VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Vizualizace optimalizačního procesu v SQL Serveru

Vizualization Tool of SQL Server Optimization Process

Vít Maňásek

Bakalářská práce

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

Ostrava, 2021

Abstrakt

Základem každého pokročilého databázového systému je tzv. optimalizátor, který se snaží nalézt co nejefektivnější plán vykonání pro vstupní SQL příkaz. Proces optimalizace využívá tzv. memo struktury, která celý proces zefektivňuje. SQL Server umožňuje vypsat základní podobu memo struktury v textové podobě. Memo struktura může být relativně velká a textová podoba bývá nepřehledná i pro relativně jednoduché SQL příkazy. Cílem této práce je grafické zobrazení memo struktury na základě textových výstupů SQL Serveru.

Klíčová slova

SQL Server; SQL dotaz; optimalizace; přepisovací pravidla; plán vykonání dotazu; memo struktura

Abstract

The basis of every advanced database system is the so-called optimizer, which tries to find the most efficient execution plan for the input SQL statement. The optimization process uses a so-called memo structure, which streamlines the entire process. SQL Server allows you to list the basic form of the memo structure in text form. The memo structure can be relatively large and the text is usually confusing even for relatively simple SQL statements. The aim of this work is a graphical representation of the memo structure based on the text outputs of SQL Server.

Keywords

SQL Server; SQL query; optimization; transformation rules; query execution plan; memo structure

Poděkování

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

Obsah

Seznam použitých zkratek a symbolů

SQL	–	Structured Query Language
CPU	–	Central Processing Unit
I/O	–	Input/Output

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Účelem optimalizátoru je poskytnout plán provedení pro vložený SQL příkaz. Plán generovaný optimalizátorem je strom skládající se z jednotlivých operací/*operátorů*, které vypočítají daný dotaz. Za tímto účelem generuje možné *alternativní plány*. Plány, které nás dovedou ke stejnému výsledku, ale jiným způsobem.

Jak ale můžeme zjistit, jakým způsobem optimalizátor došel k optimálnímu plánu? Informace o tom, jak optimalizátor postupoval, jsou uloženy v memo struktuře. SQL server umožňuje vypsat část memo struktury v textové podobě, nicméně z textové podoby je velmi těžké vyčíst jakékoliv srozumitelné informace, a to i pro jednoduché dotazy.

Cílem mé práce je získat informace z memo struktury a graficky je znázornit do přehledné podoby. Aplikace zobrazí finální plán dotazu s průběhem jak optimalizátor došel k finálnímu plánu. Dále aplikace bude vyobrazovat přepisovací pravidla, která vedla ke změně plánu.

V kapitole číslo 2 popisuji jednotlivé části SQL serveru, kde podrobněji popisuji optimalizaci dotazu. U optimalizace dotazu vysvětluji, k čemu slouží statistiky, co jsou to přepisovací pravidla a memo struktura, a jak pomocí *hints* dokážeme získat informace o provedení dotazu.

V kapitole číslo 3 se věnuji popisu implementaci mé webové aplikace. Jakým způsobem zpracovávám vstup pro zobrazení diagramů a jaké všechny funkce aplikace má.

Kapitola 2

Teoretická část

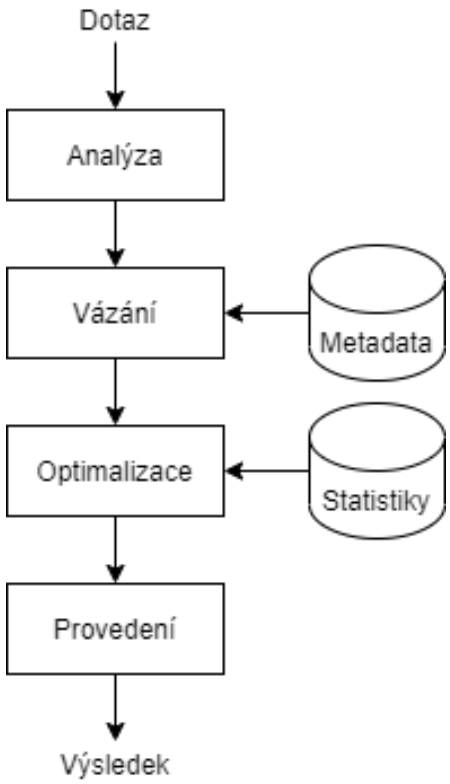
Optimalizátor dotazů SQL serveru je optimalizátor založený na ceně. Cenu bereme jako jednotku náročnosti na výpočet. Proto generuje jednotlivé plány neboli způsoby vypočítání dotazu, u kterých určuje jejich cenu. Výsledné plány pak následně porovná mezi sebou a vybere plán s nejmenší cenou. Tím se rozumí plán, který se provede nejrychleji. Vyhledávání jednotlivých plánů má také svou cenu. Z toho důvodu by bylo cenově neefektivní vyhledat pro každý dotaz všechny možné plány. Optimalizátor se proto snaží najít rovnováhu mezi dobou optimalizace a kvalitou vybraného plánu.

Z toho důvodu optimalizátor je komponenta, která má velký dopad na výkon SQL serveru. Koneckonců výběr správného plánu může znamenat rozdíl v milisekundách, v minutách, ale klidně i v řádech hodin.

2.1 části SQL serveru

V SQL Serveru existuje **storage engine** a **query processor**, kteří se podílejí na vykonání SQL příkazů. *Storage engine* provádí CRUD operace na data z disku. *Query processor* přijímá všechny dotazy odeslané na SQL server, nalezne plán pro jeho provedení, poté provede plán a poskytne požadované výsledky.

Dotazy se odesílají na SQL server pomocí jazyka SQL, který je *high-level declarative language* [1]. Tento jazyk pouze definuje co má být výstupem SQL dotazu. Již ale nedefinuje kroky, jakým způsobem se potřebné data získají z databáze. Jednotlivé dotazy se následně zpracovávají v krocích, které naleznete na obrázku 2.1.



Obrázek 2.1: Kroky zpracování dotazu

- **Analýza a vázání** - jsou běžně mezi prvními operacemi, které SQL server provede při zadání dotazu do databáze. *Analýza* první zkontroluje, jestli dotaz má platnou syntaxi a poté z použitých informací v dotazu vytvoří **logický strom**. Tím se myslí rozdělení SQL dotazu na logické operátory. Podrobnější popis logického stromu můžete nalézt v kapitole 2.1.1. *Analýza* kontroluje pouze správnost SQL syntaxe. Nekontroluje již, jestli je zadaný správný název tabulky nebo název sloupce. Tuto činnost má na starost *vázání*, které naváže údaje z dotazu na tabulky a sloupce v databázi.

- **Optimalizace dotazu** - má následně na starost vybrat plán, který se bude blížit tomu nejefektivnějšímu. Tento proces se skládá z těchto dvou kroků:

  - **vygenerování možných plánů** - I pro relativně jednoduchý dotaz může existovat mnoho způsobů výpočtů (alternativních plánů), které dosáhnou stejného výsledku. Hlavním kritériem alternativních plánů tedy je, aby byly ekvivalentní. Pro vytvoření alternativního plánu optimalizátor transformuje logické operátory z logického stromu pomocí **přepisovacích pravidel** na jiné logické, nebo fyzické operátory, které provádí stejnou činnost. Optimalizátor musí převézt všechny logické operátory na fyzické operátory

(Merge join, Index seek,...), aby u nich pak mohl následně vypočítat odhadovanou cenu. Podrobnější popis generování možných plánů se nachází v kapitole 2.4.

  - **vypočítání ceny jednotlivých plánů** - chceme-li odhadnout cenu plánu, musíme určit ceny všech fyzických operátorů, jež plán obsahuje. Optimalizátor k tomu používá cenové vzorce, které zohledňují využití zdrojů, jako je I/O, CPU a paměť. Tento cenový odhad závisí převážně na algoritmu, který určitý fyzický operátor využívá. Tak stejně cenový odhad závisí i na odhadovaném počtu záznamů, které bude muset zpracovat. Odhad počtu záznamů se pak nazývá **odhad mohutnosti**. Více o odhadu mohutnosti se nachází v kapitole 2.2. Jakmile optimalizátor určí odhadovanou cenu pro všechny plány, tak vybere plán s nejmenší cenou, který pak předá dál do *execution engine*.

- **Provedení dotazu** - jakmile je dotaz optimalizován, výsledný plán je proveden v *execution engine* k načtení požadovaných dat. Vygenerovaný finální plán může být následně uložen v paměti (*plan cache*), aby mohl být použit, jestliže stejný dotaz bude chtít uživatel provést znovu. Pokud je v paměti uložený platný plán pro náš dotaz, pak lze přeskočit krok pro kompilaci SQL příkazů a vypočítání jejich ceny. [2]

  Opětovné použití plánu však nemusí být vždy ideálním řešením pro daný dotaz. A to převážně v závislosti na parametrech v dotazu. Může to nastat například v případě, když v jednom dotazu určíme parametr, který nám omezí výstup na 50 výsledků (2.1) a v následujícím stejném dotazu změníme parametr, aby výstup měl počet výsledků v řádech milionů (2.2). Tím pádem ideální plán pro dotaz s 5 výsledky už nemusí být ideální pro dotaz s miliony výsledků. Tento problém se nazývá *parameter sniffing* a je znám převážně u procedur se vstupem [3].

```
DECLARE @parameter INT
SET @parameter = 'Vatikan'
SELECT *
FROM Customer
WHERE Country = @parameter
```

Kód 2.1: SQL vracející 50 výsledků

```
DECLARE @parameter INT
SET @parameter = 'China'
SELECT *
FROM Customer
WHERE Country = @parameter
```

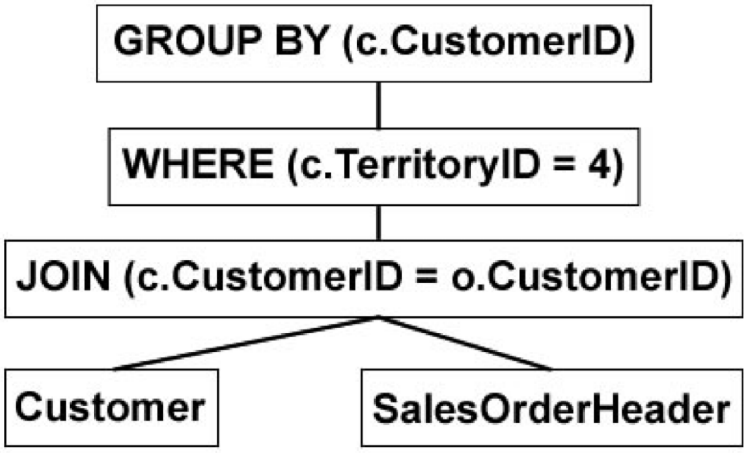
Kód 2.2: SQL vracející miliony výsledků

2.1.1 Logický strom

Jak už jsem výše zmínil, analýza a vázání vytváří **logický strom**, který se skládá z **logických operátorů**. Popis logických operátorů můžeme nalézt v kapitole 2.3. Pod pojmem logický strom si můžeme představit základní rozdělení dotazu na jednotlivé části, jako je třeba *join*, zvolení tabulky, podmínka *where* a podobně. Vizuální příklad, jak logický strom může vypadat, můžeme vidět na obrázku 2.2.

