *TPC-H* Q8_B . . . . . 79
D.1 nepoužití přepisovacího pravidla Q1 . . . . . 80
D.2 nepoužití přepisovacího pravidla Q2 . . . . . 80
D.3 nepoužití přepisovacího pravidla Q3 . . . . . 81
D.4 nepoužití přepisovacího pravidla Q4 . . . . . 81
D.5 nepoužití přepisovacího pravidla Q5 . . . . . 81
D.6 nepoužití přepisovacího pravidla Q6 . . . . . 81

# Kapitola 1

# Úvod

V dnešní době jsou databáze nezbytným prvkem mnoha aplikací a systémů, z tohoto důvodu je důležité optimalizovat dotazy na databázi tak, aby byly co nejefektivnější a nejrychlejší. Jedním ze způsobů, jak toho dosáhnout, je pomocí přepisování dotazů.

Apache Calcite je open-source framework pro optimalizaci dotazů, který umožňuje vytvářet přepisovací pravidla pro transformaci dotazů. Tato pravidla mohou být vytvořena pro různé typy dotazů a mohou být použita pro optimalizaci různých databázových systémů.

Cílem této diplomové práce je vytvořit nové přepisovací pravidlo pro Apache Calcite, které umožní efektivní přepis dotazů obsahujících window funkce.

V 2. kapitole bude popsáno prostředí Apache Calcite a jeho fungování. Bude se také zabývat optimalizací dotazu včetně přepisovacích pravidel a tvorby plánů. Na konci kapitoly bude podrobně vysvětlena problematika window funkcí a jejich fungování.

Kapitola 3 se zaměří na technické prostředí Apache Calcite a implementaci nového přepisovacího pravidla. Práce bude ukazovat, jak vytvořit databázový procesor v Apache Calcite a co vše musí být obsaženo v novém pravidle. Dále bude podrobně popsána tvorba samotného pravidla včetně jednotlivých částí. Na závěr této kapitoly bude prezentováno testování pravidla na umělých i reálných datech.

# Kapitola 2

# Apache Calcite

Apache Calcite je open-source framework vytvořený v roce 2014 firmou Apache Software Foundation.

Apache Calcite slouží jako rozšiřitelný parser a optimalizátor SQL dotazů. Narozdíl od klasické SQL databáze, jako je třeba Microsoft SQL Server, který pracuje jen s jedním typem uložení dat, nad kterým následně provádí dotazování. Apache Calcite jako framework nemá definované ukládání dat, algoritmy pro zpracování dat a uložiště pro ukládání metadat. To nám dovoluje po použití adaptérů mít databázový systém, který umí pracovat s jakýmkoliv datovým uložištěm. Adaptér je zde označen jako rozšíření Apache Calcite, které se stará o datové uložiště a práci s ním. Pro Apache Calcite existuje mnoho adaptérů, jako je například CSV, ElasticSearch, MongoDB a mnoho dalších adaptérů (obrázek 2.1), každý z nich zpracovává jiný typ dat. Pokud pro naše uložiště neexistuje oficiální adaptér, je možné si ho vytvořit přesně na míru. V této práci se bude využívat prostředí Apache Calcite k vytvoření vlastního přepisovacího pravidla.

V současnosti Apache Calcite používají významné firmy, jako je Flink, Storm Drill a další velké firmy, které využívají jeho výhody. Tyto firmy můžeme vidět na obrázku 2.1.

Obrázek 2.1: Firmy využívající Apache Calcite a Adaptéry Apache Calcite [1]

## 2.1 Části Apache Calcite

Apache Calcite se skládá ze tří hlavních částí:

- **Parser** - slouží k interpretaci SQL dotazů. Rozdělí SQL dotaz na atomické části, které nazýváme *relační operátory*. Tyto relační operátory jsou uspořádány do stromu nazývající se *logický plán*. Detailnější popis logického plánu je popsán v následující kapitole 2.2. Jednotlivé relační operátory jsou i validovány, to znamená, že parser kontroluje, jestli SQL dotaz obsahuje validní syntaxi a jestli proměnné v SQL dotazu (názvy tabulek, sloupců,...) existují ve schématu databáze.

- **Optimalizátor** - má na starost vytvořit takzvaný *plán* z relačních operátorů získaných z parseru, který se bude blížit tomu nejefektivnějšímu. K tomu mu dopomáhají přepisovací pravidla, která jsou popsána v kapitole 2.4

- **Adaptér** - slouží k připojení k různým zdrojům dat, například k relační databázi, NoSQL databázi, csv souborům, Hadoop a dalším. Příklady adaptérů a jejich popis jsou popsány v kapitole 2.5. Tato část je hlavním rozdílem od klasické databáze. Klasická databáze totiž umí

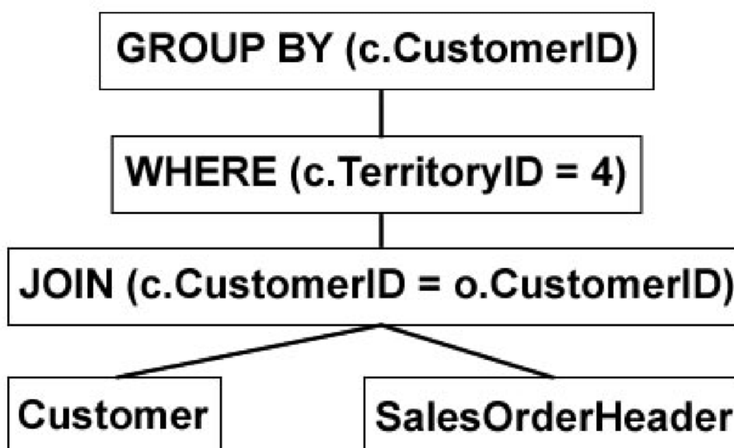
pracovat jen s jedním zdrojem dat, který se nedá měnit. V Apache Calcite můžeme pracovat i s více datovými zdroji najednou.

## 2.2 Logický plán

Jak už jsem výše zmínil, parser vytváří **logický plán**, což je strom **relačních operátorů**. Je to logická reprezentace toho, jak provést dotaz, aniž by se brala v úvahu fyzická implementace dotazu. Je to základní rozdělení dotazu na jednotlivé relační operátory, jako je třeba join, zvolení tabulky, podmínka *where* a podobně. Nebere se zde v potaz fyzická implementace jednotlivých relačních operátorů. Vizuální příklad, jak se SQL dotaz převede do logického stromu můžeme vidět na obrázku 2.2. [2]

```
1  SELECT c.CustomerID, COUNT(*)
2  FROM Sales.Customer c JOIN Sales.SalesOrderHeader o
3  ON c.CustomerID = o.CustomerID
4  WHERE c.TerritoryID = 4
5  GROUP BY c.CustomerID
```

Kód 2.1: SQL pro logický strom




Obrázek 2.2: Logický strom [2]

## 2.3 Fyzický plán

Fyzický plán se liší od logického plánu tím, že u fyzického plánu bereme v úvahu fyzickou implementaci dotazu. To znamená, každý relační operátor ve fyzickém plánu je již konkrétní algoritmus s jasně definovanými výstupy pro dané vstupy.

Například v logickém plánu budeme mít relační operátor join, který reprezentuje spojení dvou tabulek. Nedefinuje již jakým konkrétním způsobem se provede spojení. Ve fyzickém plánu tento relační operátor již bude představovat *nested loop join*, *merge join*, nebo *hash join*, které již definují přesný algoritmus pro spojení dvou tabulek.

Pro každý relační operátor ve fyzickém plánu určuje optimalizátor jeho odhadovanou cenu. Pro logický plán se cena nedá odhadnout, jelikož cena se dá odhadnout, jen pokud víme přesný algoritmus operátoru. Ohledně určování cen pro relační operátory je kapitola 2.6.

## 2.4 Přepisovací pravidla

Výše jsem zmiňoval, že nám parser vytvoří logický plán. Z tohoto logického plánu lze vytvořit několik ekvivalentních jak logických plánů, tak fyzických plánů. Příklady můžeme vidět v kapitole 2.4.1 a 2.4.2. Cílem optimalizátoru je vytvořit co nejvíce ekvivalentních plánů a vybrat plán, který bude mít nejmenší cenu. Optimalizátor může vytvořit nový plán, který se bude lišit jen v jednom relačním operátoru nebo taky se může lišit úplně celý původní plán, vždy oba plány musí být ekvivalentní a vracet stejný výsledek.

V Apache Calcite existují dva hlavní typy přepisovacích pravidel:

- **Logické přepisovací pravidlo** - slouží k přepsání logického plánu nebo jeho části na jiný ekvivalentní logický plán nebo jeho část.
- **Fyzické přepisovací pravidlo** - slouží k přepsání logického relačního operátoru na fyzický relační operátor a díky tomu přepisuje logický plán na fyzický plán.

### 2.4.1 Příklady logického přepisovacího pravidla

Příkladem logického přepisovacího pravidla může být **pravidlo komutativity a asociativity**, která se používají při použití operátoru join při spojení dvou tabulek.

**Pravidlo komutativity** znamená, že *A join B* je ekvivalentní *B join A* a že spojení tabulek A a B v jakémkoliv směru bude vracet stejný výsledek, jak můžeme vidět v kódu 2.2. Všimněme si také, že pokud uplatníme toto pravidlo 2x, tak nám vygeneruje původní výraz. To znamená, pokud použijeme tuto transformaci, tak získáme *B join A*. Pokud bychom ale použili znovu stejnou transformaci, tak získáme znovu *A join B*. Optimalizátor nicméně umí vyřešit tento problém, aby se vyhnulo duplicitním výrazům.

```
1    A join B <=> B join A
```

Kód 2.2: Pravidlo komutativity

Ve stejném smyslu funguje i **pravidlo asociativity**. Pravidlo ukazuje, že *(A join B) join C* je ekvivalentní výrazu *A join (B join C)*, kde oba výrazy opět vrací stejný výsledek, který bude vypadat jako v kódu 2.3.

```
1    (A join B) join C <=> A join (B join C)
```

Kód 2.3: Pravidlo asociativity

Při použití těchto pravidel budeme mít sice ekvivalentní výrazy, ale při implementaci fyzického výrazu pro *join* se cena nákladů na výpočet může lišit na bázi toho, jestli použijeme logickou strukturu *A join B* nebo *B join A* a podobně.

[3]

### 2.4.2 Příklady fyzického přepisovacího pravidla

Příkladem fyzického přepisovacího pravidla může být přepsání logického relačního operátoru *join* na fyzický relační operátor. Logický relační operátor *join* se dá přepsat na 3 různé fyzické relační operátory *join*. Každý z těchto fyzických *joinů* implementuje jiný algoritmus pro spojení dvou tabulek. Důležité je si uvědomit, že žádný fyzický *join* není lepší než ostatní. Vždy záleží na konkrétní situaci, jak vysvětluji níže [3]:

- **Nested loop join** - používá jednoduchý algoritmus. Mějme tento jednoduchý dotaz A join B. Tabulka A je vnitřní vstup a tabulka B vnější vstup. Pro každý řádek ve vnitřním vstupu se vyhledá odpovídající řádek z vnějšího vstupu.

  Používá se zejména, když je jeden vstup velmi malý a vyhledání v druhém vstupu je rychlé.

- **Merge join** - vyžaduje, aby obě tabulky byly seřazeny podle slučovací podmínky *joinu*. Výhodou toho, že obě tabulky jsou seřazené je to, že se join vyřeší jedním průchodem oběma tabulkami.

  Používá se převážně, když na obě tabulky existuje index na slučovací podmínku. Díky tomu je pak jednoduché seřadit obě tabulky, což je kritérium pro tento algoritmus.

- **Hash join** - má podobné rysy jako Merge join. Obě tabulky se zde projdou rovněž jen jednou. Ale na rozdíl od Merge join, Hash join nevyžaduje, aby tabulky byly seřazeny. Hash join funguje tak, že se vytvoří **Hash tabulka** v paměti. Optimalizátor následně použíje odhad mohutnosti pro zjištění menší tabulky, z které následně vytvoří Hash tabulku. V Hash tabulce jsou jen nezbytné informace pro *join*. To je id řádku a hodnota pro spojení. Poté co je vytvořena a naplněna hash tabulka, druhá tabulka, nazvěme ji prozkoumávaná tabulka, bude porovnávaná s hash tabulkou jako u Nested loop joinu. Pokud se řádky budou rovnat, tak budou vráceny. Po spojení tabulek se Hash tabulka smaže.

  Na rozdíl od nested loop joinu nemusíme tolikrát procházet celou tabulku s mnoha sloupci, ale bude procházet zmenšenou dočasnou tabulkou se dvěma sloupci. Používá se převážně při spojování velkých tabulek.

## 2.5 Calcite adaptéry

V Apache Calcite jsou adaptéry komponenty, které převádí databázový systém na univerzální relační model, který je používaný pro zpracování dotazů. Adaptéry poskytují rozhraní pro připojení k různým databázovým systémům a umožňují provádět dotazy na datech v těchto systémech.

Adaptér má na starost dodat Apache Calcite schéma databáze, statistiky a fyzické relační operátory, které komunikují již s určitým databázovým systém.

Pro Apache Calcite existuje mnoho oficiálních i neoficiálních adeptérů. Například [4]:

- **CSV adaptér** - data jsou uložené v CSV souborech.
- **MongoDB adaptér** - propojení s MongoDB databází, která zpracuje dotaz.
- **Spark adaptér** - propojení s Apache Spark.
- **Lucene adaptér** - data jsou uložené v Lucene indexu.
- **JDBC adaptér** - propojení s JDBC API.

Apache Calcite nemusí vždy provést celý dotaz. Některé systémy používají jen některé části Apache Calcite. Například některé systémy využívají jen parser, jiné používají i optimalizátor plánu od Apache Calcite.

## 2.6 Určování ceny

Z předchozí kapitoly víme, že jeden SQL dotaz dokážeme přepsat do několika různých variant/plánů. Tyto plány se mezi sebou musí porovnat a vybrat plán, který bude cenově nejefektivnější. Cena dotazu se v databázích odhaduje pomocí těchto kritérií:

- Počet přístupů na disk
- Doba provedení, kterou CPU potřebuje k provedení dotazu
- Náklady na komunikaci v distribuovaných nebo paralelních databázových systémech

Cena dotazu před provedením je vždy jen odhad a může se lišit od finální ceny. Vypočítání přesné ceny by nebylo efektivní vzhledem k výkonu.

Apache Calcite moduluje cenu jako skalár představující počet zpracovávaných řádků. To znamená, že plán s nejmenší cenou bude plán, který bude pracovat s co nejméně řádky v databázi.

Ne všechny adaptéry Apache Calcite mají implementovanou tvorbu statistik, díky kterým dokážeme získat počet zpracovávaných řádků. Pro lepší práci optimalizátoru je uživatel nucen tuto funkcionalitu sám naimplementovat. Když tvorba statistik není implementovaná, Apache Calcite dává pevnou konstantu 100 jako počet zpracovaných řádků pro každý objekt, se kterými pracuje. [5]

## 2.7 Heavy DB

Heavy DB je databázový systém, který využívá komponenty z Apache Calcite. To konkrétně parser a optimalizátor. Tento databázový systém je navržen pro využití masivního paralelismu pomocí CPU a GPU. Je dělaný pro zpracování velkého množství dat. Tento databázový systém je v téhle práci vybrán z důvodu, že má implementovaný optimalizátor z Apache Calcite, tím pádem není problém ho rozšířit o nové přepisovací pravidlo v rámci optimalizátoru. Ale dalším důležitým kritériem je, že oproti základní verzi Apache Calcite se základním adaptérem, jako je csv adaptér nebo lucene adaptér, tento databázový systém je optimalizovaný pro reálnou práci. U základní verze Apache Calcite jsou všechny operace pomalé a z toho důvodu přepisovací pravidlo, které jsem vytvořil, vytváří pomalejší plán, než byl původní. U Heavy DB nebo ostatních enterprise databází postavených na Apache Calcite mé pravidlo by mělo být výhodné použít. [6]

## 2.8 Window funkce

Window funkce, které jsou také známy jako analytické OLAP funkce, jsou konstrukcí jazyka SQL, která je součástí SQL standartu. Window funkce dovolují elegantně vyjádřit mnoho užitečných typů dotazů, které by se vyjadřovaly jiným způsobem těžkopádně. Například komulativní četnost.

Výhodou window funkcí je provádět složité výpočty a analyzovat data v relačních databázových systémech s vysokou účinností a s menším množstvím kódu. Využití window funkcí nám mnohdy i zrychlí provedení dotazu.

