FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Nativní podpora vlastních analytických funkcí v PostgreSQL
Student:	Pavel Špecht
Vedoucí:	Ing. Pavel Stěhule
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Vlastní analytické nebo také "window" funkce lze v PostgreSQL implementovat nativně v jazyku C, případně plv8 (javascript) nebo zprostředkovaně jako agregační funkce v jakémkoliv jazyku, který podporuje možnost vytvářet agregační funkce.
Cílem práce je:
- seznámit se s nativním API
- navrhnout a realizovat stejné API pro programovací jazyk PLpgSQL (experimentální rozšíření, úpravy interpretu PLpgSQL)
- porovnat výkon implementace analytické funkce jako agregační funkce nebo pomocí nativního API.
Není cílem této práce připravit patch v kvalitě a obsahu, který by mohl být zařazen do komunitního vývoje. V této práci jde o přenesení funkcionality J8 do PLpgSQL a změření výkonu. Výsledkem by mělo být také doporučení, zda-li by se mělo stávající API PLpgSQL rozšířit nebo zda-li se nemá tomuto tématu věnovat další pozornost, jelikož je výkon s použitím agregačních funkcí dostatečný.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 30. ledna 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Nativní podpora vlastních analytických funkcí v PostgreSQL

Pavel Špecht

Katedra softwarového inženýrství
Vedoucí práce: Ing. Pavel Stěhule

1. června 2020

Poděkování

Rád bych poděkoval vedoucímu práce, Ing. Pavlovi Stěhule, za rychlou odezvu, ochotu, podporu a užitečné rady při vypracování práce. Rodina, přátelé a hlavně přítelkyně (slovo hlavně jsem byl přinucen napsat) mi byli velkou oporou.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 1. června 2020 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Špecht, Pavel. *Nativní podpora vlastních analytických funkcí v PostgreSQL.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Hlavní téma práce spočívá v nalezení způsobu, jak vytvořit *window* funkce v procedurálním jazyce PL/pgSQL, který je součástí databázového systému PostgreSQL. Nejdříve je představen koncept agregačních a *window* funkcí doplněný o jejich rozdíly a společné vlastnosti. Ve zdrojových kódech procedurálního jazyku se jsou identifikovány části, pomocí kterých se obecně funkce zpracovávají. Znalosti interních mechanismů je následně využito při tvorbě rozhraní umožňující operace s *window* funkcemi. Využívá se zde spojení několika již existujících vzorů, které ve výsledku umožní předání objektu obsahujícího potřebné informace na místa, kde se *window* funkce vytváří. Pro práci s tímto objektem je připravena sada funkcí sloužící jako prostředník mezi interpretem jazyku PL/pgSQL a tělem vlastní *window* funkce. V závěrečné části práce se porovná rychlost zpracování dotazu funkce v jazyce PL/pgSQL s ostatními existujícími možnostmi, jak lze v systému PostgreSQL *window* funkci vytvořit. Více než dvojnásobná rychlost zpracování dotazu oproti „konkurenční“ variantě indikuje nejen splnění jednoho z hlavních cílů, ale i konkurenceschopnost rozhraní. Pozitivní výsledky testů ukazují, že by mělo smysl rozhraní realizovat, neboť by bylo v praxi použitelné.

Klíčová slova implementace rozhraní, window funkce, PL/pgSQL, PostgreSQL, procedurální jazyk, porovnání rychlostí zpracování dotazu

Abstract

The main topic of this thesis can be considered as finding the way, how to create window functions in the procedural language PL/pgSQL, which is part of the database system PostgreSQL. Firstly, aggregate and window functions are described with their differences and common characteristics included. Secondly, key parts of the procedural language, responsible for function processing in general, are identified. The system knowledge allows the author to create interface managing window functions. Combination of several existing patterns provides access from the PL/pgSQL function body to the object, which bears necessary information. Set of functions, used as an intermediary between the PL/pgSQL interpreter and a window function body, is prepared to operate with the mentioned object. Lastly, interface creation is described along with performance comparison of custom window function based on SQL query with other existing possibilities of creating window function in PostgreSQL. More than twice faster query processing against "rival" option indicates a fulfilment of the thesis' goal. Moreover, a competitiveness of the interface was achieved. Promising test results imply the usability of possible interface.

Keywords interface implementation, window functions, PL/pgSQL, PostgreSQL, procedural language, query performance comparison

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Relační databázový systém PostgreSQL získal za řadu let své existence mnoho věrných uživatelů. Veřejně dostupné zdrojové kódy, vytvořené odborníky ve svém oboru v jazyce C, mají bezesporu mnoho kladných i negativních stránek. Jako každý moderní databázový systém nabízí i PostgreSQL možnost pracovat s několika procedurálními jazyky. V této práci bude blíže představen procedurální jazyk PL/pgSQL.

Většina databázových systémů umožňuje uživateli provádět nad databází SQL dotazy obsahující agregační funkce. PostgreSQL není výjimkou. Analytické funkce představují do jisté míry alternativu k agregačním funkcím. V PostgreSQL jsou rovněž podporovány. V případě použití analytických funkcí je k dispozici rozhraní napsané v jazyce C (zkráceně bude označováno jako C API). Z jiných variant lze použít JavaScript rozhraní, zapouzdřující funkce ze C API, napsané v rámci důvěryhodného [1] procedurálního jazyku PL/v8.