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

Kód 2.3: SQL pro obrázek 2.2



Obrázek 2.2: Logický strom [2]

2.2 Statistiky a určení ceny

Jak už jsem výše uvedl, optimalizátor dotazů SQL serveru je optimalizátor založený na ceně, a tedy kvalita plánů, které generuje, je přímo závislá na přesnosti odhadu jejich ceny. Stejným způsobem je odhadovaná cena založena na algoritmech, nebo použitých operátorech a jejich odhadu mohutnosti. Tedy pro správný odhad ceny je potřeba co nejpřesněji odhadnout počet záznamů vrácených dotazem.

SQL server vytváří a aktualizuje **statistiky**, aby pomohl optimalizátoru určit **odhad mohutnosti**, tedy odhadovaný počet výsledků. Odhad mohutnosti se určuje pro každý fyzický operátor v plánu zvlášť. Co je to fyzický operátor, je vysvětleno v následující kapitole 2.3.

2.2.1 Tvorba statistik

Statistiky jsou generovány v několika různých případech. Buď automaticky vytvořená optimalizátorem, nebo při tvorbě indexu a nebo ručně vytvořený. Statistiky můžeme rozdělit do dvou komponent:

- **hustota** - jak už nám název říká, určuje hustotu znaků v jednotlivých sloupcích v tabulce. Pomocí hustoty můžeme zjistit jak moc se opakují hodnoty v určitých sloupcích a to nám dopomůže vylepšit odhadu počtu řádků například při použití GROUP BY operace.
- **histogram** - je statistika četnosti určité hodnoty. Díky histogramu jsme schopni zjistit odhad mohutnosti řádků pro konkrétní hodnotu ve sloupci. Histogram se využívá při agregaci pro zjištění počtu řádků pro každou skupinu, nebo při použití $=$, $<$, $>$, LIKE, apod. operátorů v podmínce.

Histogram oproti hustotě je náročnější na vytvoření a na uložiště. Všechny statistiky je důležité aktualizovat, aby odhad mohutnosti zůstával co nejpřesnější. Pokud SQL server zjistí při optimalizaci dotazu, že statistiky pro výpočet daného dotazu je zastaralý, tak automaticky aktualizuje statistiky.

2.3 Typy operátorů

Základní operátory v plánu, se kterým pracuje optimalizátor, jsou tyto:

- **Logické operátory** - popisují relační algebraickou operaci použitou ke zpracování příkazu. Jinými slovy, logické operátory koncepčně popisují jakou operaci je třeba provést. Už ale neurčují jaký algoritmus bude pro dosažení výsledku použit. Například logický operátor *join*, který popisuje, že se mají spojit dvě tabulky. Již ale nepopisuje, jaký algoritmus se má použít, pro spojení. Vizuální zobrazení logických operátorů můžeme vidět na obrázku 2.2 logického stromu.
- **Fyzické operátory** - implementují operaci popsanou logickými operátory. Každý fyzický operátor je již konkrétní algoritmus s jasně definovanými výstupy pro dané vstupy. Například některé fyzické operátory přistupují ke sloupcům, nebo řádkům z tabulky, indexu, nebo pohledu. Ostatní fyzické operátory provádějí další operace, jako jsou výpočty, agregace a spojení tabulek. Například fyzický operátor Hash join, který už definuje přesný algoritmus spojení dvou tabulek. Příklady jednotlivých fyzických operátorů pro operaci *join* a group aggregation jsou popsány v kapitole 2.4.2.

Pro každý fyzický operátor určuje optimalizátor jeho odhadovanou cenu. Pro logické operátory se cena nedá odhadnout, jelikož cena se dá odhadnout jen pokud víme přesný algoritmus operátoru.

2.4 Přepisovací pravidla

V kapitole 1 jsem mluvil o tom, že optimalizátor hledá alternativní plány pro každý dotaz. Pro vyhledání alternativních plánů optimalizátor používá přepisovací pravidla. Přepisovací pravidla, jak už název může napovídat, přepisují jednotlivé části plánu na jinou ekvivalentní část. Části plánu, které optimalizátor přepisuje, jsou logické operátory. Optimalizátor má na vstupu logický strom skládající se z logických operátorů. Ty může buď převést na jiný ekvivalentní logický operátor, nebo přímo už na určitý fyzický operátor. Na fyzické operátory už ale nemůžeme uplatnit žádné přepisovací pravidlo. Ve výsledném plánu následně zbydou jen fyzické operátory, aby optimalizátor mohl určit cenu plánu a porovnat ho s jinými plány.

Přepisovací pravidla se skládají ze dvou částí: **vzor** a **náhrada**. Vzor je logický operátor pro přepsání. Náhrada je přepsaná ekvivalentní forma vzoru, který je i výstupem přepisovacího pravidla.

Jsou tyto 3 typy přepisovacích pravidel:

- **Zjednodušující pravidla** - vytváří jednodušší logické stromy, než je jejich výstup. Většinou se používají během analýzy dotazu, před úplnou optimalizací.
- **Prozkoumávající pravidla** - také nazývány logické přepisovací pravidla. Generují logický ekvivalent z logického operátoru. Probíhá během optimalizace dotazu.
- **Implementační pravidla** - také nazývány fyzické přepisovací pravidla, generují fyzický ekvivalent logického operátoru. Také probíhá během optimalizace dotazu.

Optimalizátor dotazů tedy ke generování a zkoumání možných alternativních plánů používá přepisovací pravidla. Všechny logické i fyzické plány, generované optimalizátorem, jsou ukládány do **memo struktury**. K fyzickým alternativám je ještě připsána jejich odhadovaná cena. Více informací o memo struktuře se nachází v kapitole 2.5. Nicméně je důležité mít na mysli, i když tyto alternativní plány jsou ekvivalentní a vrací stejný výsledek, jejich fyzická implementace může mít velmi odlišnou cenu. Pro finální plán se vybere následně z memo struktury nejlepší (nejlevnější) fyzické alternativy.

2.4.1 Příklady prozkoumávajících pravidel

Příkladem prozkoumávajícího pravidla může být **pravidlo komutativity a asociativity**, které se používají při použití *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 2.4. Všimněme si také, že pokud uplatníme tohle pravidlo 2x, tak nám vygeneruje původní výraz. To znamená, pokud použijeme tuto transformaci, získáme *B join A*. Pokud bychom ale použili znovu stejnou transformaci, získáme znovu *A join B*. Optimalizátor nicméně umí vyřešit tento problém, aby se vyhnulo duplicitním výrazům.

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

Kód 2.4: 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 2.5.

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

Kód 2.5: pravidlo asociativity

Při použití těchto pravidel budeme mít sice ekvivalentní výrazy, ale při implementaci algoritmů fyzických operátorů 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.

2.4.2 Příklady implementačních pravidel

- **Logický operátor join** - neboli spojení dvou tabulek. Logický operátor *join* se dá přepsat na 3 různé fyzické operátory *join*. Každý z těchto fyzických *joinů* implementuje jiný algoritmus pro spojení dvou tabulek. Důležité je si uvědomit, že žádný fyzický *join* není obecně lepší než ostatní. Vždy záleží na konkrétní situaci, jak zde vysvětlím.

  - **Nested loop join** - používá jednoduchý algoritmus. Mějme tenhle 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.

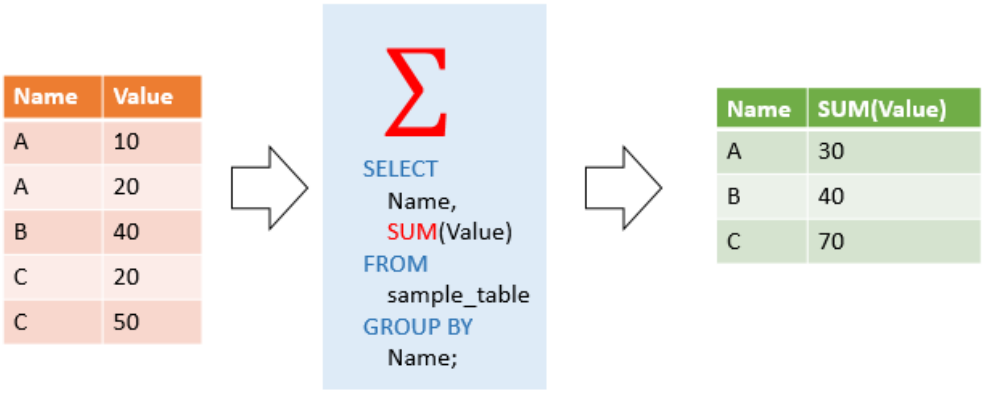
  - **Merge join** - vyžaduje aby obě tabulky byly seřazené podle slučovací podmínky *joinu*. Výhodou toho, že obě tabulky jsou seřazené je to, že se join vyřeší jedním průchodem oběma tabulkami.

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 seřazené. Hash join funguje tak, že se vytvoří dočasná **Hash tabulka** v paměti. Optimalizátor 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. Hash join také zablokuje, během spojení, úpravu tabulky, ze které je vytvořena hash tabulka [2]. Poté co je vytvořena a naplněna hash tabulka, druhá tabulka, nazvěme ji prozkoumávaná tabulka, bude porovnávaná s hash tabulkou jako u Nested loop joinu. S tím rozdílem, že hash tabulka bude mnohokrát menší, než originální tabulka. 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 tabulku se 2 sloupci. Používá se převážně při spojování velkých tabulek.

- **Logický operátor group aggregation** - tento operátor se používá k agregaci dat, neboli ke shlukování záznamů dle atributů a vypočítání nějaké agregované hodnoty. Výsledkem může být jediná hodnota, například průměrný plat společnosti, nebo to může být hodnota za skupinu. Jako je průměrný plat podle oddělení. SQL server má dva operátory implementující agregaci, které můžou být použity pro vyřešení dotazů s funkcionalitou (jako SUM, AVG nebo MAX), GROUP BY klauzuli nebo DISTINCT funkci. Příklad agregace můžeme vidět na obrázku 2.3.