Výše jsem zmínil, že window funkce elegantněji vyjadřuje dotaz na komulativní četnost. Pro příklad mějme tabulku, která obsahuje seznam žáků a jejich známky, a požadujeme získání komulativní četnosti známek, kterou můžeme vidět na obrázku 2.4.

| Žák | Známka z matematiky |
|---|---|
| Adam | 1 |
| Božena | 1 |
| Cyril | 2 |
| David | 2 |
| Emil | 1 |
| Filip | 3 |
| Gábina | 1 |
| Helena | 1 |
| Ivan | 4 |
| Jan | 4 |
| Karel | 1 |
| Ludvík | 2 |
| Marie | 3 |
| Nela | 1 |
| Ondra | 1 |
| Petr | 2 |

Obrázek 2.3: Tabulka pro komulativní četnost

| Známka | Četnost | Kumulativní četnost | |
|---|---|---|---|
| 1 | 8 | 8 | |
| 2 | 4 | 12 | = 8 + 4 |
| 3 | 2 | 14 | = 8 + 4 + 2 |
| 4 | 2 | 16 | = 8 + 4 + 2 + 2 |
| Celkem | 16 | | |

Obrázek 2.4: Komulativní četnost žáků podle známek

Tento výsledek můžeme získat dotazem bez použití window funkce následovně (musíme předpokládat, že tabulka žáků obsahuje numerický primární klíč id. Bez něj bychom nebyli schopni docílit žádaného výsledku):

```
1 SELECT t1.zak, t1.znamka, SUM(t2.znamka) AS comulative_sum
2 FROM zaci t1
3 INNER JOIN zaci t2 on t1.id() >= t2.id()
4 GROUP BY t1.zak, t1.znamka
5 ORDER BY t1.zak
```

Kód 2.4: Komulativní součet

Pomocí window funkcí můžeme výsledek získat tímto dotazem:

```
1   SELECT zak, znamka, sum(znamka) OVER (PARTITION BY znamka ORDER BY zak) AS
       comulative_sum
2   FROM zaci
```

Kód 2.5: Komulativní součet pomocí window funkcí

Dotaz pomocí window funkce je zde jednak přehlednější, tak i úspornější, co se týče databázového výkonu. Bez použití window funkce, musíme v dotazu použít různé poddotazy nebo joiny, které nemusí být nejefektivnější. Podrobnější popis fungování a struktury window funkcí popisuji v kapitole 2.8.1. [8]

### 2.8.1 Struktura a fungování window funkcí

Jeden z hlavních principů SQL je, že výstupní pořadí záznamů (mimo využití klauzule *order by*) není definováno. To dovoluje vytvářet mnoho důležitých optimalizací, ale kvůli toho tvorba závislých dotazů na pořadí je poměrně obtížná. Tím, že se umožní odkazovat na sousední záznamy přímo, window funkce umožňuje snadno vyjádřit takové dotazy.

Pro pochopení fungování window funkcí jsou důležité dva body:

- Window funkce se provádí po většině ostatních klauzulích (zahrnující i *group by* a *having*), ale před konečným *order by* (myšleno *order by* mimo window funkci) a odstraněním duplicit pomocí *distinct*.

- Window funkce nám modifikuje hodnotu aktuálního záznamu, ale nemůže přidat nebo odebrat záznam. Počet záznamů zůstává stejný po použití window funkce.

Proto window funkce je povolena být v klauzuli *select* nebo *order by*. Ale již nemůže být window funkce v podmínce *where*.

Syntaxe window funkce se vyznačuje slovem **over** a obsahuje klauzule **partition by**, **order by** a **rows** nebo **range**. Jak můžeme vidět v kódu 2.6.

```
1   OVER (
2      [ <PARTITION BY clause> ]
3      [ <ORDER BY clause> ]
4      [ <ROW or RANGE clause> ]
5   )
```

Kód 2.6: Window funkce syntaxe

[9]

- **Rozdělení** (*partition by*) - Tato klauzule rozdělí vstup jedním, nebo více výrazy na nezávislé skupiny. Na rozdíl od klasické agregace *group by*, window funkce neredukuje všechny záznamy ve skupině na jeden záznam, ale pouze logicky rozděluje záznamy do určitých skupin. Pokud rozdělení není definované, považuje se, že všechny záznamy patří do jedné skupiny.

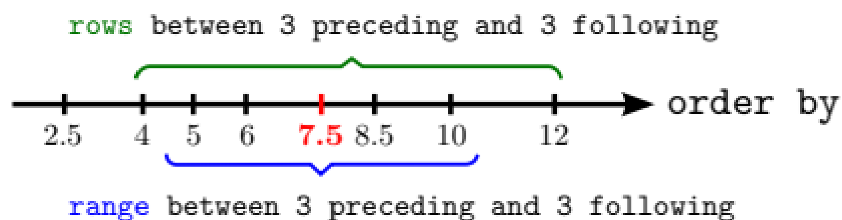
- **Řazení** (*order by*) - V rámci skupiny z předchozího bodu nám klauzule *order by* definuje pořadí ve skupině během vyhodnocování window funkce. Tato funkce ovlivňuje jen pořadí ve výpočtu window funkce a ne nutně konečné pořadí výsledku. Není-li zadané řazení, výsledky některých window funkcí se stanou nedeterministické.

- **Ohraničení** (*framing*) - Kromě klauzule pro rozdělení má window funkce klauzuli *framing*. Ta definuje hranice aktuální skupiny. Pomocí této klauzuly můžeme brát v potaz v aktuální skupině i záznamy z jiné skupiny.

Tato klauzule má 2 režimy, které můžeme vidět na obrázku 2.5.

– **Řádky** (*rows*): Režim pro řádky přímo určuje, kolik řádků/záznamů před nebo za aktuálním záznamem je součástí ohraničení. Na obrázku 2.5 je nastavená hranice pro 3 záznamy před a za aktuálním záznamem. Proto se skládá z hodnot 4, 5, 6, 7.5, 8.5, 10 a 12. Lze také zadat ohraničení, který nebude zahrnovat aktuální záznam, který je zde 7.5.

– **Rozsah** (*range*): Režim pro rozsah je počítán tak, že se vezme aktuální záznam a od něj se odečte anebo přičte definovaná hodnota. Zde konkrétně je aktuální záznam 7.5 a od něj se odečte a přičte hodnota 3. Tak získáme dolní hranici 4.5 a horní hranici 10.5. Proto tento *frame* obsahuje hodnoty 5, 6, 7.5, 8.5 a 10.




Obrázek 2.5: Framing [8]

Kromě nastavení statických hodnot se můžou nastavit do klauzulí tyto hodnoty:

– **Aktuální řádek** (*current row*): nastaví se aktuální řádek.

– **Neomezený předchozí** (*unbounded preceding*): nastaví se první záznam ze skupiny.

– **Neomezený následující** (*unbounded following*): nastaví se poslední záznam ze skupiny.

Pokud ohraničení není nastavené a je nastavené řazení (*order by*), výchozí hodnota pro ohraničení je: *range between unbounded preceding and current row*. To nám zapříčiní ohraničení, které je konzistentní od prvního záznamu ve skupině až po aktuální záznam, což je užitečné pro počítání komulativní četnosti, kterou jsem ukazoval na obrázku 2.5. Dotazy bez řazení (*order by*) jsou počítány, jako by měly ohraničení nastavené *range between unbounded preceding and unbounded following*. Ohraničení má efekt jen na některé window funkce.

[8]

### 2.8.2 Druhy window funkcí

V SQL se definuje řada window funkcí pro různé účely, ty se dají rozdělit do těchto kategoriích:

- **Agregační funkce** - Tyto funkce slouží k výpočtu souhrnných hodnot z řady záznamů. Mezi funkce patří například AVG(), SUM(), MAX() a MIN().
- **Rank funkce** - Tyto funkce vám umožňují získat informace o pořadí záznamů, například RANK(), DENSE_RANK() a ROW_NUMBER().
- **Value funkce** - Tyto funkce vám umožňují získat informace o souvisejících záznamech, například LAG() pro předchozí záznam a LEAD() pro následující záznam.

### 2.8.3 DENSE_RANK() funkce

Pro mou diplomovou práci jsem vytvořil přepisovací pravidlo, které přepisuje dotaz obsahující window funkci *dense rank*. V této kapitole popíšu strukturu a fungování *DENSE_RANK* window funkce.

Tato funkce má za úkol vypočítat **pořadí** pro každý záznam ve skupině, která se rozděluje pomocí klauzule *partition by*. Tato funkce ignoruje ohraničení (*framing*). Syntaxe pro *dense rank* vypadá následovně:

```
1   DENSE_RANK() OVER (PARTITION BY column1, column2 ORDER BY column3 DESC) AS
        rank_value
```

Kód 2.7: DENSE_RANK() syntaxe

Pro lepší pochopení předvedu fungování window funkce *dense rank* na příkladu. Mějme tabulku žáků, která obsahuje jméno žáka, třídu a počet bodů z olympiády. Bude vypadat následovně:

| Žák | Třída | Body |
|---|---|---|
| Adam | 6 | 55 |
| Božena | 6 | 81 |
| Cyril | 7 | 67 |
| David | 9 | 59 |
| Emil | 6 | 81 |
| Filip | 9 | 71 |
| Gábina | 8 | 63 |
| Helena | 8 | 69 |
| Ivan | 9 | 83 |
| Jan | 6 | 71 |
| Karel | 8 | 91 |
| Ludvík | 7 | 50 |
| Marie | 7 | 61 |
| Nela | 9 | 81 |
| Ondra | 7 | 93 |
| Petr | 8 | 77 |

Obrázek 2.6: Tabulka pro funkci *dense rank*

Chceme-li získat pořadí žáků podle počtu bodů, které dostali v olympiádě a zároveň chceme mít pořadí zvlášť pro každou třídu, tak tuto reprezentaci můžeme napsat s využitím window funkce *dense rank* tímto způsobem:

```
1  SELECT
2     jmeno,
3     trida,
4     body,
5     DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) as poradi
6  FROM zaci;
```

Kód 2.8: DENSE_RANK()

| Žák | Třída | Body | Rank |
|---|---|---|---|
| Emil | 6 | 81 | 1 |
| Božena | 6 | 81 | 1 |
| Jan | 6 | 71 | 3 |
| Adam | 6 | 55 | 4 |
| Ondra | 7 | 93 | 1 |
| Cyril | 7 | 67 | 2 |
| Marie | 7 | 61 | 3 |
| Ludvík | 7 | 50 | 4 |
| Karel | 8 | 91 | 1 |
| Petr | 8 | 77 | 2 |
| Helena | 8 | 69 | 3 |
| Gábina | 8 | 63 | 4 |
| Ivan | 9 | 83 | 1 |
| Nela | 9 | 81 | 2 |
| Filip | 9 | 71 | 3 |
| David | 9 | 59 | 4 |

Obrázek 2.7: Výsledek pro window funkci *dense rank*

Tento dotaz 2.8 přidělí každému žákovi pořadí v olympiádě podle třídy, což je vyobrazeno na obrázku 2.7. Můžeme vidět, že Božena a Emil z 6. třídy mají stejný počet bodů. Proto při použití window funkce *dense rank* je následující žák Jan na třetí pozici. Tím pádem se zde přeskočí druhá pozice. Pokud bychom nechtěli, aby se přeskakovaly pozice, tak bychom použili místo window funkce *dense rank* window funkci *rank*, která by vrátila výsledek na obrázku 2.8. Obě funkce jsou vhodné pro jiné situace a je na uživateli, jakou funkci použije.

| Žák | Třída | Body | Rank |
|---|---|---|---|
| Emil | 6 | 91 | 1 |
| Božena | 6 | 81 | 1 |
| Jan | 6 | 71 | 2 |
| Adam | 6 | 55 | 3 |
| Ondra | 7 | 93 | 1 |
| Cyril | 7 | 67 | 2 |
| Marie | 7 | 61 | 3 |
| Ludvík | 7 | 50 | 4 |
| Karel | 8 | 91 | 1 |
| Petr | 8 | 77 | 2 |
| Helena | 8 | 69 | 3 |
| Gábina | 8 | 63 | 4 |
| Ivan | 9 | 83 | 1 |
| Nela | 9 | 81 | 2 |
| Filip | 9 | 71 | 3 |
| David | 9 | 59 | 4 |

Obrázek 2.8: Výsledek pro window funkci *rank*

### 2.8.4 Nevýhody window funkcí

U některých dotazů, které využívají window funkce se velmi obtížně vytváří ekvivalentní dotaz bez použití window funkcí. Mnohdy je takový ekvivalent i výkonnostně mnohonásobně horší a nedává smysl nevyužít window funkce. Například, když vezmeme z minulé kapitoly na *dense rank* kód 2.8, který přepíšeme na následující ekvivalentní dotaz:

```
1  SELECT zaci.jmeno, zaci.trida, zaci.body,
2     (SELECT COUNT(DISTINCT body)
3     FROM zaci zaci2
4     WHERE zaci2.trida = zaci.trida AND
5        zaci2.body >= zaci.body
6     ) AS poradi
7  FROM zaci;
```

Kód 2.9: Přepsání *dense ranku*

Tento přepis dotazu nedává smysl, jelikož není možné, aby přepsaný SQL dotaz byl rychlejší, spíše bude pomalejší a to i několikanásobně. Testoval jsem oba dotazy na 10 milionech záznamech, kde window funkce doběhla v rámci řádu sekund a dotaz s poddotazem 2.9 nedoběhl ani o 5 hodin později, kdy jsem byl nucen zpracování dotazu zastavit. Testování bylo provedeno nad databázových systémech SQL Server, Postgre SQL a Heavy DB. Popis testovacího prostředí a verzí databázových systémů popíšu později v této kapitole.

U takového dotazu nemá smysl pokoušet se ho sestavit bez window funkcí, ale jak existují dotazy, u kterých je jednoznačně lepší použití window funkcí, tak existují dotazy, kde je použití window funkcí znatelně pomalejší, a to například, když chceme najít žáky s nejvyšším počtem bodů ve své třídě. Pro stejná data, jako jsou na obrázku 2.6 chceme získat následující výsledek:

| Žák | Třída | Body | Rank |
| --- | --- | --- | --- |
| Emil | 6 | 91 | 1 |
| Božena | 6 | 81 | 1 |
| Ondra | 7 | 93 | 1 |
| Karel | 8 | 91 | 1 |
| Ivan | 9 | 83 | 1 |

Obrázek 2.9: Výsledek pro získání největšího záznamu ze skupiny

Tento výsledek jsme schopni získat pomocí window funkce 2.10 nebo pomocí self-joinu 2.11.

```
1  SELECT *
2  FROM (
3     SELECT *,
4        DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5     FROM zaci
6  ) ranking
7  WHERE rank_value = 1;
```

Kód 2.10: Dense rank pro získání největšího záznamu ze skupiny

```
1  SELECT c.*
2  FROM zaci c
3  JOIN (
4     SELECT trida, MAX(body) rank_value
5     FROM zaci
6     GROUP BY trida
7  ) ranking ON c.trida = ranking.trida AND
8  c.body = ranking.min_payment;
```

Kód 2.11: Self-join pro získání největšího záznamu ze skupiny

Oba dotazy jsou ekvivalentní a vrací stejný výsledek, ale použití window funkce zde může být znatelně pomalejší.

#### 2.8.4.1 Testovací prostředí

Pro tento druh dotazu jsem provedl detailnější testování nad třemi databázovými systémy, kde jeden obsahuje komponenty z Apache Calcite. Testování bylo provedeno nad těmito databázovými systémy:

- **SQL Server** - verze 15.0.2000 běžící na Windows 11. Čas byl získáván pomocí komponenty *Client Statistics*, která získává *execution time*, který udává dobu provedení dotazu v milisekundách.
- **Postgre SQL** - verze 15.2 běžící na Windows 11. Čas byl získáván pomocí *EXPLAIN ANALYSE* klauzule, která získává údaj *Execution Time*, který udává dobu provedení dotazu v milisekundách.

- **HeavyDB** (Apache Calcite) - verze Core 6.4.3 Free edition bežící na Ubuntu 20.04. HeavyDB je databáze, která využívá komponenty parseru a optimalizátoru z Apache Calcite. Čas získáváme z webového rozhraní SQL editoru, kde se zobrazuje čas provedení dotazu.

Všechny zmíněné databázové systémy používaly procesor intel I7-9750H a měly k dispozici 16GB RAM paměti o frekvenci 2666MHz. Každé měření bylo provedeno 10x a výsledný čas je průměr celého měření. Testováno bylo provedeno na 10 milionech náhodně vygenerovaných záznamech. CSV soubor s testovacími daty můžeme nalézt v příloze E.

#### 2.8.4.2 Testovací dotazy

Testovací dotaz zde musel být upraven od dotazu 2.10 a 2.11. Byla přidána agregace do finálního výsledku, aby dotaz vracel jeden řádek, dále byla přidána podmínka na sloupec body, podle které určuji selektivitu dotazu. Hodnota podmínky na sloupec body odpovídá počtu řádků, který dotaz vrací. Upravený dotazy pro testování můžeme vidět v kódech 2.12 a 2.13.

```
1    SELECT sum(body)
2    FROM (
3       SELECT
4          *,
5          DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
6       FROM zaci
7       WHERE body <= @variable
8    ) ranking
9    WHERE rank_value = 1;
```

Kód 2.12: Testovací dotaz pro window funkci

```
1    SELECT sum(body)
2    FROM zaci c
3    JOIN (
4       SELECT trida, MAX(body) rank_value
5       FROM zaci
6       WHERE body <= @variable
7       GROUP BY trida
8    ) ranking ON c.trida = ranking.trida AND
9    c.body = ranking.rank_value;
```

Kód 2.13: Testovací dotaz pro self-join

#### 2.8.4.3 Výsledek testování

Výsledky testování pro window funkci se nachází na obrázku 2.10, testování pro self join můžeme vidět na obrázku 2.11.