Implementovat vlastní *window* [2] funkce ve vysokoúrovňovém procedurálním jazyku PL/pgSQL nelze, neboť není zpřístupněno interní API, na kterém jsou *window* funkce postavené. Agregační funkce je možné vytvořit v rámci PostgreSQL i v PL/pgSQL. Najde-li se řešení, jak zpřístupnit *window* funkce v PL/pgSQL, využijí jej noví i stávající uživatelé PostgreSQL, pokud si budou chtít napsat vlastní *window* funkci přímo v PL/pgSQL, protože při implementaci pomocí C API nelze definici funkce umístit do SQL skriptu. Je totiž nutné připojit soubor v jazyce C s definicí funkce.

Přesné téma práce si autor nezvolil sám, ale kvůli předchozím akademickým i praktickým zkušenostem hledal v databázově orientované oblasti. Předchozí zkušenost se systémem PostgreSQL se skládala pouze z pár hodin během výuky na vysoké škole. Je nutné podoktnout, že práce neslouží jako oficiální rozšíření systému PostgreSQL, ale pouze zkoumá nové nevyužité možnosti, které do oficiální verze mohou být následně přidány. Aby čtenář získal

[1] Běžný uživatel (práva) v něm může vytvářet funkce či procedury (*trusted language*)

[2] Označení analytická funkce je ekvivalentní, ale v rámci práce autor preferuje *window*

potřebnou znalost o tom, čeho se práce týká, v teoretické části se nejprve seznámí s agregačními a *window* funkcemi. Bez jejich vzájemného propojení by nebylo možné splnit jeden z úkolů v praktické části. Po popisu a porovnání zmíněných funkcí následuje stručný úvod do struktury zdrojových kódů PostgreSQL. Aby bylo možné vytvořit nové rozhraní pro *window* funkce, je nutné vědět, jak pracují interní mechanismy v PL/pgSQL, jak je funkce v tomto jazyce zpracována, a jakou část práce vykonávají jednotlivé komponenty. Jelikož již existují způsoby, jak *window* funkci v rámci PostgreSQL systému vytvořit, jsou tyto možnosti představeny včetně ukázek funkcí pomocí těchto technologií vytvořených.

Praktická část práce začíná popisem kroků a úprav, které byly třeba vykonat, aby bylo možné předat objekt obsahující potřebná data do těla PL/pgSQL funkce. Aby rozhraní fungovalo stejně jako je tomu v jazyce C, bylo potřeba vytvořit sadu pomocných funkcí, které slouží jako prostředník mezi interpretem PL/pgSQL a tělem funkce. Následně se vytvoří vlastní *window* funkce využívající nové PL/pgSQL rozhraní, otestuje se, že funguje správně, a následně proběhne testování všech možností, jak vytvořit *window* funkci v systému PostgreSQL (ve smyslu rychlosti zpracování dotazu nad stejnou vstupní množinou). Získané hodnoty se porovnají, aby byl představen výsledek, jak si nové řešení vede oproti ostatním. Na závěr je z naměřených dat vyvozena konkurenceschopnost nového řešení.

Cíle práce

I když je teoretická část důležitou součástí práce, nemá jasně měřitelný cíl pomocí SMART metodiky (používá se v projektovém řízení ke stanovení cílů). Bez definování pojmů a objasnění vnitřních principů v systému PostgreSQL by nebylo možné splnit cíle praktické části. Mezi klíčové prvky teoretické části patří vytvoření *window* funkce, zpracování funkce komponentami v PL/pgSQL a zmapování existujících způsobů, jak vytvořit *window* funkci v PostgreSQL. Jedná se o nutný úvod před hodnotitelnými výkony. Praktická část práce obsahuje tři cíle. Prvním cílem je implementovat funkční rozhraní pro práci s *window* funkcemi pomocí PL/pgSQL. Mělo by umožňovat ekvivalentní řešení problémů, jako je tomu u C API. Druhý cíl bude splněn, pokud *window* funkce, napsaná v procedurálním jazyce PL/pgSQL, bude vracet stejný výsledek jako její vestavěný funkční ekvivalent. Splnění třetího cíle lze definovat jako dosažení stejných či lepších výsledků v testování nově vzniklého rozhraní s existujícím „konkurenčním“ řešením (být lepší, než C API se jeví jako vysoce nepravděpodobné). Z textu je patrná jasná návaznost a závislost jednotlivých cílů, proto bez úspěšného zvládnutí předchozího nelze zvládnout následný cíl.

Kapitola 1

Agregační a window funkce

Nejen kvůli názvu práce se dá předpokládat, že se primárně shrnou *window* funkce (používá se také označení analytické, ale v textu bude preferováno označení *window*). Jelikož se v praktické části používají i agregační funkce, je vhodné popsat i jejich činnost. Za společný prvek těchto funkcí lze označit sumarizaci a sdružení dat. Následující řádky přiblíží to, jak obě funkce použít, vytvořit a čím se liší [1].