Obrázek 2.3: Group aggregation
[4]

  - **Stream Aggregate** - Dotazy, které používají agregaci a vrací pouze jedinou hodnotu a nepoužívají GROUP BY klauzule, se nazývají **skalární agregace**. Příklad dotazu se skalární operací můžeme vidět v kódu 2.6. Skalární agregace je vždy implementována pomocí Stream Aggregate operátoru. Při použití Stream Aggregate pro agregaci s klauzulí GROUP BY, která rozdělí výsledek na skupiny, je nutné mít vstup seřazený podle sloupce, který považujeme za skupinu pro agregaci.

    V podstatě tento algoritmus vytvoří z prvního přečteného řádku skupinu a jeho agregovanou hodnotu. V následujících řádcích kontroluje, jestli skupina je stejná. Pokud ano, přičte se agregovaná hodnota řádku. Pokud ne, vytvoří se nová skupina.

```
SELECT count(*) as "count of all employees"
FROM Emplyee
```

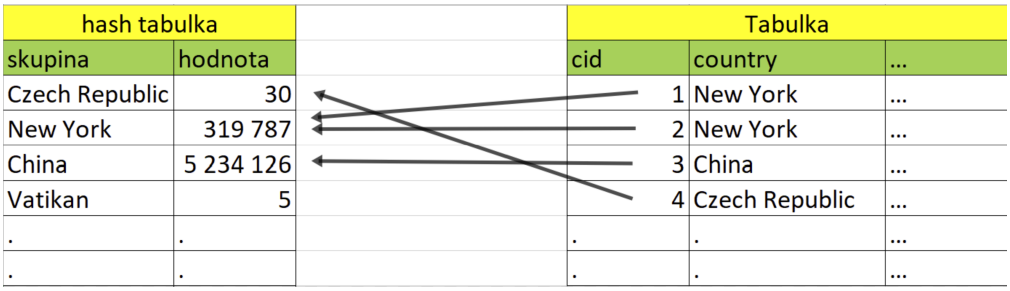
Kód 2.6: Dotaz se sklarální agregací

  - **Hash Aggregate** - funguje velmi podobně jako **Hash join** zmíněný výše. Optimalizátor může zvolit Hash Aggregation pro velké tabulky, kde data nejsou seřazené a jeho odhad mohutnosti odhadne jen na pár skupin. Hash aggregate je podobný jako Stream aggregate s tím rozdílem, že zde data nemusí být seřazená.

Vytvoří se dočasná Hash tabulka v paměti. Pro každý řádek z tabulky se zkontroluje skupina, jestli již existuje v Hash tabulce. Pokud neexistuje, zapíše se skupina s aggregovanou hodnotou do Hash tabulky. Pokud skupina v Hash tabulce existuje, vyhledá se řádek v Hash tabulce s tou skupinou a přičte se k Hash hodnotě agregovaná hodnota řádku. Pro každou skupinu v agregaci, na konci algoritmu, bude existovat jeden řádek v hash tabulce s výslednou hodnotou. Příklad hash tabulky můžeme vidět na obrázku 2.4.

```
SELECT country, count(id) as "count"
FROM Emplyee
GROUP BY country
```

Kód 2.7: Dotaz pro obrázek 2.6



Obrázek 2.4: Hash tabulka pro agregaci

2.4.3 Zobrazení přepisovacích pravidel

Jestliže nás zajímá, jaká přepisovací pravidla byla použita pro náš dotaz, můžeme pro to využít statistiky přepisovacích pravidel a to konkrétně *sys.dm_exec_query_transformation_stats*. Statistiky přepisovacích pravidel ukládají všechna přepisovací pravidla, která byla provedena v konkrétní databázi. K názvům jednotlivých přepisovacích pravidel jsou ukládané informace, jako je třeba kolikrát bylo pravidlo uplatněno. Viz. kód ve výpise 2.8.

Tyto statistiky jsou ale pro všechny dotazy, které byly kdy provedeny na danou databázi. Pro získání přepisovacích pravidel pro jeden dotaz musíme uložit stav statistik před provedením dotazu a po provedení dotazu. Následně rozdíl mezi oběma statistikami nám ukáže přepisovací pravidla pro daný dotaz. [5]

```sql
select *
FROM sys.dm_exec_query_transformation_stats;
```

Kód 2.8: dotaz pro obrázek 2.7

	name	promise_total	promise_avg	promised	built_substitute	succeeded
1	JNtoNL	1421457	136,639142555032	10403	2864	2808
2	LOJNtoNL	834818	449,310010764263	1858	1858	1829
3	LSJNtoNL	37518	446,642857142857	84	84	84
4	LASJNtoNL	26158	451	58	58	58
5	JNtoSM	2224280	417,235040330144	5331	5182	2456
6	FOJNtoSM	2724	454	6	6	6
7	LOJNtoSM	187968	426,231292517007	441	417	361
8	ROJNtoSM	185698	425,912844036697	436	412	357
9	LSJNtoSM	29544	410,333333333333	72	66	0
10	RSJNtoSM	29544	410,333333333333	72	66	57
11	LASJNtoSM	6810	454	15	15	3
12	RASJNtoSM	6810	454	15	15	15
13	IdxJNtoSM	3996	444	9	9	9
14	FOJnoneqToSM	1200	200	6	6	0
15	JNtoHS	1916002	182,45900390439	10501	3698	2258
16	FOJNtoHS	5472	456	12	12	12
17	LOJNtoHS	629110	338,595263724435	1858	845	769
18	ROJNtoHS	628654	338,532040926225	1857	844	769

Obrázek 2.5: Statistika přepisovacích pravidel

2.4.4 Použitá přepisovací pravidla v aplikaci

Výše jsem detailněji popsal prozkoumávací (2.4.1) a implementační (2.4.2) pravidla pro *join* a group aggregation, které jako jediné byly neoficiálně dokumentované [2]. Zde popíšu další přepisovací pravidla, o kterých už jsem nenašel žádné informace. Pro zjištění, k čemu slouží následující přepisovací pravidla, jsem zjišťoval v jakých typech dotazů se konkrétní přepisovací pravidlo použije, a jak se změní finální plán, když konkrétní přepisovací pravidlo vypnu. Každé přepisovací pravidlo jsem testoval na desítkách dotazů.

- **EnforceSort** - je jedno ze základních přepisovacích pravidel. Toto pravidlo nám převede logický operátor *get* na fyzický operátor *EnforceSort*, který nám třídí vstup podle daného kritéria.
- **SelectToFilter** - je další ze základních přepisovacích pravidel. Pokud máme na tabulku nějaký filtr, použije se tohle přepisovací pravidlo, které nám převede logický operátor *select* na fyzický operátor *filter*. Ten následně vybere řádky vstupu, které splňují podmínku filtru.
- **GetToScan** - je podobný jako *EnforceSort*. Pokud výstup nepotřebujeme mít setřízený, použije se na ní tohle pravidlo, které nám převede logický operátor get na fyzický operátor *Range* nebo *TableScan*.

- **ImplRestrRemap** - neboli **Implement Restrict Remap** se používá, pokud chceme již vytvořený výstup z *joinu*, nebo group aggregation dál omezit. Třeba odstraněním duplicit. Toto pravidlo se uplatňuje buď na logický operátor *join* nebo na logický operátor group aggregation, který převede na fyzický operátor RestrRemap.
- **EnforceBatch** - se používá při vynucení použití Batch Mode. Co je to Batch mode je popsáno na této webové stránce [6]. EnforceBatch se implementuje na logický operátor *join*.
- **ProjectToComputeScalar** - pro výpočet nové hodnoty z existující hodnoty v tabulce se využívá skalární výpočet. Při využití skalárního výpočtu se použije i tohle pravidlo, které nám převede logický operátor *project* na fyzický operátor *ComputeScalar*.

2.5 Memo struktura

Memo struktura je struktura jednotlivých operátorů, které jsou generovány a analizovány optimalizátorem. Tyto operátory můžou být logický, nebo fyzický operátor.

Pro každou optimalizaci SQL dotazů je vytvořena nová memo struktura. Ta se následně rozděluje na **skupiny**. Skupina se pak skládá z ekvivalentních operátorů. **Všechny operátory z jedné skupiny musí vracet stejný výstup.**

Optimalizátor první zkopíruje originální logický strom do memo struktury tak, že každý logický operátor je vložen do své vlastní skupiny. Poté začne optimalizační proces, během kterého jsou uplatněná přepisovací pravidla generující ekvivalentní operátory. Jednotlivé operátory jsou vloženy do příslušné skupiny v memo struktuře. Přepisovací pravidla mohou také vytvořit nový ekvivalentní operátor, který nepřísluší do žádné existující skupiny. V takové situaci se vytvoří nová skupina, do které se vloží operátor.

Jak už jsem zmínil, každý operátor ve skupině je logický, nebo fyzický, jako je třeba *join* a výsledný plán je tvořen kombinací těchto operátorů.

I když existuje možnost, že mohou různé kombinace přepisovacích pravidel vytvořit stejný výraz, memo struktura je tvořena tak, aby se tomu vyhnula. Jak duplikátům, tak i redundantním optimalizacím. Tímto způsobem šetří paměť a je efektivnější.

2.5.1 Příklad memo struktury

Mějme následující dotaz 2.9.

```
SELECT d.name, count(e.id) as "count"
FROM Employee as e
JOIN Department as d on e.dId=d.Id
GROUP BY d.name
```

Kód 2.9: Dotaz pro pro příklad zobrazení memo struktury

První co se zapíše do memo struktury je logický strom. Ten může vypadat jako na obrázku 2.6. Na logický strom může optimalizátor ještě provézt **zjednodušené pravidlo**, které bylo popsáno i s ostatními pravidly v kapitole 2.4.

Skupina 4	aggregation 3
Skupina 3	join 1 & 2
Skupina 2	Scan Department
Skupina 1	Scan Employee

Obrázek 2.6: Memo struktura 1

Následně optimalizátor použije **prozkoumávající pravidla**. Jelikož máme v dotazu *join*, tak podle kapitoly 2.4.1 můžeme provézt pravidlo komutativity. V memo struktuře to následně bude vypadat jako na obrázku 2.7.

Skupina 4	aggregation 3	
Skupina 3	join 1 & 2	join 2 & 1
Skupina 2	Scan Department	
Skupina 1	Scan Employee	

Obrázek 2.7: Memo struktura 2

Po prozkoumávajících pravidlech se použijí **implementační pravidla**. V tomhle příkladě to je skenování jednotlivých tabulek, vytvoření fyzického operátoru pro *join* a agregaci. Viz obrázek 2.8.