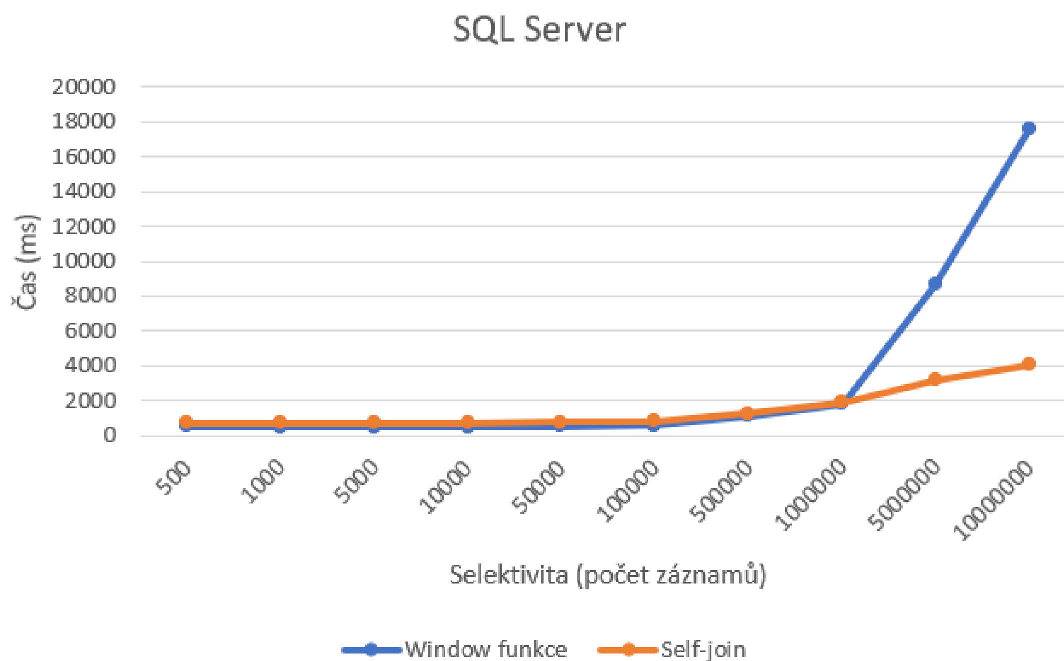
| Selektivita (počet záznamů) | 500 | 1000 | 5000 | 10000 | 50000 | 100000 | 500000 | 1000000 | 5000000 | 10000000 |
|---|---|---|---|---|---|---|---|---|---|---|
| | čas (ms) | | | | | | | | | |
| **SQL Server** | 518 | 502 | 506 | 507 | 541 | 590 | 1121 | 1822 | 8648 | 17557 |
| **Postgre SQL** | 576 | 547 | 551 | 503 | 541 | 593 | 673 | 907 | 5523 | 11182 |
| **HeavyDB** | 103 | 94 | 74 | 79 | 92 | 111 | 328 | 585 | 3488 | 6398 |
| **Výsledek** | 4448 | 8945 | 44946 | 89928 | 449934 | 899931 | 4499917 | 8999936 | 44999950 | 89999916 |

Obrázek 2.10: Testování window funkce

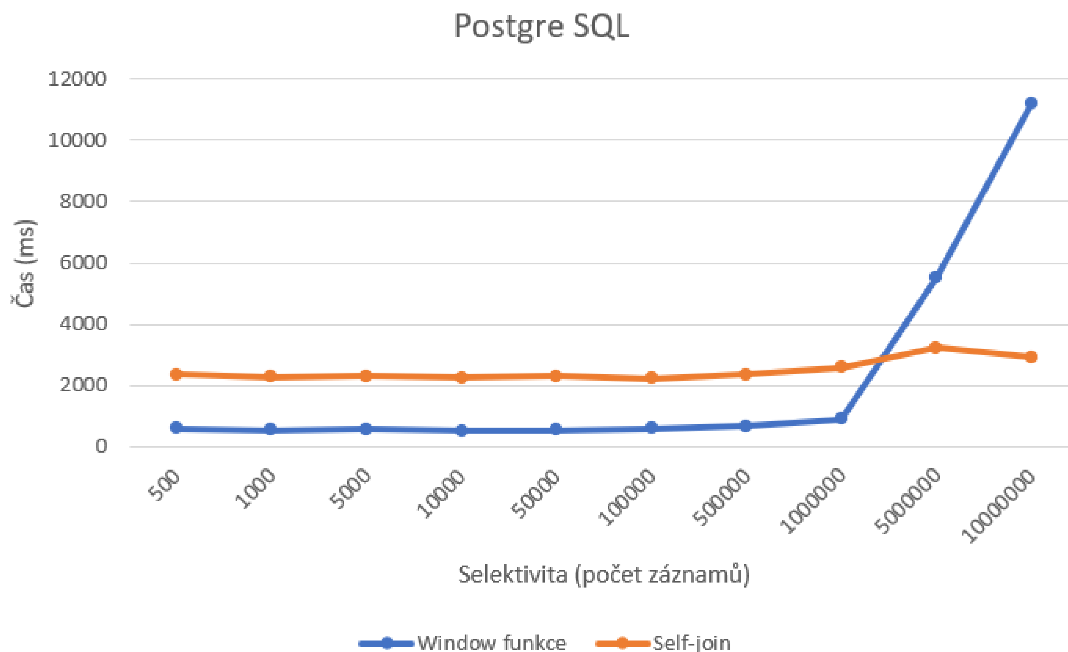
| Selektivita (počet záznamů) | 500 | 1000 | 5000 | 10000 | 50000 | 100000 | 500000 | 1000000 | 5000000 | 10000000 |
|---|---|---|---|---|---|---|---|---|---|---|
| | čas (ms) | | | | | | | | | |
| **SQL Server** | 722 | 727 | 728 | 713 | 768 | 826 | 1250 | 1918 | 3177 | 4041 |
| **Postgre SQL** | 2353 | 2281 | 2305 | 2240 | 2314 | 2236 | 2346 | 2591 | 3233 | 2909 |
| **HeavyDB** | 486 | 518 | 493 | 510 | 510 | 506 | 486 | 475 | 562 | 626 |
| **Výsledek** | 4448 | 8945 | 44946 | 89928 | 449934 | 899931 | 4499917 | 8999936 | 44999950 | 89999916 |

Obrázek 2.11: Testování self-joinu

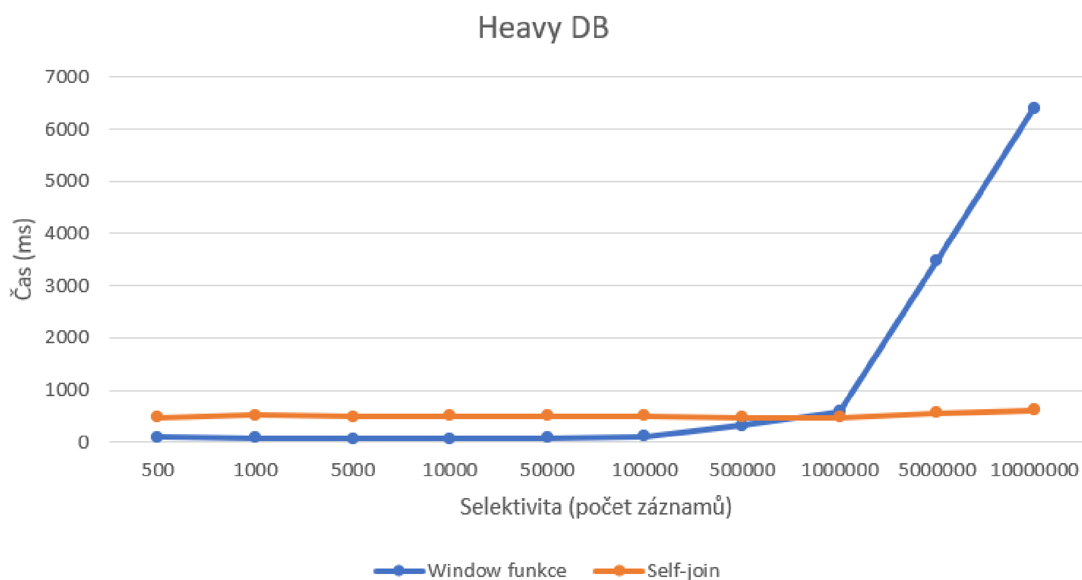
Grafické porovnání window funkce oproti *self-joinu* jsou vidět na obrázcích 2.12, 2.13 a 2.14.




Obrázek 2.12: SQL Server window funkce vs self-join




Obrázek 2.13: Postgre SQL window funkce vs self-join




Obrázek 2.14: Heavy DB window funkce vs self-join

Jak můžeme vidět, tak pro všechny tři databázové systémy jsme dostali podobné výsledky. Pro malou selektivitu bylo výhodnější použít window funkci, ale když se selektivita dostala nad jeden milion řádků, tak náročnost window funkce začala rapidně narůstat do bodu, kdy přerostla i self-

join. Self-join zde měl skoro konstantní čas u všech testů. Jedině u SQL Serveru začal růst čas i u self-joinu, ale stále méně než u window funkce.

Je to převážně z důvodu, že window funkce má zde větší náročnost než self-join, jelikož window funkce musí provést *sort* nad velkou tabulkou a udržovat skupiny pro všechny řádky, byť potřebujeme jen řádky s největším množství bodů za třídu (rank=1), tím pádem většinu práce window funkce zahodíme touto podmínkou. Toto je velmi náročné pro paměť při vyšším počtu řádků. Self-join zde již počítá konkrétní maximum.

Tento problém se netýká jen window funkce *DENSE RANK*, ale defacto všech window funkcí, kde po výpočtu window funkce vyfiltrujeme podmínkou většinu práce window funkce.

Většina známých databázových systémů nedokáže sama přepsat window funkci na self-join, i když by to bylo cenově výhodnější, z tohoto důvodu jsem vytvořil přepisovací pravidlo do Apache Calcite, které to dokáže.

# Kapitola 3

# Implementace

Jak jsem již popisoval na konci minulé kapitoly, tak cílem této práce bylo vytvořit přepisovací pravidlo, které bude přepisovat window funkci na klauzuli *GROUP BY*. Pro mé pravidlo jsem si zvolil přepis window funkce *dense rank* na *self-join*, který jsem popisoval v minulé kapitole. Přepisovací pravidlo je implementováno v prostředí Apache Calcite a není vázáno na určitý typ uložiště. V této kapitole popíšu podrobně prostředí Apache Calcite a fungování pravidla.

## 3.1 Prostředí Apache Calcite

Přepisovací pravidlo bylo vyvíjeno na verzi Apache Calcite 1.27.0. Apache Calcite je napsaný v jazyce Java. Pro vývoj byla použita Java SDK verze 17 a jako adaptér byl použit Apache Lucene, kde typ adaptéru nehraje velkou roli, jelikož přepisovací pravidlo přepisuje jen logickou část plánu a vůbec neinteraguje s uložištěm. Z tohoto důvodu lze pravidlo implementovat do jakéhokoliv adaptéru Apache Calcite. Pro zprovoznění adaptéru Apache Lucene mi pomohl workshop BOSS'21. Využil jsem jejich implementaci pro vytvoření Lucene indexu a následné zpracování dotazu [10].

Apache Calcite používá mnoho moderních technik vývoje softwaru. Mezi ně patří objektově orientované programování, použití generických typů, lambda výrazů, stream API, reflection API a mnoho dalších.

Pro tvorbu a testování pravidla jsem naimplementoval procesor pro spuštění dotazu, který obsahuje parsování a validování dotazu, převedení dotazu do logického plánu, provedení přepisovacích pravidel, vytvoření fyzického plánu a spuštění dotazu, který popisuji v kapitole 3.3. Ale abychom mohli pracovat s databází, tak jsem nadefinoval schéma databáze a vytvořit objekty v ní.

## 3.2 Tvorba schématu a dat

Pro tvorbu schématu databáze jsem vytvořil třídu typu enum **TestDatabase**. V této třídě jsou definované dvě tabulky *zaci* a *zaci_small*. Obě mají stejné schéma, které můžete vidět na obrázku

3.6 v kapitole 3.5.4.2. Dále v této třídě jsem nadefinoval tabulky z databáze TPC-H, kterou popisuji v kapitole 3.5.4.3. Zde lze přidat libovolný počet tabulek dle potřeby. Tato třída slouží jako definice schématu databáze.

Pro každou tabulku ze schématu musí být vytvořen Lucene index s daty. Toto testovací prostředí nemá implementovanou funkcionalitu pro klauzuli *insert* v dotazu, proto se zde pro vytvoření Lucene indexu využívá třída **DatasetIndexer**. Třída načítá csv soubory s daty a převádí je do Lucene indexu. Csv soubor musí mít jako oddělovač svislou čáru: |. Podporované kódování je UTF-8 NO BOM. Csv soubor musí být uložen v adresáři: solution/src/main/resources/data/test-database.

Po vytvoření schématu ve třídě **TestDatabase** a vytvoření csv souboru můžeme spustit třídu **DatasetIndexer**, která vytvoří Lucene index s daty.

## 3.3 Dotazový procesor

Vytvořil jsem třídu ***LuceneQueryProcessor***, která má za úkol zpracování a provedení dotazu. Zde můžeme vidět, co všechno databázový systém musí provést pro provedení dotazu. Tato třída provádí tyto operace:

- Načte SQL dotaz, který je na vstupu.
- Načte schéma databáze, konkrétně schéma *TestDatabase*.
- Rozparsuje SQL dotaz na části.
- Ověří validnost dotazu.
- Převede dotaz do logického plánu.
- Načte přepisovací pravidla, která se mají použít pro dotaz.
- Převede logický plán pomocí přepisovacích pravidel do fyzického plánu.
- Provede fyzický plán s nejnižší cenou.

Tato třída požaduje dva argumenty pro spuštění. První argument může mít hodnotu *USERULE*, nebo *NORULE* a definuje, jestli má být použito mé pravidlo, nebo ne. Druhý argument je cesta k SQL dotazu, který se má provést, pro příklad vstupní argumenty mohou vypadat následovně: *NORULE queries/tpch/zaci_small.sql*.

Po spuštění třídy *LuceneQueryProcessor* s příslušnými argumenty, se nám ukáže SQL dotaz, logický plán, fyzický plán, výsledek a čas vykonání dotazu, což můžeme vidět na obrázku 3.1.

```
[Parsed query]
SELECT SUM(`BODY`)
FROM (SELECT *, DENSE_RANK() OVER (PARTITION BY `TRIDA` ORDER BY `BODY` DESC) AS `RANK_VALUE`
FROM `ZACI_SMALL`) AS `RANKING`
WHERE `RANK_VALUE` = 1
[Logical plan]
LogicalAggregate(group=[{}], EXPR$0=[SUM($0)]), id = 5
  LogicalProject(body=[$3]), id = 4
    LogicalFilter(condition=[=($4, 1)]), id = 3
      LogicalProject(id=[$0], jmeno=[$1], trida=[$2], body=[$3], RANK_VALUE=[DENSE_RANK() OVER (PARTITION BY $2 ORDER BY $3 DESC)]), id = 2
        LogicalTableScan(table=[[ZACI_SMALL]]), id = 1

[Physical plan]
EnumerableAggregate(group=[{}], EXPR$0=[SUM($0)]): rowcount = 1.5, cumulative cost = {231.70625007152557 rows, 1691.0 cpu, 0.0 io}, id = 35
  EnumerableCalc(expr#0..4=[{inputs}], body=[$t3]): rowcount = 15.0, cumulative cost = {230.0 rows, 1691.0 cpu, 0.0 io}, id = 34
    EnumerableCalc(expr#0..4=[{inputs}], expr#5=[1:BIGINT], expr#6=[=($t4, $t5)], proj#0..4=[{exprs}], $condition=[$t6]): rowcount = 15.0, cumulative cost = {215.0 rows, 1601.0 cpu, 0.0 io}, id = 33
      EnumerableWindow(window#0=[window(partition {2} order by [3 DESC] aggs [DENSE_RANK()])]): rowcount = 100.0, cumulative cost = {200.0 rows, 301.0 cpu, 0.0 io}, id = 32
        EnumerableTableScan(table=[[ZACI_SMALL]]): rowcount = 100.0, cumulative cost = {100.0 rows, 101.0 cpu, 0.0 io}, id = 17

Elapsed time 258ms
87830425

Process finished with exit code 0
```

Obrázek 3.1: Výstup programu

## 3.4 Přepisovací pravidlo

Po nachystání testovacího prostředí běžící na Apache Lucene jsem začal s vývojem přepisovacího pravidla. Pro Apache Calcite neexistuje dokumentace, která by popisovala, jak vytvořit nové přepisovací pravidlo, proto jsem čerpal z již existujících přepisovacích pravidel, které jsou implementované v Apache Calcite. V této kapitole popisuji, jak se přepisovací pravidlo vytváří.