1.1 Agregační funkce

Přes rozmanité spektrum použití mají agregační funkce společné sdružení řádků do skupin na základě definované logiky, kdy se následně hodnota společná pro celou skupinu vrátí na výstupu v podobě jediného řádku. Vykonání agregační funkce se skládá v systému PostgreSQL nejčastěji ze tří základních částí. Nejprve se nastaví počáteční podmínky pro proměnné, do kterých se ve druhé části po vyhodnocení řádku ze vstupu v dané skupině uloží aktuální mezivýpočet. Na závěr se typicky provede konečná operace s nasbíranými daty (např. dosazení hodnot do matematického vzorce). SQL dotaz s agregační funkcí zpravidla, ne však povinně, obsahuje klíčové konstrukce `GROUP BY` a `HAVING`. První zmíněná konstrukce, `GROUP BY`, slouží ke sloučení výsledku dotazu podle společné vlastnosti do skupin určených podle hodnot ve specifikovaných sloupcích. Konstrukce `HAVING` umožňuje omezit řádky na výstupu podle vypočítané hodnoty agregační funkce [2, 3].

Na typickém zástupci, funkci vypočítávající součet hodnot s názvem `sum`, je demonstrováno použití. V první fázi se dočasný součet nastaví na nulovou hodnotu. Následně se po každém načteném řádku ze skupiny aktualizuje dočasný součet hodnot. Ve třetí fázi se součet vrátí finální funkcí. Na další straně je uveden příklad vytvoření vlastní sumarizační agregační funkce.

```
CREATE FUNCTION custom_accumulator (tmp int, new_value int)
RETURNS int AS
$$
DECLARE
    new_tmp int;
BEGIN
    new_tmp := tmp + new_value;
    RETURN new_tmp;
END;
$$
LANGUAGE plpgsql IMMUTABLE;
```

Výpis kódu 1.1: Akumulátor pro agregační funkci

Funkce `custom_accumulator`, dostupná ve výpisu kódu 1.1, představuje akumulátor (aktualizuje mezivýsledek) pro agregační funkci [3]. Jelikož sloupec hodnot v ukázkové tabulce je datového typu *Integer*, proto i agregační funkce u všech svých proměnných používá tento datový typ. Klíčové slovo `IMMUTABLE` signalizuje, že se jedná o *read-only* přístup do databáze. Funkce tedy nemá přístup ke zdrojům databáze a je závislá čistě na funkčních argumentech. Pokud bude slovo použito nevhodně, může to vést k neočekávanému chování systému. Pomocí klauzule `LANGUAGE` se specifikuje použitý jazyk. V tomto případě procedurální jazyk PL/pgSQL, který bude v příští kapitole přiblížen. Značka dollaru, případně 36. znak v tabulce *ASCII*, v tomto případě nemá zvláštní význam. Jinak slouží k *escaping* (způsob kódování) jednoduchých uvozovek v textovém řetězci, neboť použití některých znaků má definovaný význam [4].

```
CREATE FUNCTION final_function(tmp int)
RETURNS int AS
$$
BEGIN
    RETURN tmp;
END;
$$
LANGUAGE plpgsql IMMUTABLE STRICT;
```

Výpis kódu 1.2: Finální funkce pro agregační funkci

Finální funkce v tomto případě není složitá, pouze vrací celkový součet hodnot, které nashromáždil akumulátor. V jiných případech se zde mohou počítat složité matematické vzorce na základě akumulátorem dodaných informací.

[3] Komentáře se napříč prací u výpisu kódu nacházejí v samostatných odstavcích. Ve zdrojových kódech samozřejmě jsou.

```
CREATE AGGREGATE customSUM(int) (
    sfunc = custom_accumulator,
    stype = int,
    finalfunc = final_function,
    initcond = '0'
);
```

Výpis kódu 1.3: Vytvoření vlastní agregační funkce

Registrace nové agregační funkce ve výpisu kódu 1.3 obsahuje kromě specifikované akumulátorové funkce (`sfunc`) a finální funkce (`finalfunc`) i popis datového typu sloužícího pro potřeby mezivýsledku (`stype`) a počáteční nastavení hodnot (`initcond`) [5].

1.2 Window funkce

Window funkce typicky nepočítají hodnotu na základě jednoho řádku, pokud tak dotaz není přímo specifikován, ale na základě definované skupiny řádků nazývané jako *partition* či její podskupiny nazývané *frame*. *Frame* lze chápat jako uspořádanou skupinu řádků pro každý řádek v rámci *partition*. Výslednou hodnotu obsahuje každý řádek ze vstupu a členění do skupin má pouze logický vztah při zpracování funkce. Funkce neumožňuje přístup pouze k datům vztahujícím se k aktuálnímu řádku, ale kvůli výpočtu lze navíc přistupovat k hodnotám ostatních řádků [6].