Skupina 4	aggregation 3		Hash aggregate 3
Skupina 3	join 1 & 2	join 2 & 1	Nested loop join 1 & 2
Skupina 2	Scan Department		Clustered index Scan
Skupina 1	Scan Employee		Clustered index Scan

Obrázek 2.8: Memo struktura 3

Do memo struktury se zapsal jeden fyzický operátor pro join a agregaci. Ale existují i jiné fyzické operátory a to například Merge join a Stream aggregate, což je popsané v kapitole 2.4.3. Tyhle operátory vyžadují, aby tabulka byla seřazená, což momentálně není. Proto na obě tabulky musí být uplatněno implementační pravidlo pro seřazení tabulky. V memo struktuře to následně bude vypadat jako na obrázku 2.9.

Skupina 4	aggregation 3		Hash aggregate 3	Stream Aggregate 3
Skupina 3	join 1 & 2	join 2 & 1	Nested loop join 1 & 2	Merge sort 1 & 2
Skupina 2	Scan Department		Clustered index Scan	sort
Skupina 1	Scan Employee		Clustered index Scan	sort

Obrázek 2.9: Memo struktura 4

Můžeme si všimnout, že pro v memo struktuře není 3. způsob *joinu* Hash join. To je z důvodu, jak už jsem zmínil v úvodu, že optimalizátor negeneruje nutně vždy všechny možné plány a tím pádem memo struktura nemusí vždy obsahovat všechny možné jak už prozkoumávající, tak i implementační pravidla. Tohle je zjednodušený pohled na memo struktury. Přesná syntaxe memo struktury, kterou vrací SQL server, můžeme vidět v kapitole 2.6.4.1.

2.6 Hints

Hints jsou doplňující informace k prováděnému dotazu. Slouží například k tomu, když chceme ovlivnit, jak optimalizátor bude zpracovávat dotazy, nebo chceme jen zobrazit debug informace o dotazu. Pro použití hints se definují příkazem **OPTION**, který se napíše přímo za kód dotazu [7]. Je celá řada hints, zde vysvětlím jen ty, které jsem použil k vypracování této bakalářské práce.

2.6.1 RULEOFF

Pokud nám nevyhovuje, jak optimalizátor optimalizuje dotazy, můžeme vypnout některá přepisovací pravidla a tím donutit optimalizátor, aby dotaz zpracoval jiným způsobem. Vypnutí přepisovacího pravidla se definuje příkazem *RULEOFF(<název přepisovacího pravidla>)*. Výsledný dotaz může pak vypadat následovně.

```
SELECT * FROM A OPTION (RULEOFF('JNtoSM'))
```

Kód 2.10: Vypnutí přepisovacího pravidla

Kde v tomto případě zakáže použití přepisovacího pravidla, které převede logický *join* do fyzického Merge join, pro daný dotaz.

Vypnutí přepisovacích pravidel jde i globálně pro celou databázi, a to za použití tohoto příkazu *DBCC RULEOFF(<název přepisovacího pravidla>)* [8].

Toto se ale silně nedoporučuje používat na produkční databázi. Může to vést k tomu, že databáze bude špatně a pomalu pracovat.

Pro zobrazení všech přepisovacích pravidel, které můžeme vypnout, spustíme příkaz uvedený v kódu 2.11.

```
DBCC TRACEON (3604)
DBCC SHOWONRULES
```

Kód 2.11: Zobrazení přepisovacích pravidel

2.6.2 QUERYTRACEON

Pro zobrazení podrobností ohledně zpracování dotazu se používá *QUERYTRACEON <číslo>* [9]. QUERYTRACEON dokáže zobrazit různé informace ohledně dotazu. Každé číslo zadané za QUERYTRACEON zobrazí jiné informace o dotazu. V této práci jsem ale používal:

- **QUERYTRACEON 3604** - kopíruje debug informace z debugu do hlavního okna se zprávami v textovém formátu.
- **QUERYTRACEON 8607** - zobrazuje finální strom operátorů v debugu, které byly použity ve finálním plánu dotazu.
- **QUERYTRACEON 8615** - zobrazuje memo strukturu finálního plánu v debugu. Můžeme zde vidět postup z logického stromu na finální strom.

Textový výstup z QUERYTRACEON 8607 a QUERYTRACEON 8615 jsou vstupem do mé aplikaci, která je popsána v kapitole 3.

2.6.3 MAXDOP

Jak už jsem výše zmínil, optimalizátor pro vypočítání optimálního plánu bere v potaz i fyzický výkon stroje, na kterém se nachází databáze. Proto se finální plán může lišit na počítači s 1 CPU jádrem a na počítači s 8 CPU jádry. Přikaz *MAXDOP <počet jader>* nastavuje kolik jader se použije pro výpočet daného dotazu.

2.6.4 Příklady zobrazení

V mé aplikaci kombinuji textový výstup z memo struktury a z finálního stromu pro ucelenější memo strukturu. Výstup pro memo struktury v SQL serveru nevrací celou strukturu. Vrací pouze část memo struktury, která se podílela na sestavení finálního plánu vykonání.

2.6.4.1 Memo struktura

Pro příklad bude stačit jednoduchý dotaz s jedním *joinem* (2.12). Výslednou memo strukturu můžeme vidět v kódu 2.13.

```
SELECT *
FROM A
JOIN B ON A.fkb=B.id
OPTION (QUERYTRACEON 3604,
     QUERYTRACEON 8615,
     MAXDOP 1)
```

Kód 2.12: Zobrazení memo struktury - kód

```
Root Group 5: Card=1.00001e+06 (Max=1.10001e+06, Min=0)
   4 PhyOp_HashJoinx_jtInner 4.1 3.4 2.0 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total
      0)= 119.201 (Distance = 2)
   1 LogOp_Join 4 3 2 (Distance = 1)
   0 LogOp_Join 3 4 2 (Distance = 0)
Group 4: Card=10004 (Max=11004.4, Min=0)
   1 PhyOp_Range 1 ASC Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 1.07429 (
      Distance = 1)
   0 LogOp_Get (Distance = 0)
Group 3: Card=1.00001e+06 (Max=1.10001e+06, Min=0)
   4 PhyOp_Range 1 ASC Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total0)= 106.927 (
      Distance = 1)
   2 PhyOp_Sort 3.4 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)=938.179 (Distance =
       0)
   0 LogOp_Get (Distance = 0)
Group 2:
   0 ScaOp_Comp 0.0 1.0 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 3 (Distance =
      0)
Group 1:
   0 ScaOp_Identifier Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 1 (Distance = 0)

Group 0:
   0 ScaOp_Identifier Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)= 1 (Distance = 0)
```

Kód 2.13: Zobrazení memo struktury - zkrácený výstup

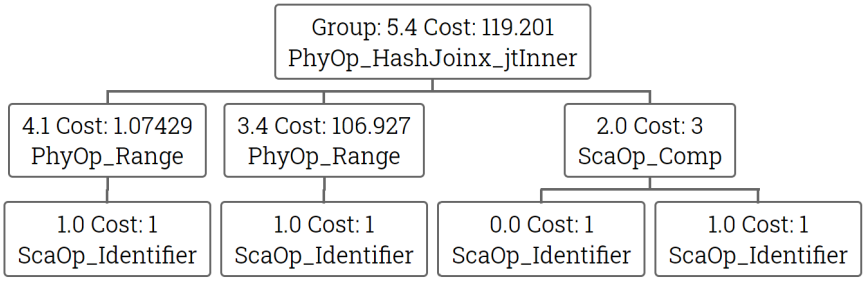
Jak jsem vysvětloval, memo struktury v kapitole 2.5, v kódu 2.13 můžeme vidět, že je rozdělena na jednotlivé skupiny, které jsou očíslované. V jednotlivých skupinách pak jsou ekvivalentní operátory. Tento výstup není oficiálně dokumentován Microsoftem.

Pro detailnější popsání si vezmeme následující operátor ze skupiny 5.

```
4 PhyOp_HashJoinx_jtInner 4.1 3.4 2.0 Cost(RowGoal 0,ReW 0,ReB 0,Dist 0,Total 0)=
    119.201 (Distance = 2)
```

Kód 2.14: Operátor 5.4 z memo struktury

Tento operátor má označení ve skupině 5 číslem 4. Tím můžeme říct, že jeho označení v memo struktuře je 5.4. Za číslovkou 4 se nachází jeho název *PhyOp HashJoinx jtInner*. Název také vyjadřuje, jestli jde o fyzický, nebo logický operátor. Za názvem se nachází čísla odkazujících operátorů, které tento operátor vyžaduje pro provedení. To znamená, pokud tam je číslo 4.1, tak vyžaduje operátor, který je ve skupině 4 s označením 1. Následně pokud operátor je fyzický, je zde uvedena jeho cena. Když si tedy vezmeme operátor číslo 5.4 a spojíme ho s požadovanými operátory (4.1,...), vyjde nám tenhle diagram, jako na obrázku 2.10.



Obrázek 2.10: Diagram plánu z memo struktury

Vidíme, že Hash join se skládá ze 2 tabulek, které jsou procházeny operátorem *PhyOp Range*, a z podmínky joinu. Tohle nám dalo už celý jeden plán. Abychom věděli, z jaké skupiny máme začít, aby nám to dalo celý diagram, tak proto je před jednou skupinou napsané slovo **root**. To označuje, že to je výchozí skupina pro všechny plány. Pomocí tohoto procházení dokážeme zobrazit i všechny alternativní plány, ale i ekvivalentní operátory pro jakýkoliv operátor.

Jak ale můžeme na obrázku 2.10 vidět, nejsou zde vidět názvy jednotlivých tabulek, nebo jednotlivé podmínky v diagramu. Tento výstup v SQL serveru tyto informace nemá. Ale tyto informace se vyskytují ve finálním stromu.