Tvorbu přepisovacích pravidel rozděluji do dvou bodů, definice přepisovacího pravidla, kdy se má použít a implementace pravidla.

### 3.4.1 Definice přepisovací pravidlo

Pro definici přepisovacího pravidla byla vytvořena abstraktní třída **WindowFunctionRule**. Tato třída má za úkol definovat, za jakých podmínek se má pravidlo provést. To, že se pravidlo provede ale nutně neznamená, že se použije ve finálním plánu, vybere se totiž plán s nejnižší cenou.

Prvně se definuje, kdy se má pravidlo uplatnit, to znamená, pro jaký logický plán se uplatní. Mám dva dotazy, kde dotaz 3.1 chci převést na dotaz 3.2.

```
1  SELECT sum(body)
2  FROM (
3  SELECT
4  *,
5  DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
6  FROM zaci_small
7  WHERE body <= 10000000
8  ) ranking
9  WHERE rank_value = 1;
```

Kód 3.1: Dotaz A

```sql
1  SELECT sum(body)
2  FROM zaci c
3  JOIN (
4  SELECT trida, MAX(body) rank_value
5  FROM zaci_small
6  WHERE body <= 10000000
7  GROUP BY trida
8  ) ranking ON c.trida = ranking.trida AND
9  c.body = ranking.rank_value;
```

Kód 3.2: Dotaz B

Když si oba dotazy pustíme v Apache Calcite, tak uvidíme jejich logické plány na obrázku 3.2 a 3.3.

```
[Logical plan]
LogicalAggregate(group=[{}], EXPR$0=[SUM($0)]), id = 6
  LogicalProject(body=[$3]), id = 5
    LogicalFilter(condition=[=($4, 1)]), id = 4
      LogicalProject(id=[$0], jmeno=[$1], trida=[$2], body=[$3], RANK_VALUE=[DENSE_RANK() OVER (PARTITION BY $2 ORDER BY $3 DESC)]), id = 3
        LogicalFilter(condition=[<=($3, 10000000)]), id = 2
          LogicalTableScan(table=[[ZACI_SMALL]]), id = 1
```

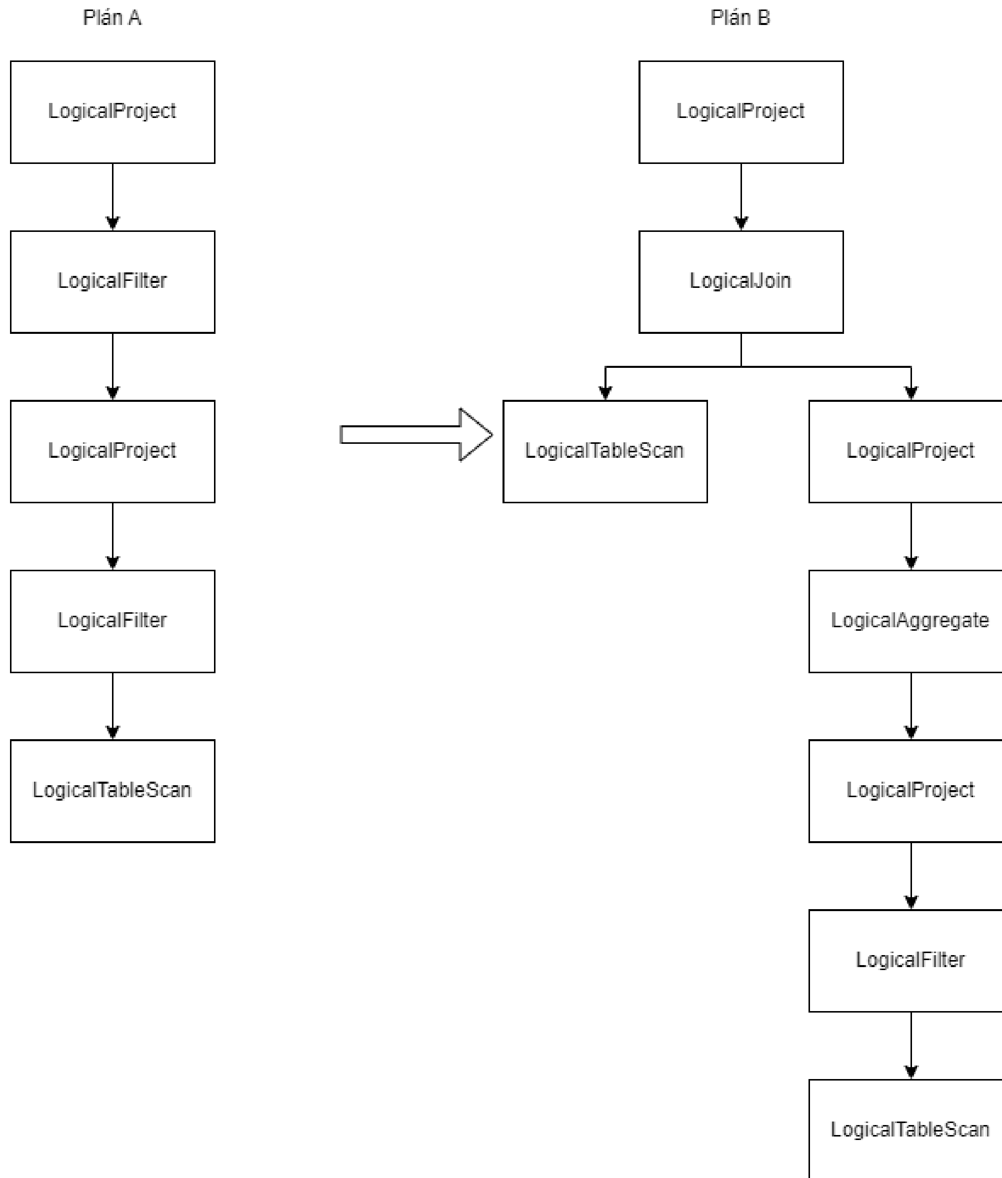
Obrázek 3.2: Plán A

```
[Logical plan]
LogicalAggregate(group=[{}], EXPR$0=[SUM($0)]), id = 11
  LogicalProject(body=[$3]), id = 10
    LogicalJoin(condition=[AND(=($2, $4), =($3, $5))], joinType=[inner]), id = 9
      LogicalTableScan(table=[[ZACI_SMALL]]), id = 1
      LogicalProject(TRIDA=[$0], RANK_VALUE=[$1]), id = 7
        LogicalAggregate(group=[{0}], RANK_VALUE=[MAX($1)]), id = 6
          LogicalProject(trida=[$2], body=[$3]), id = 5
            LogicalFilter(condition=[<=($3, 200)]), id = 4
              LogicalTableScan(table=[[ZACI_SMALL]]), id = 3
```

Obrázek 3.3: Plán B

Tyto logické plány se čtou podle odsazení. Řádek, který je nejvíce odsazený doprava (zde LogicalTableScan) se provede jako první, následně druhý nejvíc odsazený řádek. Každý řádek v logickém plánu reprezentuje logický operátor. Mé přepisovací pravidlo má za úkol udělat z plánu A plán B.

Pravidlo nahradí část plánu A za část plánu B. Pro přehlednost byl vytvořen diagram reprezentující přepisovací pravidlo, který je znázorněn na obrázku 3.4. Na diagramu můžeme vidět, že není poslední (vrchní) logický operátor LogicalAggregate. To je z důvodu, že v obou plánech je tento operátor identický a z toho důvodu ho nemusíme přepisovat. Všechny ostatní operátory z plánu A jsem musel nějakým způsobem modifikovat anebo transformovat na nový logický operátor. Toto popisuji v následující kapitole 3.4.2.



Obrázek 3.4: Diagram přepisovacího pravidla

Jak můžeme vidět, tak mé pravidlo přepisuje 5 logických operátorů na 8 odlišných logických operátorů. Proto se pravidlo bude brát v potaz, pokud dotaz bude obsahovat 5 operátorů nalevo, což můžeme pojmenovat jako podmínky uplatnění přepisovacího pravidla. Logické operátory z diagramu pro plán A byly přepsány do třídy **WindowFunctionRule**, jak můžeme vidět v kódu 3.3 na řádcích 3 až 7, které kopírují operátory z obrázku 3.4. Pravidlo se uplatní, pokud splní následující podmínky.

```java
default Config withOperandFor(Class<? extends Filter> filter) {
   return withOperandSupplier(
      b0 -> b0.operand(Project.class)
         .oneInput(b1 -> b1.operand(filter)
            .oneInput(b2 -> b2.operand(Project.class)
               .oneInput(b3 -> b3.operand(filter)
                  .oneInput(b4 -> b4.operand(TableScan.class)
                     .anyInputs()))))).as(Config.class);
}
```

Kód 3.3: Definice přepisovacího pravidla

### 3.4.2 Implementace přepisovacího pravidla

Implementace přepisovacího pravidla se provádí ve třídě **WindowFunctionToSelfJoinRule**. Třída dědí z abstraktní třídy *WindowFunctionRule* a implementuje třídu **SubstitutionRule**, která je klíčová pro přepisovací pravidlo. Třída *SubstitutionRule* nám definuje interface přepisovacího pravidla, kde pro definici přepisovacího pravidla je důležitá funkce **onMatch**. Funkce se spustí, když jsou splněny podmínky ze třídy *SubstitutionRule*. Na vstupu má funkce objekt typu *RelOptRuleCall*, což je třída, kde je uložena část plánu pro přepis. Zde konkrétně to je část plánu z obrázku 3.4 pro plán A, kde na začátku funkce získávám jednotlivé logické operátory do proměnných, jak můžeme vidět v kódu 3.4 na řádcích 4 až 8.

```java
   @Override
   public void onMatch(RelOptRuleCall call) {
      final RelBuilder relBuilder = call.builder();
      final Project project2 = call.rel(0);
      final Filter filter = call.rel(1);
      final Project project = call.rel(2);
      final Filter filter2 = call.rel(3);
      final TableScan tableScan = call.rel(4);
```

Kód 3.4: onMatch funkce

Cílem této funkce je transformovat vstupní objekt *RelOptRuleCall*, aby obsahovala místo původní části plánu jinou transformovanou ekvivalentní část plánu.

Má implementace funkce *onMatch*, která reprezentuje přepis plánu, se dělí do několika částí:

#### 3.4.2.1 Detekce použití

První věc, která se musí udělat, je detekce, jestli se přepis může provést. Víme, že na vstupu máme správné operátory pro přepis, ale mé přepisovací pravidlo se dá použít jen pro window funkci *DENSE_RANK* nebo *RANK*, takže musíme první detekovat, jestli je použita správná window

funkce. V kódu 3.4 získáváme logické operátory z původního plánu do proměnných. V proměnné *project* jsou uloženy informace o použité window funkci. Z této proměnné jsme schopni vytvořit podmínku, která ověřuje, zda byla použita window funkce *DENSE_RANK* nebo *RANK*. Pokud nebyla použita, tak se funkce ukončí beze změny plánu.

Po detekci správné window funkce ověřuji z původního plánu filtrační podmínku, kde přepisovací pravidlo funguje jen na podmínku *rank=1*, která má být uložena v proměnné *filter*. Pokud by v této podmínce bylo jiné číslo nebo jiný sloupec, tak se funkce ukončí beze změny plánu.

#### 3.4.2.2 Vložení operátoru *LogicalFilter*

Po ověření, že se pravidlo může provést, vytvářím nové operátory, kde je prvním operátorem *LogicalFilter*. Tvorba tohoto operátoru je jednoduchá, jelikož se operátor nemusí nijak transformovat a převezme se z původního plánu. Operátor vložím do proměnné *relBuilder*, která reprezentuje transformovaný plán. Po spuštění kódu 3.5 uvidíme pomocí debugu, že proměnná *relBuilder* má definovanou část plánu 3.6. Můžeme vidět, že po vložení operátoru *LogicalFilter* se nám dostal do plánu i jeho předek *LogicalTableScan*, to je z toho důvodu, že funkce *push* vloží nejenom dotyčný operátor, ale i všechny jeho předky z kterých čerpá, tím jsme dosáhli načtení tabulky a vyfiltrování záznamů podle podmínky.

```
1   relBuilder.push(filter2);
```

Kód 3.5: Vložení operátoru LogicalFilter

```
1   LogicalFilter(condition=[<=($3, 200)])
2     LogicalTableScan(subset=[rel#7:RelSubset#0.NONE], table=[[ZACI_SMALL]])
```

Kód 3.6: Plán po vložení operátoru LogicalFilter

#### 3.4.2.3 Tvorba prvního operátoru *LogicalProject*

Další operátor, který je na řadě, jak můžeme vidět z výše zmíněného diagramu 3.4, je operátor **LogicalProject**. Tento operátor má na starost zahodit sloupce, které nebudou potřeba pro následující agregaci.

Jak ale zjistíme, jaké jsou potřeba a jaké ne? Zjistíme to z původní window funkce, která je uložena v proměnné *project*. Bude se jednat o sloupce, které jsou použité v klauzuli *partition by* a *order by*. Všechny ostatní sloupce v této projekci jsou pro nás nepodstatné.

U vytvoření nové projekce jsem narazil na problém. Z proměnné *project*, kde je uložena window funkce dokážu vyčíst sloupce v *partition by* a *order by* jako id sloupce vygenerované z operátoru *LogicalTableScan*. Ale vytvořit novou projekci umím jen pomocí názvů sloupců. S tím mi pomohl tento kód 3.7, díky kterému jsme schopni si zobrazit aktuální sloupce v jakékoliv části přepisovacího pravidla. Díky tomuto bylo možné namapovat id sloupce k názvu toho sloupce. Tento kód jsem hojně

využíval pro různou kontrolu správnosti přepisu, jelikož při zobrazení plánu nejsou vždy vidět názvy sloupců, ale většinou jen jejich id.

```
1   distinctFields = relBuilder.peek().getRowType().getFieldList();
```

Kód 3.7: Zobrazení aktuální sloupce přepisovacího pravidla

Po získání názvů sloupců použitých v *partition by* a *order by*, byly tyto názvy uloženy do proměnné *fieldNames*. Následně byly použity pro vložení nového operátoru pomocí kódu 3.8, který doplnil plán o projekci, jak můžeme vidět v plánu 3.9.

```
1   relBuilder.project(relBuilder.fields(fieldNames));
```

Kód 3.8: Vložení operátoru LogicalProject

```
1   LogicalProject(trida=[$2], body=[$3])
2     LogicalFilter(condition=[<=($3, 200)])
3       LogicalTableScan(subset=[rel#7:RelSubset#0.NONE], table=[[ZACI_SMALL]])
```

Kód 3.9: Plán po vložení operátoru LogicalProject

#### 3.4.2.4 Tvorba operátoru *LogicalAggregate*

Po vytvoření selekce sloupců je na řadě agregace. Pro vytvoření ekvivalentního plánu je potřeba agregovat podle sloupce nebo sloupců, které byly v *partition by* klauzuli ve window funkci, a na sloupce v *order by* klauzuli musí být provedena agregační funkce MIN nebo MAX. Rozhodnutí jestli se použije MIN nebo MAX závisí na tom, zdali se řadilo sestupně, nebo vzestupně. Pokud sestupně, tak je použita agregační funkce MAX, u sestupně MIN.

Přepisovací pravidlo umí pracovat jen s jedním sloupcem v *order by* klauzuli. Pokud tam je víc sloupců, tak se přepis neprovede. V klauzuli *partition by* může být neomezené množství sloupců.

Před vytvořením operátoru pro agregaci jsem si nachystal do proměnné *fieldNamesPartition* názvy sloupců z klauzule *partition by*. Do proměnné *fieldNamesOrder* byly uloženy sloupce z klauzule *order by* ve window funkci. Také bylo definována boolean proměnná *order_desc*, která obsahuje informaci, jestli je to řazené vzestupně.