Povinná konstrukce `OVER`, následovaná v závorkách nepovinnou konstrukcí `PARTITION BY` a `ORDER BY`, určuje obsah jednotlivých skupin řádků. Pokud se v závorkách žádný výraz neuvede, počítaná hodnota je společná pro všechny řádky. Díky konstrukci `PARTITION BY` lze uvést, pomocí jakého sloupce se budou jednotlivé *partition* dělit. Následující konstrukce `ORDER BY` specifikuje pořadí řádku při zpracování *window* funkcí v rámci *partition*. Není-li použita v dotazu konstrukce `ORDER BY`, *frame* se skládá z celé *partition*. Pokud je konstrukce `ORDER BY` součástí dotazu, za *frame* se standardně označí skupina řádků od začátku *partition* po daný či hodnotou ekvivalentní řádek [7, 8].

```
Datum window_row_number(PG_FUNCTION_ARGS)
{
    WindowObject winobj = PG_WINDOW_OBJECT();
    Int64 curpos = WinGetCurrentPosition(winobj);
    WinSetMarkPosition(winobj, curpos);
    PG_RETURN_INT64(curpos + 1);
}
```

Výpis kódu 1.4: Ukázka *window* funkce z *windowfuncs.s*

Na příkladu (výpis kódu 1.4) je představena ukázka *window* funkce využívající C API v systému PostgreSQL. Pozice řádku v rámci aktuální *partition*

je vrácena touto funkcí. Pokud není rozdělení *partition* specifikováno, používá se celá vstupní množina. Datový abstraktní typ `Datum`, sloužící pro zprostředkování komunikace mezi rozhraními, bude přiblížen později. Pomocí objektu `FunctionCallInfo`, dostupného přes makro `PG_FUNCTION_ARGS`, je umožněno přistupovat k poli funkčních argumentů. Pro přístup k jednotlivým argumentům se používá syntaxe „`PG_GETARG_datovýTyp(čísloArgumentu)`" [5].

1.3 Srovnání agregačních a window funkcí

Agregační a *window* funkce mají mnoho společného. Výpočet funkcí zpravidla nezávisí na jednom řádku tabulky či pohledu, ale na množině řádků. Hlavní rozdíl lze spatřit v tom, jak je výsledek reprezentován. Zatímco na výstupu agregační funkce je řádek za každou shlukující skupinu (konstrukce `GROUP BY`), u *window* funkcí vypočtenou hodnotu obsahuje každý řádek ze vstupu a možnost zobrazit si jednotlivé řádky se tedy neztrácí. Je-li nutné získat na výstupu jednotlivé řádky z agregační funkce, máme dvě možnosti. Buď si napsat vlastní *window* funkci, která bude řešit stejný problém, nebo při volání agregační funkce uvést konstrukci `OVER`, neboť každá agregační funkce může být použita jako *window* funkce [9, 10].

```
postgres=# SELECT * FROM testTable;
 value | dep
-------+-----
     1 | A
     2 | A
     3 | B
     5 | B
     6 | C
     4 | C
     3 | A
(7 rows)

postgres=# SELECT dep, avg(value) FROM testTable GROUP BY dep ORDER BY dep;
 dep |        avg
-----+--------------------
 A   | 2.0000000000000000
 B   | 4.0000000000000000
 C   | 5.0000000000000000
(3 rows)

postgres=# SELECT dep, avg(value) OVER (PARTITION BY dep ORDER BY dep) FROM testTable;
 dep |        avg
-----+--------------------
 A   | 2.0000000000000000
 A   | 2.0000000000000000
 A   | 2.0000000000000000
 B   | 4.0000000000000000
 B   | 4.0000000000000000
 C   | 5.0000000000000000
 C   | 5.0000000000000000
(7 rows)
```

Obrázek 1.1: Porovnání výstupu agregační (druhý SQL dotaz) a *window* funkce (třetí SQL dotaz)

Kapitola 2

Databázový systém PostgreSQL

Relační databázový systém PostgreSQL, pod licencí podobnou MIT [4], umožňuje uživateli vytvářet vlastní funkce v několika programovacích jazycích. Jednou z jeho předností jsou veřejně přístupné zdrojové kódy. I když byl původně zamýšlen pro platformy typu UNIX, v současné době podporuje například i systém Windows. Často používaný název Postgres (původně se takto nazýval) není oficiálním názvem, a proto nebude v práci využíván [11, 12].

2.1 Datový typ Datum

Abstraktní datový typ `Datum` slouží v prostředí Postgres pro interní reprezentaci hodnoty bez přidané informace o tom, k jakému datovému typu přísluší. V systému PostgreSQL se používá primárně pro předání hodnot mezi rozhraními, kde není nutné pracovat se skutečnými datovými typy předávaných hodnot. Při použití si musí být vývojář vždy jistý skutečným datovým typem proměnné. V těle funkce v jazyce C používá vývojář datové typy podle svého uvážení a na konci se provede konverze výstupní proměnné na `Datum` [13].

Pro práci s `Datum` existuje sada připravených pomocných funkcí umožňující konverzi mezi `Datum` a základními datovými prvky oběma směry. V závislosti na podmínce jsou tyto funkce definované pomocí maker umožňující drobné funkční rozdíly. Z důvodu absence informace o tom, zda je `Datum` rovné `NULL`, k dispozici se naskýtá struktura `NullableDatum`, skládající se z *value* (`Datum`) a *isnull* (`boolean`) [5].