2.6.4.2 Finální strom

Pro stejný dotaz, jako u memo struktury, zobrazíme finální strom následovně, jak můžeme vidět v kódu 2.16.

```
SELECT *
FROM A
JOIN B ON A.fkb=B.id
OPTION (
QUERYTRACEON 3604,
QUERYTRACEON 8607,
MAXDOP 1
)
```

Kód 2.15: Zobrazení finálního stromu - kód

```
*** Output Tree: ***
PhyOp_HashJoinx_jtInner (batch)(QCOL: [benchmark].[dbo].[B].id) = (QCOL: [
    benchmark].[dbo].[A].fkb)
   PhyOp_Range TBL: B(1) ASC Bmk ( QCOL: [benchmark].[dbo].[B].id) IsRow: COL:
       IsBaseRow1002
   PhyOp_Range TBL: A(1) ASC Bmk ( QCOL: [benchmark].[dbo].[A].id) IsRow: COL:
       IsBaseRow1000
   ScaOp_Comp x_cmpEq
      ScaOp_Identifier QCOL: [benchmark].[dbo].[B].id
      ScaOp_Identifier QCOL: [benchmark].[dbo].[A].fkb
********************
** Query marked as Cachable
********************
** Query cachability updated to FALSE
********************
```

Kód 2.16: Zobrazení finálního stromu - výstup

Tento výstup nám vypisuje v textové formě finální strom, což je plán, který optimalizátor vybral pro výpočet dotazu. Můžeme zde vidět, že finální strom obsahuje operátory se stejným názvem. Ale k tomu ukazuje navíc informace, jako je třeba název tabulky a sloupců, se kterými operátor pracuje. Například u prvního řádku vidíme, že fyzický operátor Hash join má spojovací podmínky $B.id = A.fkb$ na tabulky nacházející se v databázi *benchmark*.

Kapitola 3

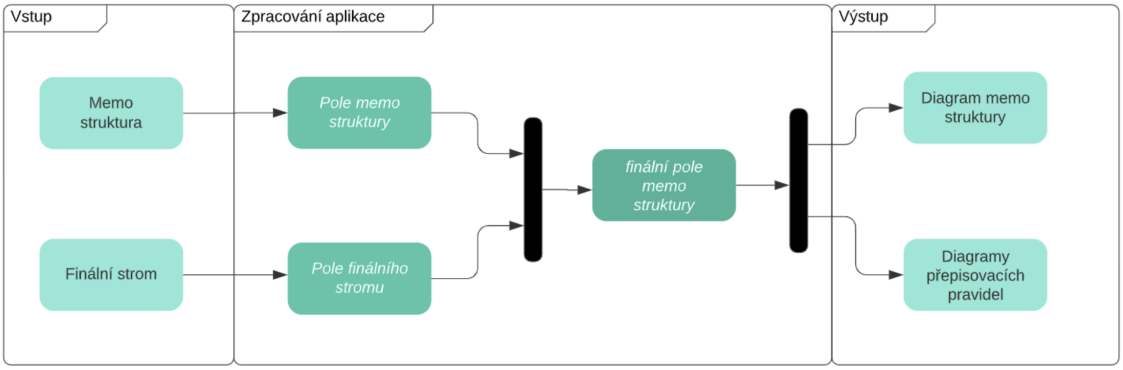
Vlastní implementace nástroje

3.1 Popis aplikace

Jedná se o webovou aplikaci umožňující vizuální zobrazení memo struktury a přepisovacích pravidel dotazu SQL serveru. Spojením informací z memo struktury a finálního stromu vytvoří přehledný diagram plánů optimalizace. Pro některé jednotlivé části diagramu, které reprezentují fyzické a logické operátory, se dají zobrazit alternativní operátory. V diagramech se zobrazují i jednotlivá přepisovací pravidla, kteréá byly použity pro daný dotaz. Vizuální výsledek se dá uložit jako offline html soubor. Aplikace běží na PHP verzi 7.4.13 a pro veškerou vizuální interakci na stránce jsem použil nadstavbu javascriptu *jquery* verze 3.3.1.

3.1.1 Schéma aplikace

Jak můžeme vidět na schématu na obrázku 3.1, aplikace po zadání vstupních údajů zpracuje vstup memo struktury a finálního stromu do polí. Následně obě pole spojí do jednoho. Z výsledného pole vytvoří diagram memo struktury a vytvoří diagramy pro přepisovací pravidla, které zde byly uplatněny.



Obrázek 3.1: Schéma aplikace

3.2 Zobrazení diagramu

3.2.1 Převedení vstupu do pole

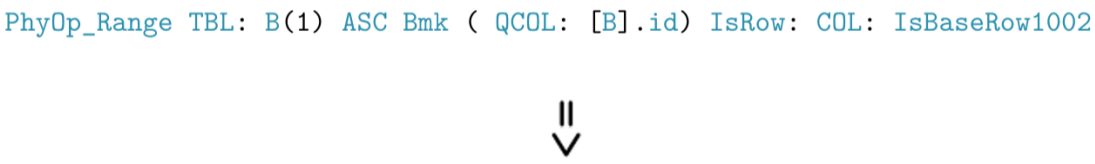
Pro zobrazení diagramu aplikace požaduje vložení memo struktury a finálního stromu v textové podobě. Tento vstup můžeme získat pomocí kódu 2.12 a 2.15, o kterém se píše v kapitole 2.6.4.

Následně aplikace převede textovou memo strukturu do přehledného objektového pole se stromovou strukturou, které budeme nazývat **memostrom**. Memostrom bude mít strukturu jako v kódu 3.1. Tento proces se vykonává v souboru *get_memo_array.php*.

```
array(
   ["číslo skupiny"] => array(
      [detail] => "dotatečné informace o skupině"
      // pole jednotlivých operátorů ve skupině
      [subgroup] => array(
         []["číslo řádku ve skupině"] => array(
            [name] => "název operátoru"
            [cost] => "cena operátoru"
            // pole vyžadovaných operátorů
            [groups] => array(
               "číslo skupiny a podskupiny operátoru oddělené tečkou"
            )
         )
      )
   )
)
```

Kód 3.1: Struktura memostromu v aplikaci

U textový výstupu finálního stromu nám stačí jen upravit popisy jednotlivých operátorů na přehlednější formu, jak můžeme vidět na obrázku 3.2 a výsledek vložit do klasického pole, které můžeme vidět v kódu 3.2 a budeme ho nazývat **finalstrom**.



Obrázek 3.2: Úprava operátoru ve finálním stromě

```
Array(
   [0] =>
   PhyOp_HashJoinx_jtInner <br> (batch)(QCOL: [benchmark].[dbo].[B].id) = (QCOL: [
       benchmark].[dbo].[A].fkb)
   [1] =>
      PhyOp_Range <br> ( QCOL: [benchmark].[dbo].[B].id)
   [2] =>
      PhyOp_Range <br> ( QCOL: [benchmark].[dbo].[A].id)
   [3] =>
      ScaOp_Comp <br> x_cmpEq
   [4] =>
         ScaOp_Identifier <br> QCOL: [benchmark].[dbo].[B].id
   [5] =>
         ScaOp_Identifier <br> QCOL: [benchmark].[dbo].[A].fkb
)
```

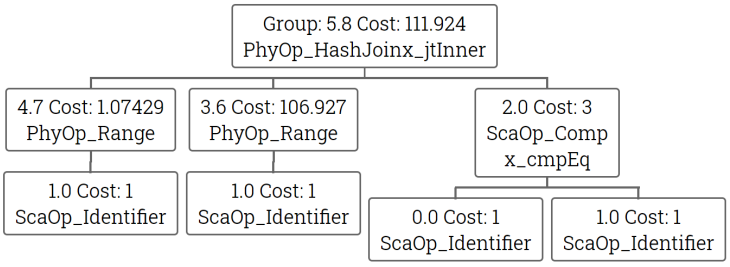
Kód 3.2: Struktura finalstromu v aplikaci

Jak si v kódu 3.2 můžeme všimnout, ve finalstromu jsou zachované originální tabulátory z textové finálního stromu. To je důležité pro rozpoznání, na jaké úrovni v diagramu je daný operátor. Tento převod se vykonává v souboru *get_tree_array.php*.

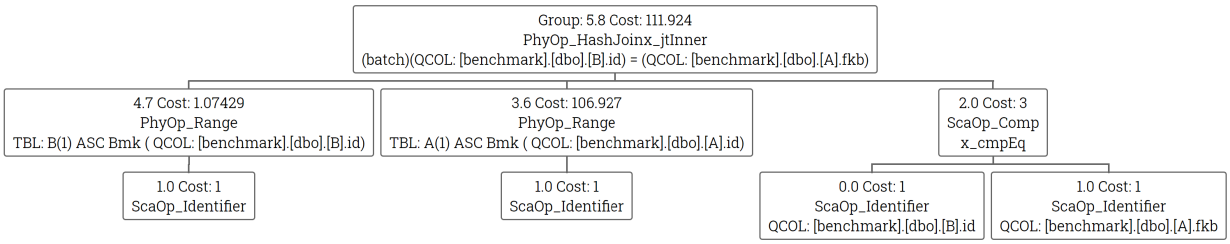
3.2.2 Spojení memo struktury s finálním stromem

Získali jsme memostrom a finalstrom. V memo struktuře máme všechny operátory, které nějakým způsobem napomohly k vytvoření finálního plánu. Ale již se tam nenachází podrobnější detaily o nich. Jako je například s jakou tabulkou nebo sloupcem pracuje daný operátor. Tyto informace získáme z finálního stromu. Proto je zapotřebí teď vytvořený memostrom a finalstrom spojit.

Podrobný popis operátorů ve finálním stromě se ale vztahuje jen na operátory z finálního plánu. To znamená, že ne pro všechny operátory z memo struktury jsme schopni určit podrobný popis.



Obrázek 3.3: Diagram plánu bez podrobnějších informací

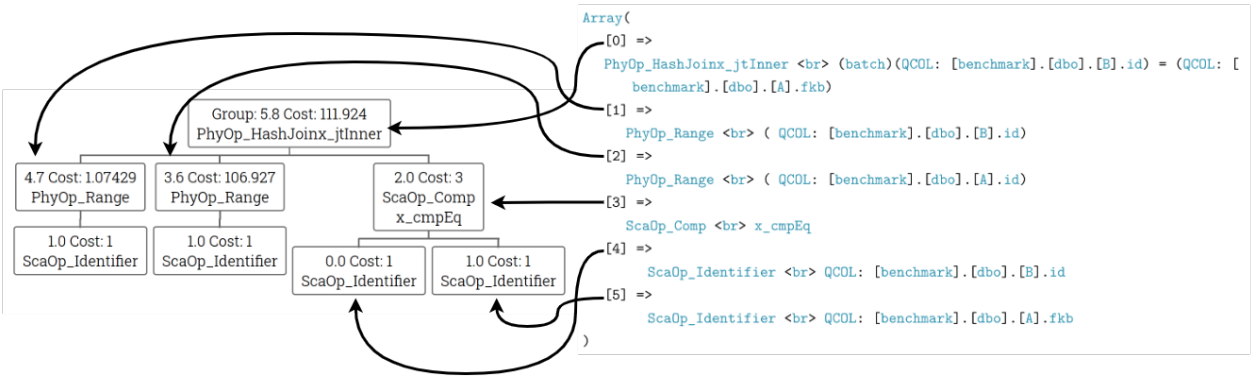


Obrázek 3.4: Diagram plánu obrázku 3.3 doplněný o informace z finálního stromu

Jak můžeme vidět v kódu 3.2, jednotlivé operátory neobsahují označení, které by se daly použít pro spojení s memo strukturou, jako je třeba skupina. Proto se na spojení obou polí musí najít jiný způsob. Cíl je z memostromu najít finální plán a v něm změnit popis jednotlivých operátorů na popis z finalstromu.