S těmito proměnnými byl vložen do plánu operátor *LogicalAgregate*, jak můžeme vidět v kódu 3.10 a v plánu 3.11, kde se rozhoduje na řádku 1, jestli bude použit MIN nebo MAX a následně se vytváří agregace. Funkce *groupKey* definuje podle čeho se agreguje a funkce *aggregateCall* definuje funkci a sloupce, které agregujeme.

```
if(order_desc) {
   relBuilder.aggregate(
      relBuilder.groupKey(fieldNamesPartition.toArray(new String[0])),
      relBuilder.aggregateCall(SqlStdOperatorTable.MAX,
         relBuilder.fields(fieldNamesOrder))
   );
} else {
   relBuilder.aggregate(
      relBuilder.groupKey(fieldNamesPartition.toArray(new String[0])),
      relBuilder.aggregateCall(SqlStdOperatorTable.MIN,
         relBuilder.fields(fieldNamesOrder))
   );
}
```

Kód 3.10: Vložení operátoru LogicalAgregate

```
LogicalAggregate(group=[{0}], agg#0=[MAX($1)])
   LogicalProject(trida=[$2], body=[$3])
      LogicalFilter(condition=[<=($3, 200)])
         LogicalTableScan(subset=[rel#7:RelSubset#0.NONE], table=[[ZACI_SMALL]])
```

Kód 3.11: Plán po vložení operátoru LogicalAggregate

#### 3.4.2.5 Tvorba operátoru *LogicalTableScan*

Následující operátor, který byl vytvořil je *LogicalTableScan*, který je nutný pro následující operátor *LogicalJoin*. Přepisovací pravidlo má za úkol načíst tabulku pro zpracování self-joinu. Kvůli této činnosti jsem měl ve třídě *WindowFunctionRule*, která definuje použití přepisovacího pravidla, vstupní operátor *LogicalTableScan*. Díky tomu mám jeho hodnoty k dispozici v přepisovacím pravidle v proměnné *tableScan* a jsem schopný ho vytvořit pomocí kódu 3.12, jak můžeme vidět v plánu 3.13.

```
relBuilder.push(tableScan);
```

Kód 3.12: Vložení operátoru LogicalTableScan

```
LogicalTableScan(table=[[ZACI_SMALL]])
LogicalAggregate(group=[{0}], agg#0=[MAX($1)])
   LogicalProject(trida=[$2], body=[$3])
         LogicalFilter(condition=[<=($3, 200)])
            LogicalTableScan(subset=[rel#7:RelSubset#0.NONE], table=[[ZACI_SMALL]])
```

Kód 3.13: Plán po vložení operátoru LogicalAggregate

#### 3.4.2.6 Tvorba operátoru *LogicalJoin*

Jak již bylo v minulé kapitole zmíněno, následující operátor je *LogicalJoin*, konkrétně self-join, který bude spojovat minulý operátor *LogicalTableScan* a *LogicalAggregate*. Zde se opět čerpá z původní window funkce, jelikož spojovací podmínky musí být na všechny sloupce, které byly použity v klauzuli *partition by* a *order by* ve window funkci. Sloupce z klauzule *partition by* je spojován pomocí reálného názvu v obou operátorech, kde se sloupce nepřejmenovávají. Pro získání konkrétního sloupce byla použita funkce **field**, jak můžeme vidět v kódu 3.14 na řádcích 4, 5, 11 a 12. Funkce má tři vstupy: počet vstupů, číslo vstupu (0 je pro vstup z *LogicalAggregate* operátoru a 1 je z *LogicalTableScan* operátoru) a název sloupce. Podmínka následně byla vytvořena pomocí funkce **equals** na řádcích 3 a 10, která označuje, že oba sloupce se musí rovnat. Spojovací podmínky jsou uložené v proměnné **conditions**.

U sloupců z klauzule *order by* to bylo složitější, jelikož v operátoru *LogicalAggregate* je na sloupce použita agregační funkci MIN nebo MAX. To zapříčinilo to, že sloupce mají jiný název, proto jsem zjistil jejich nový název, který jsem získal pomocí již použitého kódu 3.7, kde dokážu získat všechny aktuální sloupce v plánu. Nová jména těchto sloupců jsou uložena do proměnné *fieldNamesOrderAgg*.

```
1   List<RexNode> conditions = new ArrayList<>();
2   for (String colname: fieldNamesPartition) {
3      conditions.add(relBuilder.equals(
4         relBuilder.field(2, 0, colname),
5         relBuilder.field(2, 1, colname)
6      ));
7   }
8
9   for(int i = 0; i < fieldNamesOrder.size(); i++) {
10     conditions.add(relBuilder.equals(
11        relBuilder.field(2, 0, fieldNamesOrderAgg.get(i)),
12        relBuilder.field(2, 1, fieldNamesOrder.get(i))
13     ));
14  }
```

Kód 3.14: Vytvoření spojovacího podmínek pro LogicalJoin

Po vytvoření všech potřebných spojovacích podmínek v proměnné *conditions*, byl vytvořen nový operátor *LogicalJoin*, který je znázorněn v kódu 3.15 a v plánu 3.16.

```
1   relBuilder.join(JoinRelType.INNER,
2      relBuilder.and(conditions)
3   );
```

Kód 3.15: Vložení operátoru LogicalTableScan

```
1    LogicalJoin(condition=[AND(=($0, $4), =($1, $5))], joinType=[inner])
2      LogicalAggregate(group=[{0}], agg#0=[MAX($1)])
3        LogicalProject(trida=[$2], body=[$3])
4          LogicalFilter(condition=[<=($3, 200)])
5            LogicalTableScan(subset=[rel#7:RelSubset#0.NONE], table=[[ZACI_SMALL]])
6      LogicalTableScan(table=[[ZACI_SMALL]])
```

Kód 3.16: Plán po vložení operátoru LogicalJoin

#### 3.4.2.7 Tvorba druhého operátoru *LogicalProject*

Poslední operátor, který byl vytvořen je *LogicalProject*. Na první pohled může vypadat, že operátor se nemusí měnit a může se použít ten z původního plánu, ale není tomu tak. Je to z důvodu, že v původním plánu měl jiný vstup, než má teď. Vstupem v původním plánu jsou všechny sloupce z tabulky a hodnota *dense_rank* nebo *rank*. Zatímco teď jsou vstupem všechny sloupce z tabulky a agregace, proto kdybychom použili stejný operátor z původního plánu, tak při určitém vstupu bychom dostali chybu. Proto byl namapován původní *LogicalProject* operátor na aktuální vstup. Původní operátor je uložený v proměnné *project2* a z něj byli získány názvy sloupců do proměnné *finalProjectColumn*. Následně byl vytvořen druhý operátor *LogicalProject*, jak můžeme vidět v kódu 3.17 a v plánu 3.18.

```
1    relBuilder.project(relBuilder.fields(finalProjectColumn));
```

Kód 3.17: Vložení druhého operátoru LogicalProject

```
1  LogicalProject(body=[$5])
2    LogicalJoin(condition=[AND(=($0, $4), =($1, $5))], joinType=[inner])
3      LogicalAggregate(group=[{0}], agg#0=[MAX($1)])
4        LogicalProject(trida=[$2], body=[$3])
5          LogicalFilter(condition=[<=($3, 200)])
6            LogicalTableScan(subset=[rel#7:RelSubset#0.NONE], table=[[ZACI_SMALL]])
7      LogicalTableScan(table=[[ZACI_SMALL]])
```

Kód 3.18: Plán po vložení druhého operátoru LogicalProject

#### 3.4.2.8 Transformace plánu

Po vytvoření všech potřebných operátorů, které jsou v plánu 3.18, je nutné původní plán transformovat na nově vytvořený plán. Původní plán je uložený v proměnné *call* a nově vytvořený plán je v proměnné *relBuilder*. Transformace se provede pomocí kódu 3.19.

```
1    call.transformTo(relBuilder.build());
```

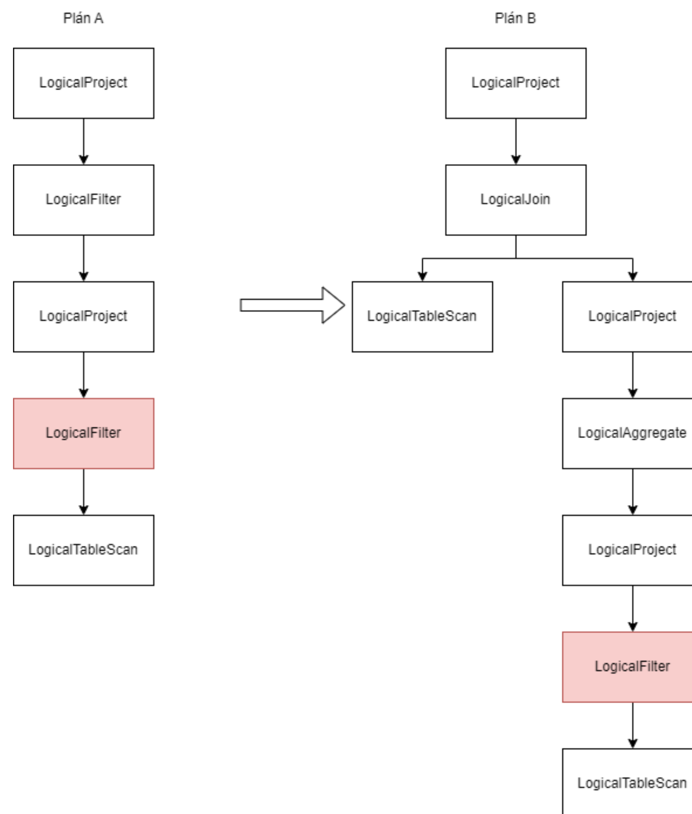
Kód 3.19: Transformace původního plánu na nový

Tímto vzniklo přepisovací pravidlo pro transformaci window funkce na self-join. Problém je, že se přepisovací pravidlo uplatní pouze pokud odhad ceny tohoto plánu bude menší než všechny ostatní plány. Jelikož pro testování používám adaptér Apache Lucene, tak u specifického adaptéru má přepis vždy větší odhadovanou cenu než použití window funkce, proto by se nikdy nově vytvořené pravidlo nepoužilo v tomto adaptéru. Naštěstí lze vynutit použití pravidla pro finální plán, a to přidáním kódu 3.20 do přepisovacího pravidla.

```
1   call.getPlanner().prune(project);
```

Kód 3.20: Vynucení přepisovacího pravidla

Toto přepisovací pravidlo dokáže přepsat jen dotaz, který v poddotazu má filtrační podmínku. Pokud tomu tak není, tak nebude splňovat požadavky ze třídy *WindowFunctionRule* a pravidlo nebude uplatněno, proto jsem vytvořil druhé přepisovací pravidlo **WindowFunctionToSelfJoin-SimpleRule**, které je obdoba výše popisovaného pravidla s rozdílem, že pravidlo je definované na dotaz bez filtrační podmínky, tím pádem jak na vstupním plánu, tak na výstupním plánu bude o operátor *LogicalFilter* méně, jak můžeme vidět v diagramu 3.5, kde červeně označený operátor v plánech nebude.



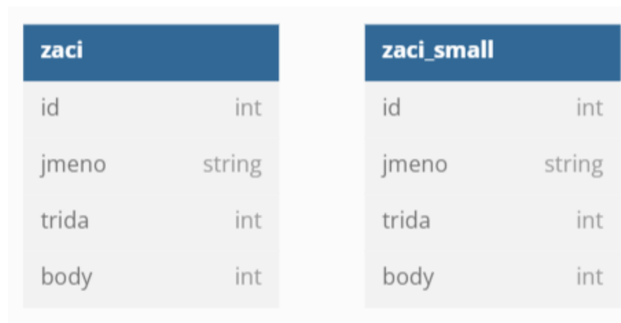

Obrázek 3.5: Diagram druhého přepisovacího pravidla

## 3.5 Testování pravidla

Po vytvoření přepisovacího pravidla byla vytvořena řada testů pro změření správnosti a výkonnosti. Testování pravidla bylo provedeno na počítači s operačním systémem Windows 11, procesorem intel I7-9750H a RAM pamětí 32GB. Testování bylo provedeno jak nad umělými daty, tak nad reálnými daty.

### 3.5.1 Umělá data

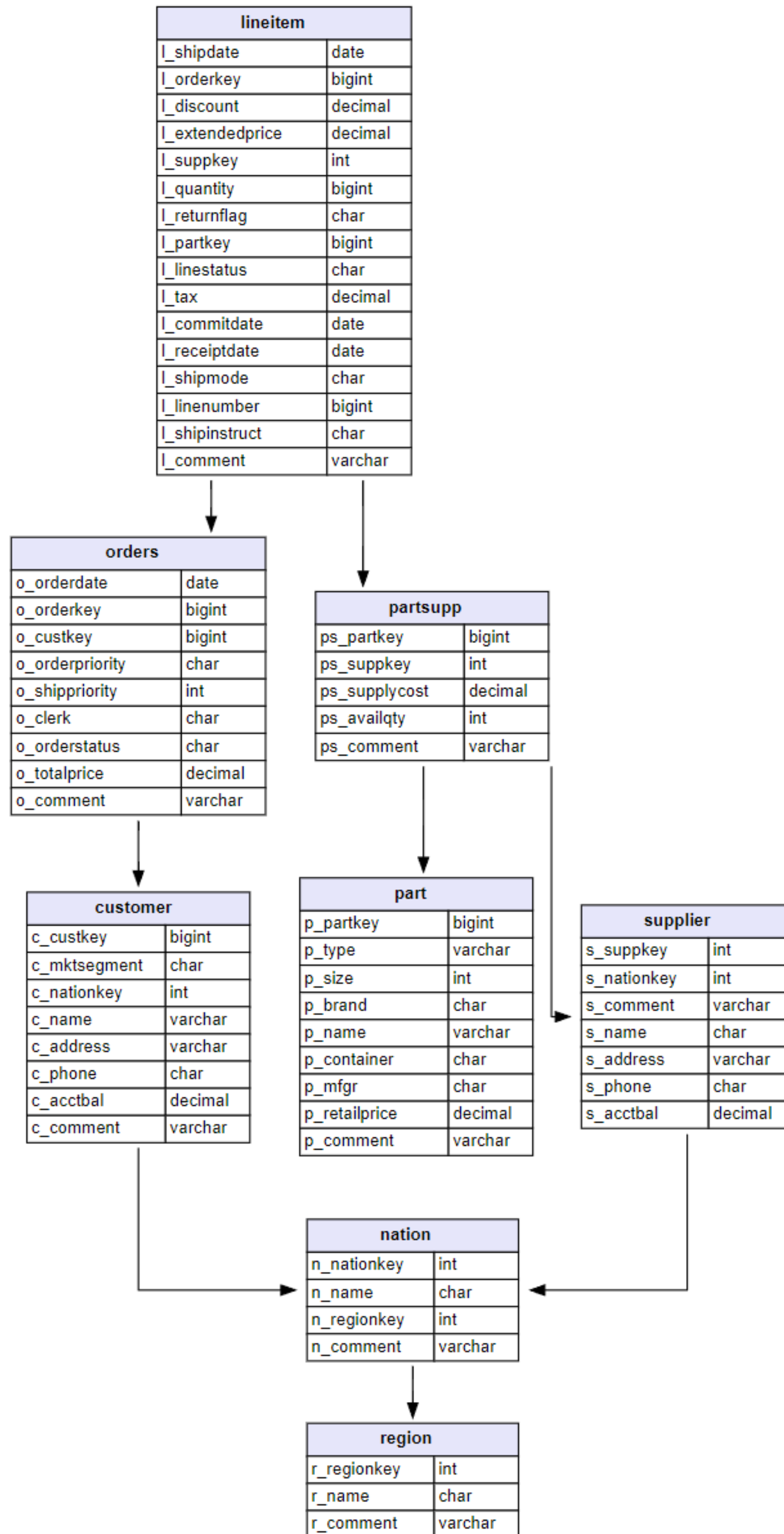
Reprezentací umělých dat je tabulka **zaci** a její zmenšená verze **zaci_small**. S tabulkou *zaci* jsem již pracoval, a to v předchozí kapitole 2.8.4, kde jsem měřil výkon window funkce oproti self-joinu nad třemi různými databázovými systémy. Data v těchto tabulkách jsou náhodně vygenerována a nachází se v příloze E. Schéma těchto tabulek je vyobrazeno níže na obrázku 3.6. Tabulka *zaci* obsahuje 10 000 000 řádků a tabulka *zaci_small* obsahuje 200 řádků.




Obrázek 3.6: Schéma tabulek zaci a zaci_small

### 3.5.2 Reálná data

Pro reprezentaci reálných dat byla použita databáze **TPC-H**, která je využívaná pro měření výkonu různých relačních databází. Byla vytvořena neziskovou organizaci TPC. Obsahuje necelých 7 miliónů záznamů. Data byla vybrána tak, aby pokryla široké odvětví průmyslu [11]. Schéma této databáze můžete vidět na následující straně, na obrázku 3.7.




Obrázek 3.7: Schéma TPCH databáze [12]

### 3.5.3 Testování výkonnosti pravidla

#### 3.5.3.1 Testování v Apache Lucene

Jako první byla testována výkonnost přepisu v optimalizaci, neboli, jak dlouho trvá přepis pravidla z window funkce na self-join. Testování bylo provedeno na již dříve zmíněné tabulce *zaci_small*, kde byl porovnáván čas dotazu s window funkcí (kód 3.1), který používá mé pravidlo, oproti dotazu, kde je již self-join (kód 3.2). Oba tyto dotazy mají identický zpracovávaný plán, kde jen první dotaz provádí transformaci plánu z window funkce. Provedl jsem 10 měření pro oba dotazy. Čas, o který je dotaz, s použitím mého přepisovacího pravidla, pomalejší, považuji jako dobu přepisu. Výsledek testování můžeme vidět na obrázku 3.8.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | průměr |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | čas (ms) | | | | | | | | | | |
| **S pravidlem** | 406 | 428 | 394 | 405 | 410 | 400 | 394 | 412 | 384 | 409 | 404,2 |
| **Bez pravidla** | 306 | 434 | 371 | 406 | 409 | 368 | 377 | 383 | 413 | 399 | 386,6 |

Obrázek 3.8: Doba trvání dotazu self-joinu s přepisem a bez

Jak můžeme vidět, tak rozdílný čas s přepisem a bez přepisu je v řádu jednotek milisekund, maximálně v řádu desítek milisekund, tím pádem můžeme považovat dobu trvání přepisu z window funkce na self-join jako čas, který je zanedbatelný, a tím pádem přepisovací pravidlo výrazně nezpomaluje optimalizaci dotazu.