[4] Neomezené použití a šíření, je nutné pouze zachovat autora a znění licence

```
Datum window_first_value(PG_FUNCTION_ARGS)
{
    WindowObject winobj = PG_WINDOW_OBJECT();
    Datum   result;
    bool    isnull;
    result = WinGetFuncArgInFrame(winobj, 0, 0, WINDOW_SEEK_HEAD, true,
        &isnull, NULL);
    if (isnull)
        PG_RETURN_NULL();
    PG_RETURN_DATUM(result);
}
```

Výpis kódu 2.1: Použití datového typu Datum

Výpis kódu 2.1 zobrazuje použití abstraktního datového typu `Datum` napříč funkcí `window_first_value`, která je součástí C API pro práci s *window* funkcemi. Makro `PG_RETURN_DATUM` slouží k předání proměnné typu `Datum` [5].

2.2 Struktura Node

Na hierarchii objektů napříč zdrojovým kódem by se hodilo použít dědičnost tříd. Nicméně v programovacím jazyce C není dědičnost podporovaná. Východiskem pro vývojáře PostgreSQL se stala kompozice struktur. Jako analogii k abstraktní třídě slouží struktura `Node`. Kompozitní typy (reprezentace struktury řádku či záznamu), vytvořené pomocí Node struktury, podporují např. serializaci [5] či kopírování. Pomocí objektu `NodeTag` se určuje účel a obsah struktury. V adresáři *src/backend/nodes* lze nalézt funkce pro práci s `Node`, od konverzí a porovnání, až po práci se stromovými strukturami a paměťovou alokací. Každá `Node` struktura se vytváří pomocí funkce `makeNode` nebo `newNode`. Pokud se `Node` vytvoří pomocí `makeNode`, bude obsahovat `NodeTag` s hodnotou, která jednoznačně určuje jeho druh (např. `T_Material` = 39, `T_Unique` = 44). Existuje 14 hlavních skupin `NodeTag`, které funkčně vymezují používání struktury `Node`. Většina z nich odkazuje do jiných souborů ve stejném adresáři [13].

2.3 Adresářová struktura zdrojových kódů PostgreSQL

V kořenovém adresáři PostgreSQL se nacházejí čtyři adresáře. Kromě dokumentace a konfiguračních souborů, lze zde nalézt adresář s uloženými extenzemi a další se zdrojovými kódy. Poslední dvě uložiště budou čtenáři přiblíženy.

[5] Proces konverze objektu do sériové podoby

2.3.1 Extenze v PostgreSQL - adresář contrib

V adresáři *contrib* se nacházejí předpřipravené extenze schválené komunitou. Extenzí nazveme skupinu funkcionalit, které nejsou nutnou součástí systému PostreSQL, ale lze je nainstalovat a používat bez nutnosti restartování databázového serveru. Aby je bylo možné použít, musí se nainstalovat příkazem `CREATE EXTENSION`. Každá extenze se musí skládat minimálně ze dvou souborů. Soubor s povinným názvem *„název-extenze.control"* specifikuje aktuální verzi SQL skriptu, umístění zdrojových souborů či seznam extenzí, které je nutné nainstalovat, neboť na nich závisí správný běh této extenze. Druhý povinný soubor, o nutné jmenné konvenci *„jménoExtenze–verzeSQLskriptu.sql"*, obsahuje deklarace funkcionalit, které nová extenze přináší. Pro automatickou instalaci extenze lze přidat *Makefile*, ale extenzi lze nainstalovat i bez něj [14].

I když jde vytvořit extenzi i rychleji, tento způsob se více podobá již existujícím extenzím v adresáři *contrib*. Extenze bude obsahovat jedinou funkci `my_concat` umožňující sloučení řetězců. Nejdříve se v adresáři *contrib* vytvoří adresář *my_concat* pojmenovaný podle názvu extenze. Toto není povinný krok, spíše se jedná o dodržení konvence a udržení pořádku ve struktuře. V adresáři *my_concat* se vytvoří následující čtyři soubory [15].

2.3.1.1 SQL skript v extenzi

Podle dříve zmíněných konvencí pojmenovaný soubor `my_concat--1.0.sql` obsahuje deklaraci slučovací textové funkce. Konstrukce `AS MODULE_PATHNAME` signalizuje, že v separovaném souboru určeném proměnnou `MODULE_PATHNAME` se nachází tělo funkce. Funkce je napsaná v jazyce C a indikátory `IMMUTABLE` (funkce vrací opakovaně stejný výsledek, který je závislý pouze na funkčních argumentech), `STRICT` (při některém argumentu nesoucí hodnotu NULL se ve funkci pokračuje) a `PARALLEL SAFE` (může být použit paralelní režim bez omezení) specifikují vlastnosti funkce [16].

```
CREATE FUNCTION my_concat(text, text)
RETURNS text
AS 'MODULE_PATHNAME'
LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;
```

Výpis kódu 2.2: Deklarace funkce *my_concat* v souboru *my_concat–1.0.sql*

2.3.1.2 Soubor možností a nastavení

V „kontrolním" souboru `my_concat.control`, dostupném ve výpisu kódu 2.3, lze nalézt důležité informace k extenzi. Výchozí verze je specifikovaná pomocí proměnné `default_version`. Pomocí proměnné `relocatable` se signalizuje možnost přesunutí objektů extenze do jiného schéma [17].

```
comment = 'ukazka vytvoreni extenze - my_concat'
default_version = '1.0'
module_pathname = '\$libdir/my_concat'
relocatable = true
```