Jak víme, finální plán, který optimalizátor vybere, má nejmenší cenu. V poli memo struktury máme uvedenou cenu jednotlivých operátorů. Proto stačí najít v root skupině, což je hlavní skupina memo struktury, operátor s nejnižší cenou. Z tohoto operátoru se skládá finální plán.

Při takovém výběru z memostromu budeme mít plán, který zbývá naplnit informacemi z finalstromu.



Obrázek 3.5: Doplnění plánu o finální strom

Na obrázku 3.5 můžeme vidět proměnu, kterou chceme udělat. Jak jsem ale zjistil, jaký operátor z finalstromu patří k operátoru z memostromu? Na obrázku 3.5 vidíme, že diagram memo struktury obsahuje dva fyzické operátory *range*, které se liší jen v ceně. A právě pomocí ceny a velikosti jednotlivých tabulek použitých v dotazu, se mi povedlo zjistit k jakému operátoru z finálního stromu patří.

Všimněme si na obrázku 3.5, že ne každý operátor z finálního plánu má nadefinovaný operátor z finalstromu. Můžeme to vidět na operátorech *1.0 ScaOp_Identifier*. Dále si můžeme všimnout pořadí finalstromu vzhledem k diagramu. Když nebudeme brát v potaz nepřiřazené operátory, operátory z finálního stromu jsou v pořadí, v jakém uzly navštěvujeme při **preorder průchodu stromu** [10].

Pomocí těchto poznatků se mi následně povedlo spojit memostrom s finalstromem. Výsledný plán pak vypadá jako na obrázku 3.4.

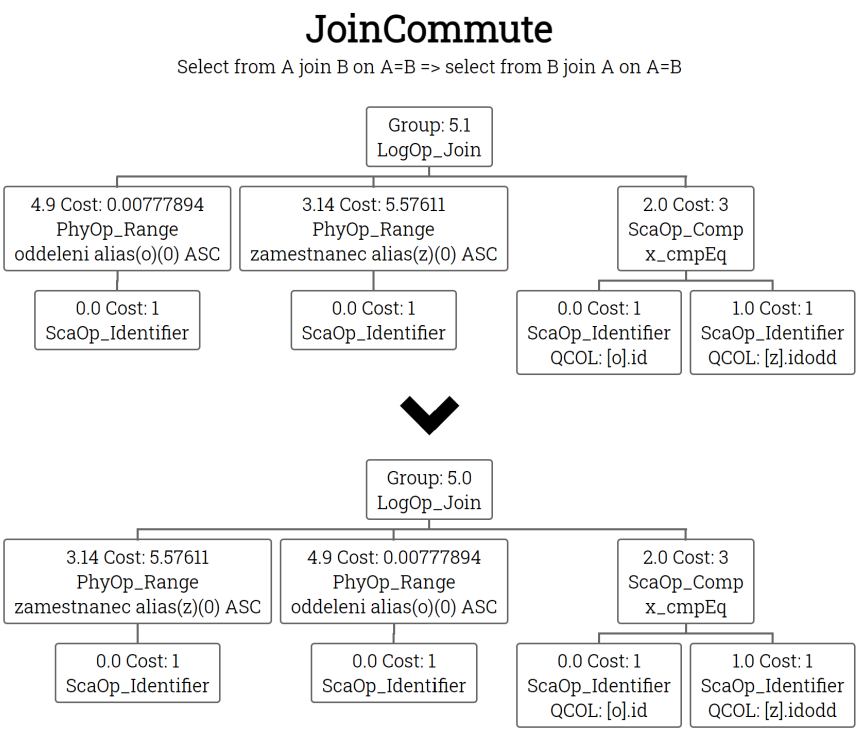
3.3 Zobrazení přepisovacích pravidel

Přepisovací pravidla, která jsou popsaná v kapitole 2.4, nám dopomáhají sestrojit finální plán. Má aplikace umožňuje grafické zobrazení některých přepisovacích pravidel, která vedla k sestrojení finálního plánu.

Pro zobrazení přepisovacích pravidel, která byla použita pro daný dotaz, můžeme zjistit pomocí statistik, které jsou popsané v kapitole 2.4.3. Takhle můžeme získat jen názvy přepisovacích pravidel, která byla použita pro daný dotaz, nicméně stále nedokážeme přesně ukázat plán před použitím přepisovacího pravidla a po použití. Pro tohle zobrazení jsem nadefinoval algoritmus, který dokáže, podle názvu operátorů v jedné skupině v memo struktuře, určit použitá přepisovací pravidla, která vedla k vytvoření nového operátoru ve skupině.

Každá skupina v memo struktuře obsahuje ekvivalentní operátory. Na začátku optimalizace každá skupina obsahovala jen jeden logický operátor. Další operátory, které se tam následně vepíši, vzniknou pomocí přepisovacích pravidel. Pomocí názvů operátorů a mým nastudovaným vědomostem ohledně přepisovacích pravidel [2], jsem nadefinoval podmínky pro sadu přepisovacích pravidel.

Každé přepisovací pravidlo se skládá ze dvou operátorů, které musí mít specifický název. Můj algoritmus se zakládá na tom, pokud ve skupině jsou oba operátory, které jsou nadefinované v přepisovacím pravidle, pak bylo použito dané přepisovací pravidlo. Například pokud jedna skupina obsahuje 2 logické operátory join, můžeme určit, že bylo použito přepisovací pravidlo join commute, které vidíme na obrázku 3.7.



Obrázek 3.6: Přepisovací pravidlo join commute

Pomocí tohoto algoritmu jsem nadefinoval sadu přepisovacích pravidel, u kterých jsem měl jistotu, co provádějí. Existuje řada dalších přepisovacích pravidel, která ale nejsou nikde dokumentovaná a z názvu není zřejmé, co konkrétně provádí. Aplikace je ale nastavena tak, že definice jednotlivých přepisovacích pravidel jsou v *json* souboru, který může být následně rozšířen o nová přepisovací pravidla.

Musíme mít na paměti to, že jsem hledal co nejobecnější způsob na určování přepisovacích pravidel. Tím pádem je možnost, že u některých dotazů, které jsem netestoval, může aplikace nějaké přepisovací pravidlo ukazovat špatně. Při svém testování na desítkách dotazů jsem na takovou situaci nenarazil a ani úvahou se mi nepodařilo přijít na situaci, kdy by toto pravidlo pro správné určení nestačilo. Jelikož SQL server nikde veřejně nedefinuje, jak pracují jednotlivá přepisovací pravidla, definice přepisovacích pravidel nacházející v této aplikaci jsou čistě založené na mých znalostech, které jsou popsány v kapitole 2.4.

V aplikaci jsem použil tyto přepisovací pravidla, které jsou v tabulce 3.1.