Dále jsem testoval výkonnost použití mého pravidla. Testování bylo prováděno nad dotazem v kódu 3.21, kde se porovnává čas s použitím mého pravidla oproti času bez použití pravidla.

```
1  SELECT sum(body)
2  FROM (
3     SELECT
4        *,
5        DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
6     FROM zaci
7  ) ranking
8  WHERE rank_value = 1;
```

Kód 3.21: Testovací dotaz pro testování výkonnosti

Z testování jsem zjistil, že adaptér Apache Lucene není moc výkonný v dotazování nad velkým množstvím dat, jelikož vykonání jakéhokoliv dotazu trvá v řádu jednotek až desítek minut. Proto jsem provedl 5 měření. Výsledek měření je vyobrazen na obrázku 3.9.

| | 1 | 2 | 3 | 4 | 5 | průměr |
|---|---|---|---|---|---|---|
| | čas (ms) | | | | | |
| **S pravidlem** | 230829 | 227216 | 220803 | 214106 | 223377 | 223266,2 |
| **Bez pravidla** | 135212 | 128919 | 130281 | 129227 | 126780 | 130083,8 |

Obrázek 3.9: Doba trvání dotazu s použitím pravidla a bez na velké databázi

Jak můžeme vidět, tak zde při použití adaptéru Apache Lucene je čas pro přepis skoro dvakrát horší, než použití window funkce, je to z toho důvodu, že adaptér Apache Lucene neumí efektivně číst velké množství dat pomocí operátoru *TableScan*. Tento operátor zabere 90 % času vykonání celého dotazu. Mé přepisovací pravidlo přepisuje dotaz na dotaz, který obsahuje dvě čtení z tabulky, tím pádem bude vždy výkonnější plán s jedním čtením tabulky (window funkce), než plán se dvěma čtením. I když bych měnil selektivitu, tak na čase vykonání skoro neuvidíme rozdíl, kvůli velkého čtení z tabulky.

Obdobné testování výkonnosti přepisovacího pravidla jsem provedl i nad reálnými daty z TPC-H databáze, kde výsledek vyšel obdobný výsledku v této kapitole.

Závěrem výkonnostního testování v Apache Lucene je, že v tomto adaptéru není výhodné použít mé přepisovací pravidlo kvůli pomalému operátoru *TableScan*.

#### 3.5.3.2 Testování výkonnosti v HeavyDB

Kvůli malé výkonnosti Apache Lucene jsem hledal jiné adaptéry a systémy využívající Apache Calcite, kde by přepis pomocí přepisovacího pravidla byl efektivní. Vyzkoušel jsem si většinu základních adaptérů, které nabízí Apache Calcite. Naneštěstí všechny, co jsem odzkoušel nebyly optimalizované pro dotaz nad velkým množstvím dat a měly stejný problém jako Apache Lucene. Proto jsem zkoumal databázové systémy, které používají optimalizátor z Apache Lucene. Taková databáze je například HeavyDB, která využívá parser a optimalizátor z Apache Calcite a která je zaměřená na práci s velkým množstvím dat, kde jsem provedl testování, které je v kapitole 2.8.4. Kde vyšlo, že pro dotaz s vysokou selektivitou nad velkým počtem dat je rychlejší použití self-joinu, než window funkce. Z tohoto důvodu můžeme předpokládat, že zde moje pravidlo bude efektivní.

Bohužel po implementaci pravidla v Apache Lucene, jsem zjistil, že na svém počítači nejsem schopný zprovoznit vývojovou verzi HeavyDB, kde bych mohl přidávat funkcionalitu. Tato verze totiž požadovala podporu architektury CUDA, kterou jsem neměl k dispozici, i když instalace a spuštění základní verze HeavyDB tento požadavek neměla. Proto jsem byl schopen si vyzkoušet HeavyDB databázi, ale již jsem nebyl schopný ji nijak modifikovat.

Jelikož jsem neměl možnost přidávat funkcionalitu do HeavyDB, tak jsem provedl analýzu, jestli mé přepisovací pravidlo lze implementovat do HeavyDB. HeavyDB využívá Apache Calcite pro parsování a využívá jeho optimalizátor, proto, aby moje přepisovací pravidlo fungovalo v HeavyDB je potřeba, aby parser, který převádí dotaz na logický plán, byl identický s parserem v Apache Lucene, kde jsem to testoval. HeavyDB musí používat stejné logické operátory, se kterými moje přepisovací pravidlo pracuje. Tyto operátory si můžeme znáznornit, když si zobrazíme logický plán pro dotaz v HeavyDB. Logický plán získáme v HeavyDB pomocí klauzule *EXPLAIN CALCITE*. Díky této klauzule uvidíme logický plán pro dotaz s window funkcí na obrázku 3.10 a logický plán pro self-join na obrázku 3.11.

```
Explanation
LogicalAggregate(group=[{}], EXPR$0=[SUM($0)])
 LogicalProject(body=[$3])
  LogicalJoin(condition=[AND(=($2, $5), =($3, $6))], joinType=[inner])
   LogicalTableScan(table=[[heavyai, zaci]])
   LogicalAggregate(group=[{0}], rank_value=[MAX($1)])
    LogicalProject(trida=[$2], body=[$3])
     LogicalFilter(condition=[<=($3, 10000000)])
      LogicalTableScan(table=[[heavyai, zaci]])
```

Obrázek 3.10: HeavyDB logický plán pro window funkci

```
Explanation
LogicalAggregate(group=[{}], EXPR$0=[SUM($0)])
 LogicalProject(body=[$3])
  LogicalFilter(condition=[=($4, 1)])
   LogicalProject(id=[$0], jmeno=[$1], trida=[$2], body=[$3], rank_value=[DENSE_RANK() OVER (PARTITION BY $2 ORDER BY $3 DESC)])
    LogicalFilter(condition=[<=($3, 10000000)])
     LogicalTableScan(table=[[heavyai, zaci]])
```

Obrázek 3.11: HeavyDB logický plán pro self-join

Jak můžeme vidět na obrázku 3.2 a 3.3, tak HeavyDB používá identické logické operátory se kterými pracuje mé přepisovací pravidlo.

Dále jsem provedl analýzu přepisovacích pravidel implementovaných v HeavyDB. Zde se inicializují pravidla ve třídě *HeavyDBPlanner* podobným způsobem, jako je inicializuju u Apache Lucene ve třídě *LuceneQueryProcessor*. Příklad přidávání přepisovacích pravidel v HeavyDB můžeme vidět v kódu 3.22 na řádcích 2, 4, 6, 7, 9, 11 a 12.

```
1   if (foundView) {
2      firstOptPhaseProgram.addRuleInstance(
3      CoreRules.JOIN_PROJECT_BOTH_TRANSPOSE_INCLUDE_OUTER);
4      firstOptPhaseProgram.addRuleInstance(CoreRules.FILTER_MERGE);
5   }
6   firstOptPhaseProgram.addRuleInstance(CoreRules.FILTER_PROJECT_TRANSPOSE);
7   firstOptPhaseProgram.addRuleInstance(
8   FilterTableFunctionMultiInputTransposeRule.Config.DEFAULT.toRule());
9   firstOptPhaseProgram.addRuleInstance(CoreRules.FILTER_PROJECT_TRANSPOSE);
10  if (foundView) {
11     firstOptPhaseProgram.addRuleInstance(CoreRules.PROJECT_MERGE);
12     firstOptPhaseProgram.addRuleInstance(ProjectProjectRemoveRule.INSTANCE);
13  }
```

Kód 3.22: HeavyDB inicializace přepisovacích pravidel

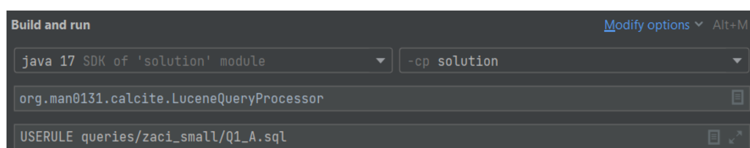
Tím pádem, pro zprovoznění mého pravidla v HeavyDB, by stačilo doplnit třídu *HeavyDBPlanner* o moje přepisovací pravidlo. Z předpokladu z měření výkonnosti window funkce a self-join v kapitole 2.8.4 a z měření doby trvání přepisu v Apache Lucene z kapitoly 3.5.3.1, můžeme vyvodit závěr, že použití mého pravidla bude efektivní v databázovém systému HeavyDB.

### 3.5.4 Testování správnosti pravidla

Vytvořil jsem sadu testovacích dotazů, pro ověření správnosti fungování přepisovacího pravidla v adaptéru Apache Lucene. Testováni bylo provedeno nad umělými daty i nad reprezentací reálných dat.

#### 3.5.4.1 Spuštění testovacích dotazů

Jak již bylo zmíněno v kapitole 3.1, tak mé testování bylo provedeno na JAVA SDK verze 17. Doporučuji pro spuštění testovacích dotazů použít stejnou verzi. Provedení dotazu se dělá spuštěním třídy **LuceneQueryProcessor**, kde před spuštěním je potřeba definovat argumenty pro spuštění a to způsob spuštění a cesta k SQL dotazu. Příklad konfigurace pro spuštění můžeme vidět na obrázku 3.12.




Obrázek 3.12: Příklad konfigurace pro spuštění

Způsob spuštění může být **USERULE** nebo **NORULE**, které nám definuje, jestli se má použít mé přepisovací pravidlo, nebo ne.

#### 3.5.4.2 Seznam testovacích dotazů nad umělými daty

Toto testování bylo provedeno nad tabulkou *zaci* a *zaci_small*, které jsou popsány v kapitole 3.5.1. Testovací dotazy se nachází v přílohách A a B. Tabulka *zaci* obsahuje velké množství záznamů, kde každý dotaz trvá jednotky až desítky minut, proto doporučuji testovat správnost nad menší tabulkou *zaci_small*.

Testovací dotazy rozděluji na typ A a B, kde typ A označuje dotaz napsaný pomocí window funkce a typ B dotaz napsaný pomocí self-joinu. Stejné číslo dotazu musí vracet stejný výsledek jak pro typ A, tak pro typ B a zároveň jak pro argument USERULE, tak pro argument NORULE.

Testování bylo provedeno způsobem, že se spustil dotaz typu A s argumentem USERULE, ten se porovnal s dotazem typu A s argumentem NORULE a ty se porovnaly s dotazem typu B (zde je již jedno jaký argument se použije).

Seznam testovacích dotazů s krátkým popisem je vypsán níže:

- **Q1** - Dotaz získávající součet bodů žáků s nejvyšším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali méně než 1000 bodů. Kód A.1, A.2, B.1 a B.2.
- **Q2** - Dotaz získávající součet bodů žáků s nejnižším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali více než 1000 bodů. Kód A.3, A.4, B.3 a B.4.

- **Q3** - Dotaz získávající součet bodů žáků s nejvyšším počtem bodů za každou třídu. Kód A.5, A.6, B.5 a B.6.
- **Q4** - Dotaz získávající součet bodů žáků s nejnižším počtem bodů za každou třídu. Kód A.7, A.8, B.7 a B.8.
- **Q5** - Dotaz získávající součet bodů žáků s nejvyšším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali méně než 1000 bodů. Dotaz využívá jinou window funkci oproti dotazu Q1, který má stejný výstup. Kód A.9, A.10, B.9 a B.10.
- **Q6** - Dotaz získávající součet bodů žáků s nejnižším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali více než 1000 bodů. Dotaz využívá jinou window funkci oproti dotazu Q2, který má stejný výstup. Kód A.11, A.12, B.11 a B.12.
- **Q7** - Dotaz získávající součet bodů žáků s nejvyšším počtem bodů za každou třídu. Dotaz využívá jinou window funkci oproti dotazu Q3, který má stejný výstup. Kód A.13, A.14, B.13 a B.14.
- **Q8** - Dotaz získávající součet bodů žáků s nejnižším počtem bodů za každou třídu. Dotaz využívá jinou window funkci oproti dotazu Q4, který má stejný výstup. Kód A.15, A.16, B.15 a B.16.
- **Q9** - Dotaz získávající všechny sloupce žáků s nejvyšším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali méně než 1000 bodů. Kód A.17, A.18, B.17 a B.18.
- **Q10** - Dotaz získávající všechny sloupce žáků s nejnižším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali více než 1000 bodů. Kód A.19, A.20, B.19 a B.20.
- **Q11** - Dotaz získávající všechny sloupce žáků s nejvyšším počtem bodů za každou třídu. Kód A.21, A.22, B.21 a B.22.
- **Q12** - Dotaz získávající všechny sloupce žáků s nejnižším počtem bodů za každou třídu. Kód A.23, A.24, B.23 a B.24.
- **Q13** - Dotaz získávající všechny sloupce žáků s nejvyšším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali méně než 1000 bodů. Dotaz využívá jinou window funkci oproti dotazu Q9, který má stejný výstup. Kód A.25, A.26, B.25 a B.26.
- **Q14** - Dotaz získávající všechny sloupce žáků s nejnižším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali více než 1000 bodů. Dotaz využívá jinou window funkci oproti dotazu Q10, který má stejný výstup. Kód A.27, A.28, B.27 a B.28.

- **Q15** - Dotaz získávající všechny sloupce žáků s nejvyšším počtem bodů za každou třídu. Dotaz využívá jinou window funkci oproti dotazu Q11, který má stejný výstup. Kód A.29, A.30, B.29 a B.30.

- **Q16** - Dotaz získávající všechny sloupce žáků s nejnižším počtem bodů za každou třídu. Dotaz využívá jinou window funkci oproti dotazu Q12, který má stejný výstup. Kód A.31, A.32, B.31 a B.32.

- **Q17** - Dotaz získávající všechny sloupce žáků s nejvyšším počtem bodů za každou třídu a jméno. Dotaz se vztahuje jen na žáky, kteří dostali méně než 1000 bodů. Kód A.33, A.34, B.33 a B.34.

- **Q18** - Dotaz získávající všechny sloupce žáků s nejvyšším počtem bodů za každou třídu a jméno. Kód A.35, A.36, B.35 a B.36.

Všechny testované dotazy vrátily očekávaný výsledek.

#### 3.5.4.3 Seznam testovacích dotazů nad reálnými daty

Testování nad databází TPC-H je podobné jako testování nad umělými daty výše. Testovací dotazy se nachází v příloze C. Dotazy jsou opět rozděleny na typ A a B. Typ A označuje dotaz napsaný pomocí window funkce a typ B dotaz napsaný pomocí self-joinu.

Testování bylo provedeno tak, že byl spuštěn dotaz typu A s argumentem USERULE, který byl porovnávám s dotazem A s argumentem NORULE a dotazem typu B. Testovací dotazy byly navrženy tak, aby kombinovaly v dotazu více tabulek.

Seznam testovacích dotazů s krátkým popisem naleznete níže:

- **Q1** - Dotaz získávající zemi, jméno a balanc účtu zákazníka s nejvyšším balancem účtu za každou zemi. Kód C.1 a C.2.

- **Q2** - Dotaz získávající zemi, jméno a balanc účtu zákazníka s nejnižším balancem účtu za každou zemi, kde zákazník nežije v Argentině. Kód C.3 a C.4.

- **Q3** - Dotaz získávající zemi, jméno a balanc účtu zákazníka s nejvyšším balancem účtu za každou zemi. Dotaz využívá jinou window funkci než dotaz Q1, který ale získává stejný výstup. Kód C.5 a C.6.

- **Q4** - Dotaz získávající zemi, jméno a balanc účtu zákazníka s nejnižším balancem účtu za každou zemi, kde zákazník nežije v Argentině. Dotaz využívá jinou window funkci než dotaz Q2, který ale získává stejný výstup. Kód C.7 a C.8.

- **Q5** - Dotaz získávající zemi, jméno a balanc účtu dodavatele s nejvyšším balancem účtu za každou zemi. Kód C.9 a C.10.

- **Q6** - Dotaz získávající zemi, jméno a balanc účtu dodavatele s nejnižším balancem účtu za každou zemi, kde dodavatel nežije v Argentině. Kód C.11 a C.12.

- **Q7** - Dotaz získávající zemi, jméno a balanc účtu dodavatele s nejvyšším balancem účtu za každou zemi. Dotaz využívá jinou window funkci než dotaz Q5, který ale získává stejný výstup. Kód C.13 a C.14.

- **Q8** - Dotaz získávající zemi, jméno a balanc účtu dodavatele s nejnižším balancem účtu za každou zemi, kde dodavatel nežije v Argentině. Dotaz využívá jinou window funkci než dotaz Q6, který ale získává stejný výstup. Kód C.15 a C.16.

Všechny testované dotazy vrátily očekávaný výsledek.