Výpis kódu 2.3: Obsah souboru *my_concat.control*

2.3.1.3 Soubor s definicí funkce v jazyce C

V souboru *my_concat.c* se nachází definice slučovací funkce. Zajištění kompatibility knihoven s instalovanou verzí PostgreSQL má na starosti makro `PG_MODULE_MAGIC`. Makro `PG_FUNCTION_INFO_V1` zprostředkovává dostupnost funkce v SQL. Text funkce je napsán na základě funkce `text_catenate` ze souboru *src/backend/utils/adt/varlena.c* [18].

```
#include "postgres.h"
#include "catalog/pg_collation.h"
#include "utils/builtins.h"
#include "utils/formatting.h"
#include "utils/hashutils.h"
#include "utils/varlena.h"

PG_MODULE_MAGIC;
PG_FUNCTION_INFO_V1(my_concat);
Datum my_concat (PG_FUNCTION_ARGS)
{
    text    *left = PG_GETARG_TEXT_PP(0);
    text    *right = PG_GETARG_TEXT_PP(1);
    text    *result;
    int     len, len1, len2;
    char    *ptr;

    len1 = VARSIZE_ANY_EXHDR(left);
    len2 = VARSIZE_ANY_EXHDR(right);
    len = len1 + len2 + VARHDRSZ;
    result = (text *) palloc(len);
    SET_VARSIZE(result, len);
    ptr = VARDATA(result);
    if (len1 > 0)
        memcpy(ptr, VARDATA_ANY(left), len1);
    if (len2 > 0)
        memcpy(ptr + len1, VARDATA_ANY(right), len2);
    PG_RETURN_TEXT_P(result);
}
```

Výpis kódu 2.4: Definice funkce *my_concat*

2.3.1.4 Makefile

Soubor *Makefile* není povinnou součástí, ale umožňuje hromadnou kompilaci souborů. Do *Makefile* souboru, který se nachází v adresáři *contrib*, se musí nakonec přidat záznam o podsložce *my_concat*.

```
MODULES = my_concat
EXTENSION = my_concat
DATA = my_concat--1.0.sql
PGFILEDESC = "Ukazkova extenze - slucovani retezcu"
REGRESS = my_concat

ifdef USE_PGXS
    PG_CONFIG = pg_config
    PGXS := $(shell $(PG_CONFIG) --pgxs)
    include $(PGXS)
else
    subdir = contrib/my_concat
    top_builddir = ../..
    include $(top_builddir)/src/Makefile.global
    include $(top_srcdir)/contrib/contrib-global.mk
endif
```

Výpis kódu 2.5: Obsah *Makefile* souboru náležícímu vlastní extenzi

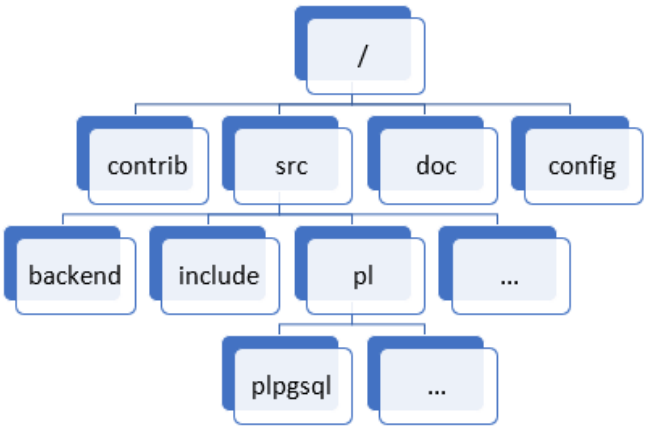
Po kompilaci a instalaci souborů se vytvoření extenze dosáhne příkazem `create extension my_concat`, avšak příkaz je nutné spustit z terminálu PostgreSQL pomocí příkazu `psql` [20].

2.3.2 Zdrojové kódy PostgreSQL - adresář src

Hlavičkové soubory se kvůli zjednodušenému přístupu při uvedení závislosti shromáždily do odděleného adresáře (*include*) od definic funkcí (*backend*). Na schématu níže (obrázek 2.1) je znázorněna zjednodušená adresářová struktura zdrojových kódů PostgreSQL. Nejen v praktické části se často odkazuje na soubory a adresáře ve struktuře zdrojových kódů, které jsou klíčové pro účely práce. Následuje popis nejdůležitějších z nich.

- /src/backend/executor/

  Část systému PostgreSQL, která má na starosti zpracování exekučního plánu. Zatímco soubor *nodeAgg.c*, nacházející se v tomto adresáři, slouží pro zpracování agregačních funkcí, v souboru *nodewindowAgg.c* naleznete funkce a struktury pro práci s *window* funkcemi. Tyto funkce umožňují například ukládat informace pro pozdější použití v rámci *window* funkce či získat počet řádků v aktuální *partition*. Zmíněné funkce jsou základem C API (windowapi.h) popsaného níže [21].