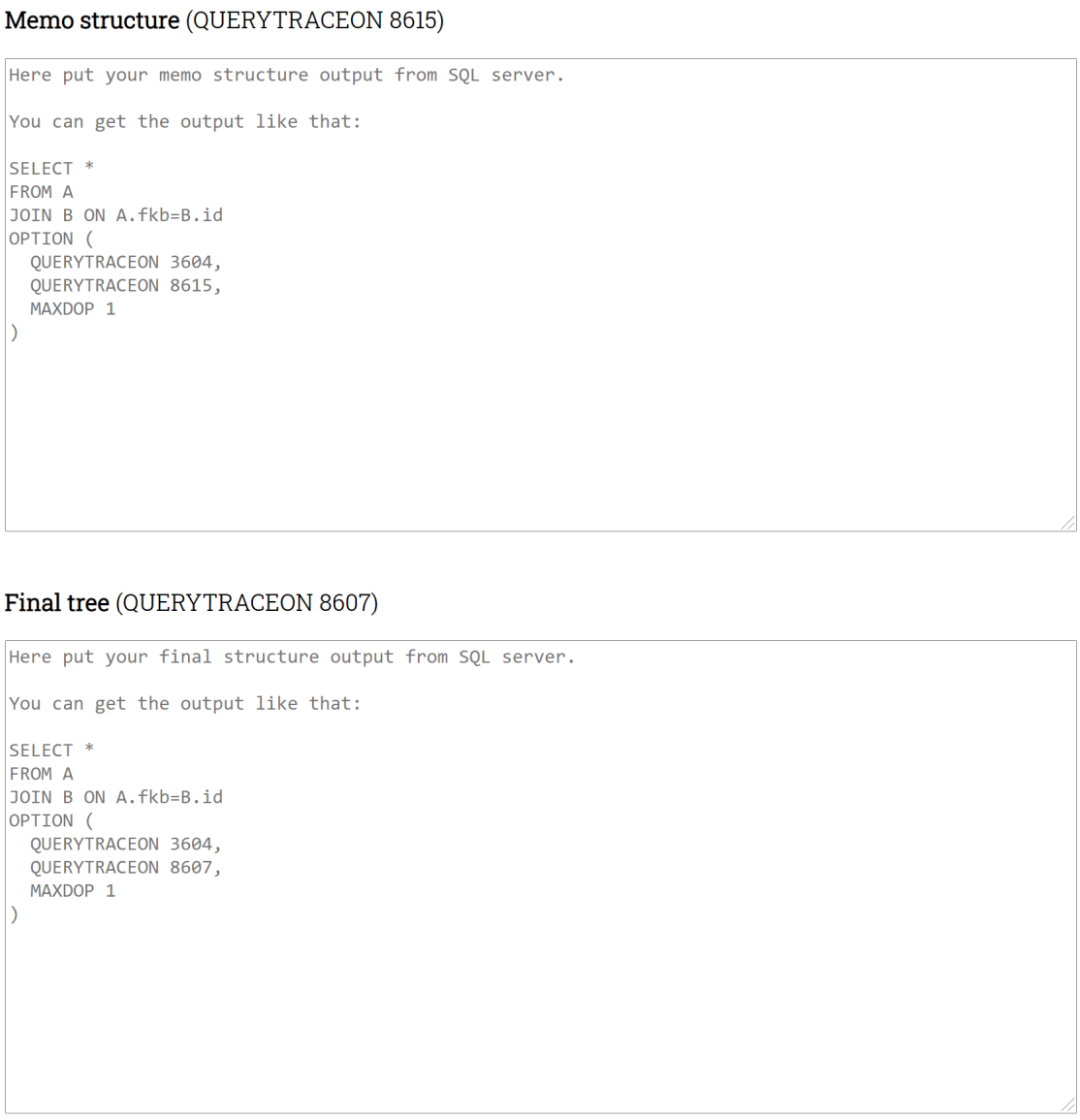
Tabulka 3.1: Přepisovací pravidla

název	původní operátor	výsledný operátor	popis
		Join commute	
JoinCommute	LogOp_Join	LogOp_Join	join commute
CommLOJN	LogOp_LeftOuterJoin	LogOp_RightOuterJoin	leftOuterJoin commute
CommROJN	LogOp_RightOuterJoin	LogOp_LeftOuterJoin	rightOuterJoin commute
CommLSJN	LogOp_LeftSemiJoin	LogOp_RightSemiJoin	leftSemiJoin commute
CommRSJN	LogOp_RightSemiJoin	LogOp_LeftSemiJoin	rightSemiJoin commute
		log join to phy join	
JNtoHS	LogOp_Join	PhyOp_HashJoin	join to hash join
JNtoSM	LogOp_Join	PhyOp_MergeJoin	join to sort merge join
JNtoNL	LogOp_Join	PhyOp_NestedJoin	join to nested loop join
LOJNtoHS	LogOp_LeftOuterJoin	PhyOp_HashJoin_jtLeftOuter	leftOuterJoin to hash join
LSJNtoHS	LogOp_RightOuterJoin	PhyOp_HashJoin_jtLeftSemi	leftSemiJoin to hash join
ROJNtoHS	LogOp_LeftSemiJoin	PhyOp_HashJoin_jtLeftOuter	rightOuterJoin to hash join
RSJNtoHS	LogOp_RightSemiJoin	PhyOp_HashJoin_jtRightSemi	rightSemiJoin to hash join
LOJNtoSM	LogOp_LeftOuterJoin	PhyOp_MergeJoin_jtLeftOuter	leftOuterJoin to sort merge
LSJNtoSM	LogOp_RightOuterJoin	PhyOp_MergeJoin_jtLeftSemi	leftSemiJoin to sort merge
ROJNtoSM	LogOp_LeftSemiJoin	PhyOp_MergeJoin_jtRightOuter	rightOuterJoin to sort merge
RSJNtoSM	LogOp_RightSemiJoin	PhyOp_MergeJoin_jtRightSemi	rightSemiJoin to sort merge
LOJNtoNL	LogOp_LeftOuterJoin	PhyOp_LoopsJoin_jtLeftOuter	leftOuterJoin to nested loop
LSJNtoNL	LogOp_RightOuterJoin	PhyOp_LoopsJoin_jtLeftSemi	leftSemiJoin to nested loop
ROJNtoNL	LogOp_LeftSemiJoin	PhyOp_LoopsJoin_jtRightOuter	rightOuterJoin to nested loop
RSJNtoNL	LogOp_RightSemiJoin	PhyOp_LoopsJoin_jtRightSemi	rightSemiJoin to nested loop
		ostatní s join	
ImplRestrRemap	LogOp_Join	PhyOp_RestrRemap	implement restrict remap
ImplRestrRemap	LogOp_LeftOuterJoin	PhyOp_RestrRemap	implement restrict remap
ImplRestrRemap	LogOp_RightOuterJoin	PhyOp_RestrRemap	implement restrict remap
ImplRestrRemap	LogOp_LeftSemiJoin	PhyOp_RestrRemap	implement restrict remap
ImplRestrRemap	LogOp_RightSemiJoin	PhyOp_RestrRemap	implement restrict remap
EnforceBatch	LogOp_Join	PhyOp_ExecutionModeAdapter	Vynucení Batch mode
JoinAssociate	LogOp_Join	LogOp_Join	Join Associatite
		log aggregation to phy aggregation	
GbAggToStrm	LogOp_GbAgg	PhyOp_StreamGbAgg	stream aggregate
GbAggToHS	LogOp_GbAgg	PhyOp_HashGbAgg	hash aggregate
GbAggToSort	LogOp_GbAgg	PhyOp_Sort	sort aggregation
		ostatní s aggregation	
ImplRestrRemap	LogOp_GbAgg	PhyOp_RestrRemap	implement restrict remap
		ostatní	
EnforceSort	LogOp_Select	PhyOp_Sort	třízení tabulky
GetToScan	LogOp_Get	PhyOp_TableScan	skenování tabulky
GetToScan	LogOp_Get	PhyOp_Range	skenování tabulky
SelectToFilter	LogOp_Select	PhyOp_Filter	filtrování tabulky
ProjectToComputeScalar	LogOp_Project	PhyOp_RestrRemap	skalární operace
GetIdxToRng	LogOp_GetIdx	PhyOp_Range	skenování tabulky

3.4 Funkce

3.4.1 Úvodní stránka

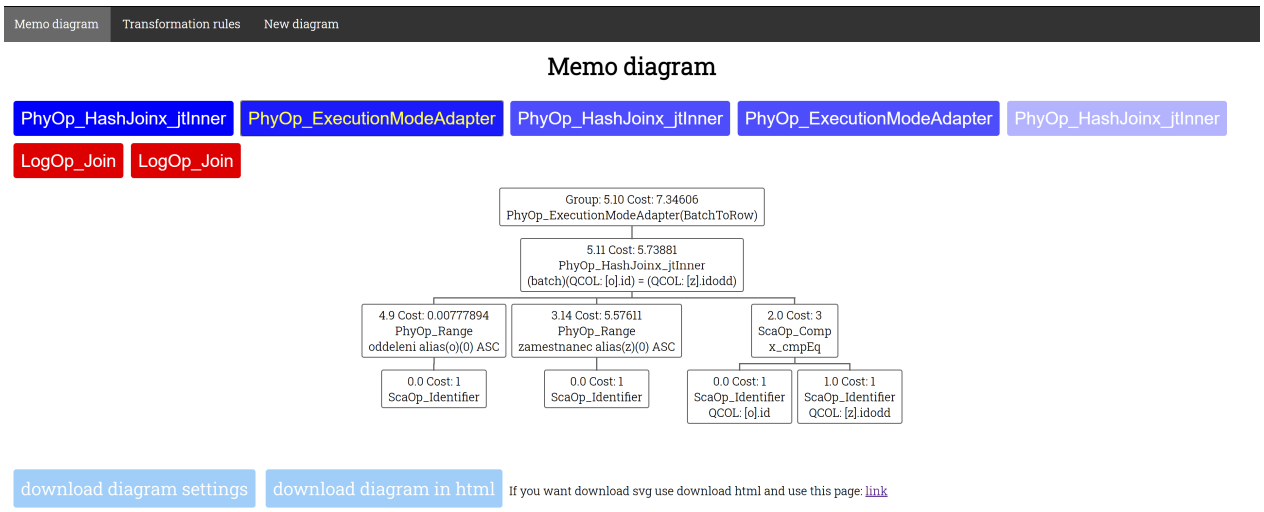
První, co potřebujeme udělat pro práci s aplikací, je získat textový výstup memo struktury a finálního stromu. Příklad kódu pro zobrazení memo struktury (2.12) a pro finální strom (2.15) je v kapitole 2.6.4. Textový výstup vložíme do příslušného textového pole.



Obrázek 3.7: Vložení textového výstupu z databáze do aplikace

3.4.2 Zobrazení memo struktury

Následně se nám zobrazí diagram finálního plánu podobně jako na obrázku 3.8.