#### 3.5.4.4 Seznam dotazů, kdy se nemá použít přepisovací pravidlo

Vytvořil jsem také dotazy, na které se přepisovací pravidlo nemá uplatňovat. Pravidlo nedokáže zpracovat některé specifické variace použití window funkce. Tyto dotazy kontrolují, jestli se reálně pravidlo nepoužije, jelikož uplatnění by vytvořilo špatný výstup, nebo chybu. Tyto dotazy se nachází v příloze D a byly testovány s argumentem USERULE, kde zde se pravidlo nemá provést a máme získat plán s window funkcí. Popis můžeme vidět níže:

- **Q1** - Dotaz získávající součet bodů žáků s **druhým** nejvyšším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali méně než 1000 bodů. Kód D.1.

- **Q2** - Dotaz získávající součet bodů žáků s **druhým** nejvyšším počtem bodů za každou třídu. Kód D.2.

- **Q3** - Dotaz získávající součet bodů žáků s nejvyšším počtem bodů za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali méně než 1000 bodů. Výstup je následně omezený jen na žáky, kteří mají id menší než 1000. Kód D.3.

- **Q4** - Dotaz získávající součet bodů žáků s nejvyšším počtem bodů za každou třídu. Výstup je následně omezený jen na žáky, kteří mají id menší než 1000. Kód D.4.

- **Q5** - Dotaz získávající součet bodů žáků nejnižším počtem bodů a zároveň nejnižším id za každou třídu. Dotaz se vztahuje jen na žáky, kteří dostali méně než 1000 bodů. Kód D.5.

- **Q6** - Dotaz získávající součet bodů žáků nejnižším počtem bodů a zároveň nejnižším id za každou třídu. Kód D.6.

Všechny testované dotazy vrátily očekávaný plán.

# Kapitola 4

# Závěr

Diplomová práce je zaměřena na tvorbu přepisovacího pravidla v Apache Calcite, které přepisuje window funkci na self-join. V teoretické části je popsán Apache Calcite, z čeho se skládá a jak funguje. Dále je podrobně vysvětlena optimalizace dotazu, konkrétně tvorba plánů a přepisovací pravidla. Konec teoretické části je věnován window funkcím, jejich skladbě, druhům a chováním. Bylo podrobně znázorněno, že plán s použitím window funkce může být za určitých podmínek méně efektivní než plán postrádající tuto funkci.

Implementační část je věnována technickému prostředí Apache Calcite a implementaci přepisovacího pravidla. Bylo ukázáno schéma databáze, databázový procesor a tvorba Lucene indexů v prostředí Apache Calcite. U implementace přepisovacího pravidla byla popsána jednotlivá transformace každého operátoru, která se přepisovala v přepisovacím pravidle, to bylo následně otestováno na sadě testovacích dotazů, které ověřovaly správnost implementace přepisovacího pravidla. Závěrem bylo zjištěno, že použití přepisovacího pravidla na adaptéru Apache Lucene není efektivní za žádných podmínek. Naopak se ukázalo, že u systému HeavyDB je dotaz bez aplikace window funkce za určitých podmínek efektivnější než když je součástí dotazu. Závěrem tedy můžeme konstatovat, že systém HeavyDB s použitím přepisovacího pravidla prokázal svou efektivnost.

Cíl diplomové práce, vytvoření přepisovacího pravidla byl splněn. Přepisovací pravidlo korektně přepisuje plán z window funkce na self-join a umožňuje přepis i různých variant dotazu. Pravidlo bylo otestováno nad umělými i reálnými daty. Pravidlo lze implementovat do jakéhokoliv databázového systému, který využívá optimalizátor z Apache Calcite. Také pravidlo může sloužit pro inspiraci tvorby jiného podobného pravidla, případně ho rozšířit, aby fungovalo na více typů window funkcí.

# Literatura


1. *Apache Calcite* [online]. [cit. 2022-04-24]. Dostupné z: https://calcite.apache.org/.
2. NEVAREZ, Benjamin. Inside the SQL Server Query Optimizer. [B.r.], S.l.: Red Gate Books, 2011. ISBN 9781906434601.
3. *Vizualizace optimalizačního procesu v SQL Serveru* [online]. [cit. 2023-03-11]. Dostupné z: https://dspace.vsb.cz/handle/10084/144054.
4. *Adapters* [online]. [cit. 2023-02-26]. Dostupné z: https://calcite.apache.org/docs/adapter.html.
5. *Add Custom Statistics* [online]. [cit. 2023-02-26]. Dostupné z: https://lists.apache.org/thread/fzodbl93x67l57of8d98w8qok4wzsrjf.
6. *Introducing HEAVY.AI* [online]. [cit. 2023-03-11]. Dostupné z: https://www.heavy.ai/.
7. *15 Types of SQL Window Functions (With Examples)* [online]. [cit. 2023-02-25]. Dostupné z: https://analyticsexplained.com/15-types-of-sql-window-functions-with-examples/.
8. LEIS, Viktor. *Efficient Processing of Window Functions in Analytical SQL Queries* [online]. [cit. 2023-02-26]. Dostupné z: https://www.vldb.org/pvldb/vol8/p1058-leis.pdf.
9. *SELECT - OVER Clause (Transact-SQL)* [online]. [cit. 2023-02-25]. Dostupné z: https://learn.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql?view=sql-server-ver16.
10. *BOSS 2021 workshop* [online]. [cit. 2022-04-24]. Dostupné z: https://boss-workshop.github.io/boss-2021/.
11. *TPC-H Vesion 2 and Version 3* [online]. [cit. 2023-03-11]. Dostupné z: https://www.tpc.org/tpch/.
12. *TPCH* [online]. [cit. 2023-03-11]. Dostupné z: https://relational.fit.cvut.cz/dataset/TPCH.

# Příloha A

# Testovací dotazy pro tabulku *zaci*

```
1  SELECT SUM(body)
2  FROM (
3     SELECT *,
4        DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5     FROM zaci
6     WHERE body <= 1000
7  ) ranking
8  WHERE rank_value = 1
```

Kód A.1: *zaci* Q1_A

```
1  SELECT SUM(body)
2  FROM zaci c
3  JOIN (
4     SELECT trida, MAX(body) rank_value
5     FROM zaci
6     WHERE body <= 1000
7     GROUP BY trida
8  ) ranking ON c.trida = ranking.trida AND
9  c.body = ranking.rank_value
```

Kód A.2: *zaci* Q1_B

```
1  SELECT SUM(body)
2  FROM (
3     SELECT *,
4        zaci() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
5     FROM zaci
6     WHERE body >= 1000
7  ) ranking
8  WHERE rank_value = 1
```

Kód A.3: *zaci* Q2_A

```sql
SELECT SUM(body)
   FROM zaci c
   JOIN (
   SELECT trida, MIN(body) rank_value
   FROM zaci
   WHERE body >= 1000
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.4: *zaci* Q2_B

```sql
SELECT SUM(body)
from (
   SELECT *,
   RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
   FROM zaci
   WHERE body <= 1000
) ranking
WHERE rank_value = 1
```

Kód A.5: *zaci* Q3_A

```sql
SELECT SUM(body)
FROM zaci c
JOIN (
   SELECT trida, MAX(body) rank_value
   FROM zaci
   WHERE body <= 1000
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.6: *zaci* Q3_B

```sql
SELECT SUM(body)
FROM (
   SELECT *,
   RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
   FROM zaci
   WHERE body <= 1000
) ranking
WHERE rank_value = 1
```

Kód A.7: *zaci* Q4_A

```sql
SELECT SUM(body)
FROM zaci c
JOIN (
   SELECT trida, MIN(body) rank_value
   FROM zaci
   WHERE body <= 1000
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.8: *zaci* Q4_B

```sql
SELECT SUM(body)
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
   FROM zaci
) ranking
WHERE rank_value = 1
```

Kód A.9: *zaci* Q5_A

```sql
SELECT SUM(body)
FROM zaci c
JOIN (
   SELECT trida, MAX(body) rank_value
   FROM zaci
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.10: *zaci* Q5_B

```sql
SELECT SUM(body)
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
   FROM zaci
) ranking
WHERE rank_value = 1
```

Kód A.11: *zaci* Q6_A

```sql
SELECT SUM(body)
FROM zaci c
JOIN (
   SELECT trida, MIN(body) rank_value
   FROM zaci
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.12: *zaci* Q6_B

```sql
SELECT SUM(body)
FROM (
   SELECT *,
   RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
   FROM zaci
) ranking
WHERE rank_value = 1
```

Kód A.13: *zaci* Q7_A

```sql
SELECT SUM(body)
FROM zaci c
JOIN (
   SELECT trida, MAX(body) rank_value
   FROM zaci
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.14: *zaci* Q7_B

```sql
SELECT SUM(body)
FROM (
   SELECT *,
   RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
   FROM zaci
) ranking
WHERE rank_value = 1
```

Kód A.15: *zaci* Q8_A

```sql
1 SELECT SUM(body)
2 FROM zaci c
3 JOIN (
4    SELECT trida, MIN(body) rank_value
5    FROM zaci
6    GROUP BY trida
7 ) ranking ON c.trida = ranking.trida AND
8 c.body = ranking.rank_value
```

Kód A.16: *zaci* Q8_B

```sql
1 SELECT id, jmeno, trida, body
2 FROM (
3    SELECT *,
4    DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5    FROM zaci
6    WHERE body <= 1000
7 ) ranking
8 WHERE rank_value = 1
```

Kód A.17: *zaci* Q9_A

```sql
1 SELECT c.id, c.jmeno, c.trida, c.body
2 FROM zaci c
3 JOIN (
4    SELECT trida, MAX(body) rank_value
5    FROM zaci
6    WHERE body <= 1000
7    GROUP BY trida
8 ) ranking ON c.trida = ranking.trida AND
9 c.body = ranking.rank_value
```

Kód A.18: *zaci* Q9_B

```sql
1 SELECT id, jmeno, trida, body
2 FROM (
3    SELECT *,
4    DENSE_RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
5    FROM zaci
6    WHERE body <= 1000
7 ) ranking
8 WHERE rank_value = 1
```

Kód A.19: *zaci* Q10_A

```
1 SELECT c.id, c.jmeno, c.trida, c.body
2 FROM zaci c
3 JOIN (
4    SELECT trida, MIN(body) rank_value
5    FROM zaci
6    WHERE body <= 1000
7    GROUP BY trida
8 ) ranking ON c.trida = ranking.trida AND
9 c.body = ranking.rank_value
```

Kód A.20: *zaci* Q10_B

```
1 SELECT id, jmeno, trida, body
2 FROM (
3    SELECT *,
4    RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5    FROM zaci
6    WHERE body <= 1000
7 ) ranking
8 WHERE rank_value = 1
```

Kód A.21: *zaci* Q11_A

```
1 SELECT c.id, c.jmeno, c.trida, c.body
2 FROM zaci c
3 JOIN (
4    SELECT trida, MAX(body) rank_value
5    FROM zaci
6    WHERE body <= 1000
7    GROUP BY trida
8 ) ranking ON c.trida = ranking.trida AND
9 c.body = ranking.rank_value
```

Kód A.22: *zaci* Q11_B

```
1 SELECT id, jmeno, trida, body
2 FROM (
3    SELECT *,
4    RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
5    FROM zaci
6    WHERE body <= 1000
7 ) ranking
8 WHERE rank_value = 1
```

Kód A.23: *zaci* Q12_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci c
JOIN (
   SELECT trida, MIN(body) rank_value
   FROM zaci
   WHERE body <= 1000
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.24: *zaci* Q12_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
   FROM zaci
) ranking
WHERE rank_value = 1
```

Kód A.25: *zaci* Q13_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci c
JOIN (
   SELECT trida, MAX(body) rank_value
   FROM zaci
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.26: *zaci* Q13_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
   FROM zaci
) ranking
WHERE rank_value = 1
```

Kód A.27: *zaci* Q14_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci c
JOIN (
   SELECT trida, MIN(body) rank_value
   FROM zaci
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.28: *zaci* Q14_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
   FROM zaci
) ranking
WHERE rank_value = 1
```

Kód A.29: *zaci* Q15_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci c
JOIN (
   SELECT trida, MAX(body) rank_value
   FROM zaci
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.30: *zaci* Q15_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
   FROM zaci
) ranking
WHERE rank_value = 1
```

Kód A.31: *zaci* Q16_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci c
JOIN (
   SELECT trida, MIN(body) rank_value
   FROM zaci
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód A.32: *zaci* Q16_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida, jmeno ORDER BY body DESC) rank_value
   FROM zaci
   WHERE body <= 1000
) ranking
WHERE rank_value = 1
```

Kód A.33: *zaci* Q17_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci c
JOIN (
   SELECT trida, jmeno, MAX(body) rank_value
   FROM zaci
   WHERE body <= 1000
   GROUP BY trida, jmeno
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
AND c.jmeno = ranking.jmeno
```

Kód A.34: *zaci* Q17_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida, jmeno ORDER BY body DESC) rank_value
   FROM zaci
) ranking
WHERE rank_value = 1
```

Kód A.35: *zaci* Q18_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci c
JOIN (
   SELECT trida, jmeno, MAX(body) rank_value
   FROM zaci
   GROUP BY trida, jmeno
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
AND c.jmeno = ranking.jmeno
```

Kód A.36: *zaci* Q18_B

# Příloha B

# Testovací dotazy pro tabulku *zaci_small*

```
1 SELECT SUM(body)
2 FROM (
3    SELECT *,
4       DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5    FROM zaci_small
6    WHERE body <= 100000
7 ) ranking
8 WHERE rank_value = 1
```

Kód B.1: *zaci_small* Q1_A

```
1 SELECT SUM(body)
2 FROM zaci_small c
3 JOIN (
4    SELECT trida, MAX(body) rank_value
5    FROM zaci_small
6    WHERE body <= 100000
7    GROUP BY trida
8 ) ranking ON c.trida = ranking.trida AND
9 c.body = ranking.rank_value
```

Kód B.2: *zaci_small* Q1_B

```
1 SELECT SUM(body)
2 FROM (
3    SELECT *,
4       zaci_small() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
5    FROM zaci
6    WHERE body >= 100000
7 ) ranking
8 WHERE rank_value = 1
```

Kód B.3: *zaci_small* Q2_A

```sql
1  SELECT SUM(body)
2     FROM zaci_small c
3     JOIN (
4     SELECT trida, MIN(body) rank_value
5     FROM zaci_small
6     WHERE body >= 100000
7     GROUP BY trida
8  ) ranking ON c.trida = ranking.trida AND
9  c.body = ranking.rank_value
```

Kód B.4: *zaci_small* Q2_B

```sql
1  SELECT SUM(body)
2  from (
3     SELECT *,
4     RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5     FROM zaci_small
6     WHERE body <= 100000
7  ) ranking
8  WHERE rank_value = 1
```

Kód B.5: *zaci_small* Q3_A

```sql
1  SELECT SUM(body)
2  FROM zaci_small c
3  JOIN (
4     SELECT trida, MAX(body) rank_value
5     FROM zaci_small
6     WHERE body <= 100000
7     GROUP BY trida
8  ) ranking ON c.trida = ranking.trida AND
9  c.body = ranking.rank_value
```

Kód B.6: *zaci_small* Q3_B

```sql
1  SELECT SUM(body)
2  FROM (
3     SELECT *,
4     RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
5     FROM zaci_small
6     WHERE body <= 100000
7  ) ranking
8  WHERE rank_value = 1
```

Kód B.7: *zaci_small* Q4_A

```sql
1 SELECT SUM(body)
2 FROM zaci_small c
3 JOIN (
4    SELECT trida, MIN(body) rank_value
5    FROM zaci_small
6    WHERE body <= 100000
7    GROUP BY trida
8 ) ranking ON c.trida = ranking.trida AND
9 c.body = ranking.rank_value
```

Kód B.8: *zaci_small* Q4_B

```sql
1 SELECT SUM(body)
2 FROM (
3    SELECT *,
4    DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5    FROM zaci_small
6 ) ranking
7 WHERE rank_value = 1
```

Kód B.9: *zaci_small* Q5_A

```sql
1 SELECT SUM(body)
2 FROM zaci_small c
3 JOIN (
4    SELECT trida, MAX(body) rank_value
5    FROM zaci_small
6    GROUP BY trida
7 ) ranking ON c.trida = ranking.trida AND
8 c.body = ranking.rank_value
```

Kód B.10: *zaci_small* Q5_B

```sql
1 SELECT SUM(body)
2 FROM (
3    SELECT *,
4    DENSE_RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
5    FROM zaci_small
6 ) ranking
7 WHERE rank_value = 1
```

Kód B.11: *zaci_small* Q6_A

```
1 SELECT SUM(body)
2 FROM zaci_small c
3 JOIN (
4    SELECT trida, MIN(body) rank_value
5    FROM zaci_small
6    GROUP BY trida
7 ) ranking ON c.trida = ranking.trida AND
8 c.body = ranking.rank_value
```

Kód B.12: *zaci_small* Q6_B

```
1 SELECT SUM(body)
2 FROM (
3    SELECT *,
4    RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5    FROM zaci_small
6 ) ranking
7 WHERE rank_value = 1
```