Obrázek 2.1: Zjednodušené schéma adresářové struktury zdrojových kódů systému PostgreSQL

- /src/include/windowapi.h

  Makra a deklarace funkcí dohromady tvoří rozhraní napsané v jazyce C pro práci s *window* funkcemi. Definice funkcí nacházející se v komponentě *executor* je uložená v souboru *nodewindowAgg.c*. Ukazatel na strukturu `WindowObjectData` reprezentuje *window* objekt (obsahuje informace o funkčních argumentech či o aktuální pozici a počtu řádků v aktuální *partition*), jehož přístupnost z těla PL/pgSQL funkce je klíčová pro splnění hlavního cíle práce [5].

- /src/include/catalog/

  V systému postgreSQL lze využít dva způsoby registrace funkce. První možností je použít makro `PG_FUNCTION_INFO_V1` je určen pro extenze, které jsou implementovány jako dynamicky linkované knihovny. Není pak nutné restartovat systém. Druhý způsob, jenž lze využít i pro vlastní datové typy apod., se používá v jádru PostgreSQL. Veřejný interface z pohledu SQL je popsán datovými soubory, které generují části systémového katalogu a kód v jazyce C. Nutností je kompilace a restart systému. Každému registrovanému prvku se musí ručně přidělit unikátní identifikátor, OID, který je v prvním případě přiřazen automaticky. Pro zjištění dostupných identifikátorů je k dispozici skript. V práci se v tomto adresáři registruje vlastní datový typ a funkce z nově vytvořeného rozhraní [18, 19].

- /src/backend/utils/adt/windowfuncs.c

  Soubor obsahuje předpřipravené *window* funkce využívající C API. Jedná se o užitečný zdroj inspirace, pokud je potřeba vytvořit vlastní *window* funkci v jazyce C. Mezi funkce definované v tomto souboru patří například `window_row_number` či `window_rank`, což jsou interní názvy pro funkce `row_number` a `rank` [5].

- /src/pl/

  Výchozí procedurální jazyky lze nalézt v tomto adresáři. Díky nim je možné psát vlastní funkce i v jiném jazyce než SQL a C. V základní verzi systému se nacházejí čtyři procedurální jazyky PL/Perl, PL/pgSQL (v následující kapitole bude přiblížen), PL/Python a PL/Tcl. Uživateli je umožněno si vytvořit vlastní procedurální jazyk [23].

2.4 Procedurální jazyk PL/pgSQL

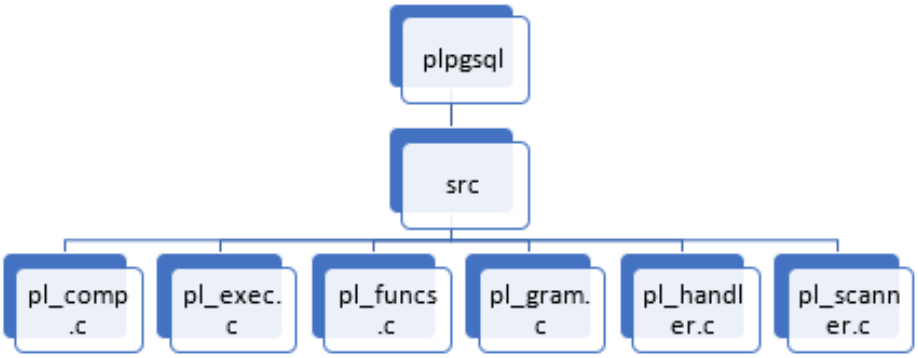
Od verze 9.0 primárně dostupný procedurální jazyk nabízí jednu velkou výhodu při zpracování SQL dotazů. Jelikož se každý SQL dotaz individuálně vyhodnocuje v databázovém serveru, pokud se klientská část a serverová část nachází na rozdílných zařízeních, může se při sérii dotazů výrazně zpomalovat režie přenášení dat po síti. Díky PL/pgSQL lze tento problém omezit, protože se příkazy spojí do bloku následně odeslaného na databázový server. Na klientskou část se vrátí z databázového serveru výsledek celého bloku. Jazyk umožňuje použití stejných funkcí, operátorů a datových typů, které lze využít v SQL [22].

2.4.1 Charakterizace jazyka

Inspirace tvůrců procedurálním jazykem PL/SQL pro relační databázový systém Oracle umožnila možnost snadné konverze procedur s Oracle systémem. PL/pgSQL umožňuje konstrukci SQL příkazů a jejich následné exekuci. I když neobohacuje PostgreSQL žádným novým datovým typem či vlastními funkcemi, zato nabízí jednoduchou možnost vytvoření triggerů, operátorů či agregačních funkcí. Pro deklaraci proměnné v těle funkce se používá příkaz `DECLARE`, nacházející se před příkazem `BEGIN`. Nastavení hodnoty proměnné jde provést dvěma způsoby. První využívající příkaz `SELECT INTO` vloží do proměnné výsledek SQL dotazu, zatímco použití konstrukce `:=` se více podobá klasickým programovacím jazykům. Obtížně detekovatelné chyby způsobené jmennými kolizemi proměnných a rezervovaných klíčových slov či sloupců je doporučené řešit použitím podtržítka (na rozdíl od Oracle systému se upřednostňuje lokální proměnná). Kvůli značnému propojení s SQL je možné používat libovolný SQL příkaz. Pokud je nutné vykonat funkci typu `VOID` (bez návratové hodnoty), používá se příkaz `PERFORM` [24].