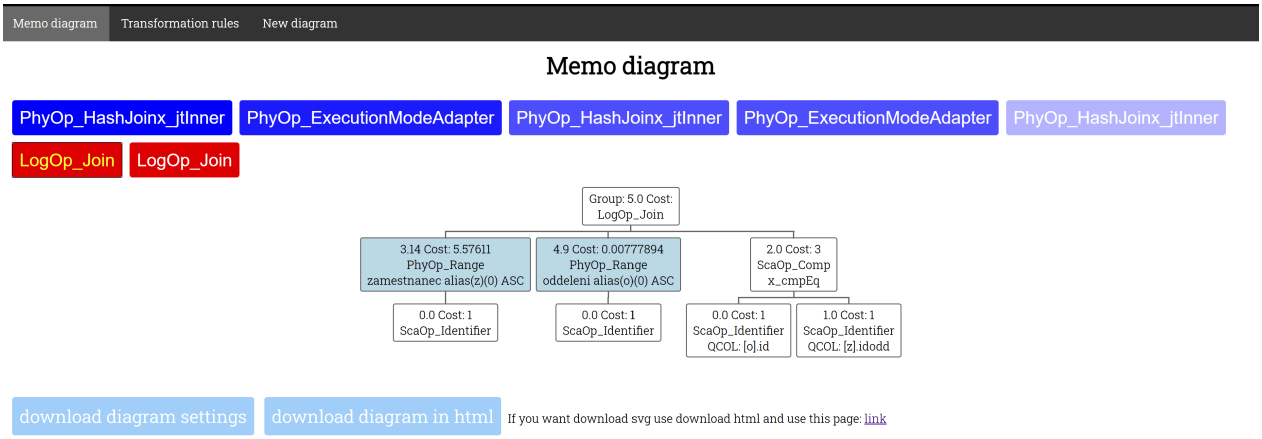


Obrázek 3.8: Aplikace zobrazení plánů

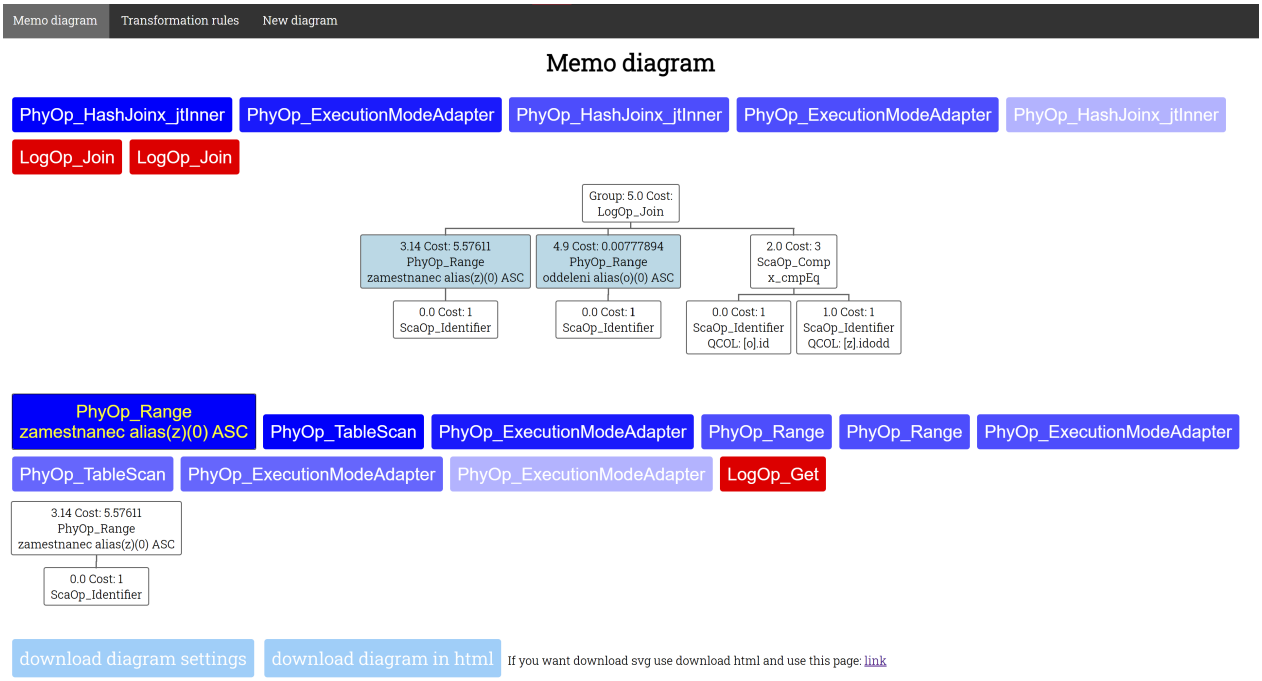
Nahoře na obrázku můžeme vidět tlačítka operátorů, které reprezentují root operátory. Jsou rozděleny na 2 barvy. Modrá reprezentuje fyzický operátor a červená logický. Výraznost modré barvy reprezentuje cenu fyzických operátorů. Čím menší cena, tím výraznější modrá barva.

Dole se nachází tlačítko pro stažení nastavení diagramů, které se dá nahrát na úvodní stránce aplikace. Slouží pro znovu zobrazení diagramů v aplikaci po zavření stránky. Také si můžeme stáhnout celý diagram do html pro tisk.

Pokud klikneme na logický operátor ze seznamu, zde například logický operátor *joinu* (obrázek 3.9), uvidíme že se plán skládá i z modrých operátorů. To značí, že jeho kořen nemá přesně nadefinované, z čeho se skládá. Má nadefinovanou jen memo skupinu, ze které se skládá. Modře označený operátor je operátor s nejmenší cenou z dané skupiny. Následně operátor jde rozkliknout pro zobrazení ostatních operátorů ze stejné skupiny jako na obrázku 3.10.



Obrázek 3.9: Aplikace zobrazení logického plánů

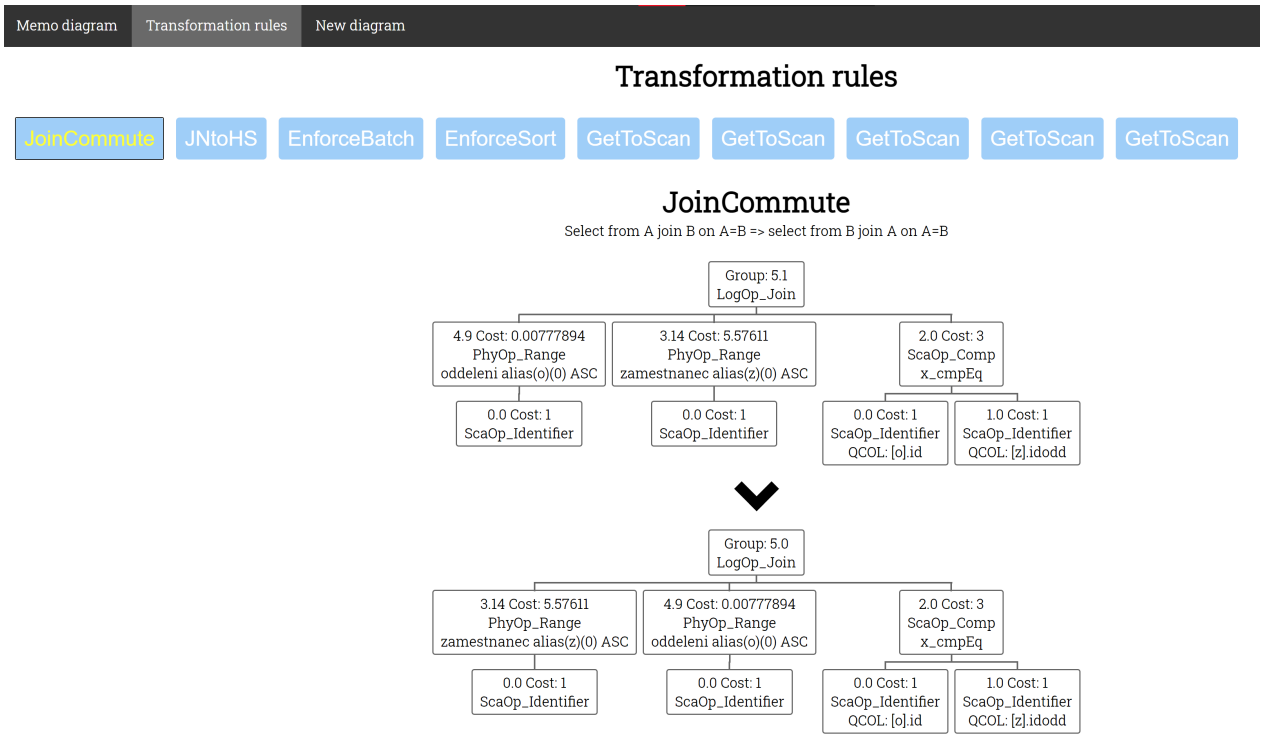


Obrázek 3.10: Aplikace zobrazení alternativních operátorů

Pokud bychom místo modře označeného operátoru chtěli zobrazit v plánu jiný operátor, můžeme na daný operátor kliknout scrolovacím tlačítkem a zvolit operátor, kterým ho chceme nahradit. V plánu se následně zobrazí místo původního operátoru, váš operátor.

3.4.3 Zobrazení přepisovacích pravidel

Druhá část aplikace slouží pro zobrazení přepisovacích pravidel. V horním menu stačí kliknout na *Transformation rules* a zobrazí se nám seznam přepisovacích pravidel, která byla použita pro daný dotaz. Jednotlivá přepisovací pravidla následně můžeme rozklikávat, kde se nám zobrazí plán před uplatněním přepisovacího pravidla a po uplatnění jako na obrázku 3.11.



Obrázek 3.11: Aplikace zobrazení přepisovacích pravidel

Kapitola 4

Závěr

Postup, jakým optimalizátor došel k finálnímu plánu bývá skryt v nepřehledné memo struktuře. Samotný SQL server nedává žádnou oficiální dokumentaci k memo struktuře a ani k přepisovacím pravidlům. Proto je složité ze samostatného SQL serveru zjistit podrobnější kroky optimalizace vedoucí k finálnímu plánu.

V bakalářské práci jsme si popsali podrobněji optimalizaci dotazu. Podle čeho optimalizátor vybírá finální plán a kde SQL server ukládá informace o optimalizaci, které sloužily jako hlavní část pro mou vizualizaci v aplikaci. Cíl bakalářské práce, vytvořit aplikaci pro vizuální zobrazení memo struktury a přepisovacích pravidel, byl splněn. Na tomto systému můžeme přehledně zjistit, jakým způsobem byl dotaz zpracován a jaké přepisovací pravidla vedla k tomuto zpracování. Aplikaci je možné rozšířit o další přepisovací pravidla.

Tato práce byla pro mě velkým přínosem. Rozšířil jsem si znalosti o to, jak SQL server, a v základu všechny SQL databáze, fungují. Převážně v oblasti optimalizace. Bakalářskou práci jsem začal dělat s velmi základními znalostmi o tom, jak databáze funguje. Pomocí doporučené literatury a jiných zdrojů jsem dokázal pochopit, co dělají jednotlivé části v databázi, a hlavně jak přesně funguje optimalizace dotazů spojená s přepisovacími pravidly a memo struktury. Díky těmto novým znalostem jsem následně mohl analyzovat seznamy přepisovacích pravidel pro velké množství SQL dotazů a snažil jsem se tak pochopit podstatu jednotlivých přepisovacích pravidel, pro které neexistuje podrobnější dokumentace. Také jsem si prohloubil své dosavadní schopnosti programování v jazyce PHP a nadstavbou javascriptu *jquery*.

Literatura

1. *Declarative programming* [online] [cit. 2021-03-20]. Dostupné z: https://en.wikipedia.org/wiki/Declarative_programming.
2. NEVAREZ, Benjamin. Inside the SQL Server Query Optimizer. [B.r.], S.l.: Red Gate Books, 2011. ISBN 9781906434601.
3. *Parameter Sniffing in SQL Server* [online] [cit. 2021-03-20]. Dostupné z: https://www.brentozar.com/archive/2013/06/the-elephant-and-the-mouse-or-parameter-sniffing-in-sql-server/.
4. *MySQL Aggregate Functions* [online] [cit. 2021-03-20]. Dostupné z: https://www.mysqltutorial.org/mysql-aggregate-functions.aspx.
5. *Inside the Optimizer: Constructing a Plan – Part 3* [online] [cit. 2021-03-20]. Dostupné z: https://www.sql.kiwi/2010/07/inside-the-optimiser-constructing-a-plan-part-3.html.
6. *Batch Mode* [online] [cit. 2021-03-20]. Dostupné z: http://www.queryprocessor.com/batch-mode-on-row-store/.
7. *Hints (Transact-SQL)* [online] [cit. 2021-03-20]. Dostupné z: https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql?view=sql-server-ver15.
8. *Inside the Optimizer: Constructing a Plan - Part 4* [online] [cit. 2021-03-20]. Dostupné z: https://www.sqlservercentral.com/articles/inside-the-optimizer-constructing-a-plan-part-4.
9. *Query Optimizer Deep Dive – Part 3* [online] [cit. 2021-03-20]. Dostupné z: https://www.sql.kiwi/2012/04/query-optimizer-deep-dive-part-3.html.
10. *Tree Traversals (Inorder, Preorder and Postorder)* [online] [cit. 2021-03-20]. Dostupné z: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/.

Příloha A

Elektronická příloha

Součástí práce je elektronická příloha, která obsahuje:

- zdrojový kód aplikace
- link na online verzi webu