Kód B.13: *zaci_small* Q7_A

```
1 SELECT SUM(body)
2 FROM zaci_small c
3 JOIN (
4    SELECT trida, MAX(body) rank_value
5    FROM zaci_small
6    GROUP BY trida
7 ) ranking ON c.trida = ranking.trida AND
8 c.body = ranking.rank_value
```

Kód B.14: *zaci_small* Q7_B

```
1 SELECT SUM(body)
2 FROM (
3    SELECT *,
4    RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
5    FROM zaci_small
6 ) ranking
7 WHERE rank_value = 1
```

Kód B.15: *zaci_small* Q8_A

```sql
1 SELECT SUM(body)
2 FROM zaci_small c
3 JOIN (
4    SELECT trida, MIN(body) rank_value
5    FROM zaci_small
6    GROUP BY trida
7 ) ranking ON c.trida = ranking.trida AND
8 c.body = ranking.rank_value
```

Kód B.16: *zaci_small* Q8_B

```sql
1 SELECT id, jmeno, trida, body
2 FROM (
3    SELECT *,
4    DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5    FROM zaci_small
6    WHERE body <= 100000
7 ) ranking
8 WHERE rank_value = 1
```

Kód B.17: *zaci_small* Q9_A

```sql
1 SELECT c.id, c.jmeno, c.trida, c.body
2 FROM zaci_small c
3 JOIN (
4    SELECT trida, MAX(body) rank_value
5    FROM zaci_small
6    WHERE body <= 100000
7    GROUP BY trida
8 ) ranking ON c.trida = ranking.trida AND
9 c.body = ranking.rank_value
```

Kód B.18: *zaci_small* Q9_B

```sql
1 SELECT id, jmeno, trida, body
2 FROM (
3    SELECT *,
4    DENSE_RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
5    FROM zaci_small
6    WHERE body <= 100000
7 ) ranking
8 WHERE rank_value = 1
```

Kód B.19: *zaci_small* Q10_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci_small c
JOIN (
   SELECT trida, MIN(body) rank_value
   FROM zaci_small
   WHERE body <= 100000
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód B.20: *zaci_small* Q10_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
   FROM zaci_small
   WHERE body <= 100000
) ranking
WHERE rank_value = 1
```

Kód B.21: *zaci_small* Q11_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci_small c
JOIN (
   SELECT trida, MAX(body) rank_value
   FROM zaci_small
   WHERE body <= 100000
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód B.22: *zaci_small* Q11_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
   FROM zaci_small
   WHERE body <= 100000
) ranking
WHERE rank_value = 1
```

Kód B.23: *zaci_small* Q12_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci_small c
JOIN (
   SELECT trida, MIN(body) rank_value
   FROM zaci_small
   WHERE body <= 100000
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód B.24: *zaci_small* Q12_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
   FROM zaci_small
) ranking
WHERE rank_value = 1
```

Kód B.25: *zaci_small* Q13_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci_small c
JOIN (
   SELECT trida, MAX(body) rank_value
   FROM zaci_small
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód B.26: *zaci_small* Q13_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
   FROM zaci_small
) ranking
WHERE rank_value = 1
```

Kód B.27: *zaci_small* Q14_A

```
1 SELECT c.id, c.jmeno, c.trida, c.body
2 FROM zaci_small c
3 JOIN (
4    SELECT trida, MIN(body) rank_value
5    FROM zaci_small
6    GROUP BY trida
7 ) ranking ON c.trida = ranking.trida AND
8 c.body = ranking.rank_value
```

Kód B.28: *zaci_small* Q14_B

```
1 SELECT id, jmeno, trida, body
2 FROM (
3    SELECT *,
4    RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5    FROM zaci_small
6 ) ranking
7 WHERE rank_value = 1
```

Kód B.29: *zaci_small* Q15_A

```
1 SELECT c.id, c.jmeno, c.trida, c.body
2 FROM zaci_small c
3 JOIN (
4    SELECT trida, MAX(body) rank_value
5    FROM zaci_small
6    GROUP BY trida
7 ) ranking ON c.trida = ranking.trida AND
8 c.body = ranking.rank_value
```

Kód B.30: *zaci_small* Q15_B

```
1 SELECT id, jmeno, trida, body
2 FROM (
3    SELECT *,
4    RANK() OVER (PARTITION BY trida ORDER BY body ASC) rank_value
5    FROM zaci_small
6 ) ranking
7 WHERE rank_value = 1
```

Kód B.31: *zaci_small* Q16_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci_small c
JOIN (
   SELECT trida, MIN(body) rank_value
   FROM zaci_small
   GROUP BY trida
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
```

Kód B.32: *zaci_small* Q16_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida, jmeno ORDER BY body DESC) rank_value
   FROM zaci_small
   WHERE body <= 100000
) ranking
WHERE rank_value = 1
```

Kód B.33: *zaci_small* Q17_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci_small c
JOIN (
   SELECT trida, jmeno, MAX(body) rank_value
   FROM zaci_small
   WHERE body <= 100000
   GROUP BY trida, jmeno
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
AND c.jmeno = ranking.jmeno
```

Kód B.34: *zaci_small* Q17_B

```sql
SELECT id, jmeno, trida, body
FROM (
   SELECT *,
   DENSE_RANK() OVER (PARTITION BY trida, jmeno ORDER BY body DESC) rank_value
   FROM zaci_small
) ranking
WHERE rank_value = 1
```

Kód B.35: *zaci_small* Q18_A

```sql
SELECT c.id, c.jmeno, c.trida, c.body
FROM zaci_small c
JOIN (
   SELECT trida, jmeno, MAX(body) rank_value
   FROM zaci_small
   GROUP BY trida, jmeno
) ranking ON c.trida = ranking.trida AND
c.body = ranking.rank_value
AND c.jmeno = ranking.jmeno
```

Kód B.36: *zaci_small* Q18_B

# Příloha C

# Testovací dotazy pro databázi *TPC-H*

```
1  SELECT n.n_name, x.c_name, x.c_acctbal
2  FROM (
3     SELECT c_nationkey, c_name, c_acctbal
4     FROM (
5        SELECT *,
6        DENSE_RANK() OVER (PARTITION BY c_nationkey ORDER BY c_acctbal DESC) rank_value
7        FROM tpch_customer
8     ) ranking
9     WHERE rank_value = 1
10 ) x
11 JOIN tpch_nation n ON x.c_nationkey = n.n_nationkey
```

Kód C.1: *TPC-H* Q1_A

```
1  SELECT n.n_name, x.c_name, x.c_acctbal
2  FROM (
3     SELECT c.c_nationkey, c.c_name, c.c_acctbal
4     FROM tpch_customer c
5     JOIN (
6        SELECT c_nationkey, MAX(c_acctbal) rank_value
7        FROM tpch_customer
8        GROUP BY c_nationkey
9     ) ranking ON c.c_nationkey = ranking.c_nationkey AND
10    c.c_acctbal = ranking.rank_value
11 ) x
12 JOIN tpch_nation n ON x.c_nationkey = n.n_nationkey
```

Kód C.2: *TPC-H* Q1_B

```sql
SELECT n.n_name, x.c_name, x.c_acctbal
FROM (
   SELECT c_nationkey, c_name, c_acctbal
   FROM (
      SELECT *,
      DENSE_RANK() OVER (PARTITION BY c_nationkey ORDER BY c_acctbal ASC) rank_value
      FROM tpch_customer
      WHERE c_nationkey != 1
   ) ranking
   WHERE rank_value = 1
) x
JOIN tpch_nation n ON x.c_nationkey = n.n_nationkey
```

Kód C.3: *TPC-H* Q2_A

```sql
SELECT n.n_name, x.c_name, x.c_acctbal
FROM (
   SELECT c.c_nationkey, c.c_name, c.c_acctbal
   FROM tpch_customer c
   JOIN (
      SELECT c_nationkey, MIN(c_acctbal) rank_value
      FROM tpch_customer
      WHERE c_nationkey != 1
      GROUP BY c_nationkey
   ) ranking ON c.c_nationkey = ranking.c_nationkey AND
   c.c_acctbal = ranking.rank_value
) x
JOIN tpch_nation n ON x.c_nationkey = n.n_nationkey
```

Kód C.4: *TPC-H* Q2_B

```sql
SELECT n.n_name, x.c_name, x.c_acctbal
FROM (
   SELECT c_nationkey, c_name, c_acctbal
   FROM (
      SELECT *,
      RANK() OVER (PARTITION BY c_nationkey ORDER BY c_acctbal DESC) rank_value
      FROM tpch_customer
   ) ranking
   WHERE rank_value = 1
) x
JOIN tpch_nation n ON x.c_nationkey = n.n_nationkey
```

Kód C.5: *TPC-H* Q3_A

```sql
1  SELECT n.n_name, x.c_name, x.c_acctbal
2  FROM (
3     SELECT c.c_nationkey, c.c_name, c.c_acctbal
4     FROM tpch_customer c
5     JOIN (
6        SELECT c_nationkey, MAX(c_acctbal) rank_value
7        FROM tpch_customer
8        GROUP BY c_nationkey
9     ) ranking ON c.c_nationkey = ranking.c_nationkey AND
10    c.c_acctbal = ranking.rank_value
11 ) x
12 JOIN tpch_nation n ON x.c_nationkey = n.n_nationkey
```

Kód C.6: *TPC-H* Q3_B

```sql
1  SELECT n.n_name, x.c_name, x.c_acctbal
2  FROM (
3     SELECT c_nationkey, c_name, c_acctbal
4     FROM (
5        SELECT *,
6        RANK() OVER (PARTITION BY c_nationkey ORDER BY c_acctbal ASC) rank_value
7        FROM tpch_customer
8        WHERE c_nationkey not in (1)
9     ) ranking
10    WHERE rank_value = 1
11 ) x
12 JOIN tpch_nation n ON x.c_nationkey = n.n_nationkey
```

Kód C.7: *TPC-H* Q4_A

```sql
1  SELECT n.n_name, x.c_name, x.c_acctbal
2  FROM (
3     SELECT c.c_nationkey, c.c_name, c.c_acctbal
4     FROM tpch_customer c
5     JOIN (
6        SELECT c_nationkey, MIN(c_acctbal) rank_value
7        FROM tpch_customer
8        WHERE c_nationkey not in (1)
9        GROUP BY c_nationkey
10    ) ranking ON c.c_nationkey = ranking.c_nationkey AND
11    c.c_acctbal = ranking.rank_value
12 ) x
13 JOIN tpch_nation n ON x.c_nationkey = n.n_nationkey
```

Kód C.8: *TPC-H* Q4_B

```sql
SELECT n.n_name, x.s_name, x.s_acctbal
FROM (
   SELECT s_nationkey, s_name, s_acctbal
   FROM (
      SELECT *,
      DENSE_RANK() OVER (PARTITION BY s_nationkey ORDER BY s_acctbal DESC) rank_value
      FROM tpch_supplier
   ) ranking
   WHERE rank_value = 1
) x
JOIN tpch_nation n ON x.s_nationkey = n.n_nationkey
```

Kód C.9: *TPC-H* Q5_A

```sql
SELECT n.n_name, x.s_name, x.s_acctbal
FROM (
   SELECT c.s_nationkey, c.s_name, c.s_acctbal
   FROM tpch_supplier c
   JOIN (
      SELECT s_nationkey, MAX(s_acctbal) rank_value
      FROM tpch_supplier
      GROUP BY s_nationkey
   ) ranking ON c.s_nationkey = ranking.s_nationkey AND
   c.s_acctbal = ranking.rank_value
) x
JOIN tpch_nation n ON x.s_nationkey = n.n_nationkey
```

Kód C.10: *TPC-H* Q5_B

```sql
SELECT n.n_name, x.s_name, x.s_acctbal
FROM (
   SELECT s_nationkey, s_name, s_acctbal
   FROM (
      SELECT *,
      DENSE_RANK() OVER (PARTITION BY s_nationkey ORDER BY s_acctbal DESC) rank_value
      FROM tpch_supplier
      WHERE s_nationkey not in (1)
   ) ranking
   WHERE rank_value = 1
) x
JOIN tpch_nation n ON x.s_nationkey = n.n_nationkey
```

Kód C.11: *TPC-H* Q6_A

```sql
SELECT n.n_name, x.s_name, x.s_acctbal
FROM (
   SELECT c.s_nationkey, c.s_name, c.s_acctbal
   FROM tpch_supplier c
   JOIN (
      SELECT s_nationkey, MAX(s_acctbal) rank_value
      FROM tpch_supplier
      WHERE s_nationkey not in (1)
      GROUP BY s_nationkey
   ) ranking ON c.s_nationkey = ranking.s_nationkey AND
   c.s_acctbal = ranking.rank_value
) x
JOIN tpch_nation n ON x.s_nationkey = n.n_nationkey
```

Kód C.12: *TPC-H* Q6_B

```sql
SELECT n.n_name, x.s_name, x.s_acctbal
FROM (
   SELECT s_nationkey, s_name, s_acctbal
   FROM (
      SELECT *,
      RANK() OVER (PARTITION BY s_nationkey ORDER BY s_acctbal DESC) rank_value
      FROM tpch_supplier
   ) ranking
   WHERE rank_value = 1
) x
JOIN tpch_nation n ON x.s_nationkey = n.n_nationkey
```

Kód C.13: *TPC-H* Q7_A

```sql
SELECT n.n_name, x.s_name, x.s_acctbal
FROM (
   SELECT c.s_nationkey, c.s_name, c.s_acctbal
   FROM tpch_supplier c
   JOIN (
      SELECT s_nationkey, MAX(s_acctbal) rank_value
      FROM tpch_supplier
      GROUP BY s_nationkey
   ) ranking ON c.s_nationkey = ranking.s_nationkey AND
   c.s_acctbal = ranking.rank_value
) x
JOIN tpch_nation n ON x.s_nationkey = n.n_nationkey
```

Kód C.14: *TPC-H* Q7_B

```sql
SELECT n.n_name, x.s_name, x.s_acctbal
FROM (
   SELECT s_nationkey, s_name, s_acctbal
   FROM (
      SELECT *,
      RANK() OVER (PARTITION BY s_nationkey ORDER BY s_acctbal DESC) rank_value
      FROM tpch_supplier
      WHERE s_nationkey not in (1)
   ) ranking
   WHERE rank_value = 1
) x
JOIN tpch_nation n ON x.s_nationkey = n.n_nationkey
```

Kód C.15: *TPC-H* Q8_A

```sql
SELECT n.n_name, x.s_name, x.s_acctbal
FROM (
   SELECT c.s_nationkey, c.s_name, c.s_acctbal
   FROM tpch_supplier c
   JOIN (
      SELECT s_nationkey, MAX(s_acctbal) rank_value
      FROM tpch_supplier
      WHERE s_nationkey not in (1)
      GROUP BY s_nationkey
   ) ranking ON c.s_nationkey = ranking.s_nationkey AND
   c.s_acctbal = ranking.rank_value
) x
JOIN tpch_nation n ON x.s_nationkey = n.n_nationkey
```

Kód C.16: *TPC-H* Q8_B

# Příloha D

# Testovací dotazy, kdy se nemá použít přepisovací pravidlo

```
1 SELECT sum(body)
2 from (
3    select *,
4    DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5    from zaci_small
6    where body <= 1000
7 ) ranking
8 where rank_value = 2
```

Kód D.1: nepoužití přepisovacího pravidla Q1

```
1 SELECT sum(body)
2 from (
3    select *,
4    DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5    from zaci_small
6 ) ranking
7 where rank_value = 2
```

Kód D.2: nepoužití přepisovacího pravidla Q2

```
1  SELECT sum(body)
2  from (
3     select *,
4     DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5     from zaci_small
6     where body <= 10000000
7  ) ranking
8  where rank_value = 1 and id < 1000
```

Kód D.3: nepoužití přepisovacího pravidla Q3

```
1  SELECT sum(body)
2  from (
3     select *,
4     DENSE_RANK() OVER (PARTITION BY trida ORDER BY body DESC) rank_value
5     from zaci_small
6  ) ranking
7  where rank_value = 1 and id < 1000
```

Kód D.4: nepoužití přepisovacího pravidla Q4

```
1  SELECT sum(body)
2  from (
3     select *,
4     DENSE_RANK() OVER (PARTITION BY trida ORDER BY body, id) rank_value
5     from zaci_small
6     where body <= 1000
7  ) ranking
8  where rank_value = 1
```

Kód D.5: nepoužití přepisovacího pravidla Q5

```
1  SELECT sum(body)
2  from (
3     select *,
4     DENSE_RANK() OVER (PARTITION BY trida ORDER BY body, id) rank_value
5     from zaci_small
6  ) ranking
7  where rank_value = 1
```

Kód D.6: nepoužití přepisovacího pravidla Q6

# Příloha E

# Elektronická příloha

Součástí práce je elektronická příloha, která obsahuje:

- Zdrojový kód aplikace
- Data pro naplnění databáze TPC-H
- Data pro naplnění tabulky *zaci*
- Data pro naplnění tabulky *zaci_small*