2.4.2 Struktura zdrojových kódů PL/pgSQL

Jakoukoliv funkci, proceduru či anonymní blok je nutné před zpracováním zkompilovat, aby se zabránilo spuštění nevhodně navrženého kódu. *Compiler* zajišťuje syntaktickou analýzu pomocí *Parser*, kde se tělo funkce rozdělí na základní části (cykly, rozhodovací konstrukce). *Handler* rozpozná, zda se jedná o funkci, trigger či anonymní blok, a následně předá data do *Executor*, kde se zpracuje exekuční plán [25].



Obrázek 2.2: Zjednodušené schéma adresářové struktury zdrojových kódů procedurálního jazyku PL/pgSQL

Kapitola 3

Existující řešení window funkcí v systému PostgreSQL

Hlavní cíl práce spočívá v nalezení způsobu, jak vytvořit *window* funkci v procedurálním jazyku PL/pgSQL. Před předložením plánu realizace se zmapují aktuální možnosti řešení, ze kterých se dá načerpat inspirace. Když je v současné době potřeba vytvořit vlastní *window* funkce, lze si vybrat v prostředí PostgreSQL pouze ze dvou ověřených variant. Jedná se o rozhraní napsané v jazyce C (C API) a o rozhraní v procedurálním jazyce PL/v8 využívající v8 *engine* [26].

3.1 Rozhraní v jazyce C - windowapi.h

První způsob, jak vytvořit *window* funkci v systému PostgreSQL, spočívá v použití C API dostupného od roku 2008. Hlavičkový soubor *windowapi.h* shromažďuje všechny funkce a pomocné konstrukce pracující s *window* funkcemi do jednoho souboru. Avšak většina funkcí, které se v souboru nacházejí, jsou definovány v souboru *nodewindowAgg.c* v rámci komponenty *Executor*. Soubor *windowfunc.c* obsahuje již hotové funkce, využívající C API, zaregistrované a připravené pro SQL dotazy [27].

Následující seznam obsahuje makra a funkce ze souboru windowapi.h, které jsou klíčové pro práci s *window* funkcemi pomocí C API.

WindowObject
: ukazatel na strukturu `WindowObjectData`

PG_WINDOW_OBJECT
: makro zpřístupňující `fcinfo->context`

WindowObjectIsValid
: makro ověřující, zda je *window* objekt validní

WinGetPartitionLocalMemory
funkce slouží pro získání informace uložené v interní paměti

WinGetCurrentPosition
funkce vrací aktuální pozici řádku v rámci aktuální *partition*

WinGetPartitionRowCount
funkce vrací počet řádků v aktuální *partition*

WinSetMarkPosition
funkce, která posune aktuální pozici řádku na novou hodnotu (lze se pohybovat pouze vpřed)

WinRowsArePeers
funkce určující při specifikováném pořadí pomocí konstrukce `ORDER BY`, zda jsou si dvě pozice rovny hodnotou (v případě nepoužití `ORDER BY` konstrukce jsou si rovny všechny)

WinGetFuncArgInPartition
funkce vrací n-tý argument pozice specifikované na vstupu, kde se pozice volí od začátku *partition*, aktuální pozice řádku či od konce *partition*

WinGetFuncArgInFrame
funkce o stejné funkcionalitě jako u `WinGetFuncArgInPartition`, ale v rámci *frame* (definovaná skupina okolních řádků kolem aktuálního řádku)

WinGetFuncArgCurrent
funkce vrací n-tý argument řádku na aktuální pozici

V příkladu *window* funkce využívající C API (výpis kódu 3.1) lze vidět použití několika funkcí z výše popsaného souboru *windowfunc.c*. Jedná se o variantu agregační funkce `max`, která vypisuje maximální hodnotu ze seznamu řádků na vstupu. Pomocná struktura slouží k uložení položky pomocí funkce `WinGetPartitionLocalMemory`. Nejprve získáme `WindowObject` přes makro `PG_WINDOW_OBJECT` a zjistíme počet řádků v *partition*. Pokud není hodnota již vypočítaná, iteruje se v cyklu přes všechny řádky v *partition* a hledá se maximální hodnota. Na závěr se výsledek uloží do paměti a vrátí na výstupu [5].

```c
typedef struct window_memory_context
{
    int calculated_value;
}   window_memory_context;

Datum custom_window_max(PG_FUNCTION_ARGS)
{
    WindowObject win_obj = PG_WINDOW_OBJECT();
    int partition_count = WinGetPartitionRowCount(win_obj);
    window_memory_context * saved = (window_memory_context*)
        WinGetPartitionLocalMemory(win_obj,sizeof(window_memory_context));
    if (saved->calculated_value == 0)
    {
        int max_value = 0;
        bool is_set = false;
        for (int i = 0; i < partition_count; i++)
        {
            bool isnull;
            bool isout;
            int tmp_value = DatumGetInt32(WinGetFuncArgInPartition(win_obj,
                0, i, 1, false, &isnull, &isout));
            if(isout){
                elog(WARNING, "Row out of the frame");
            }
            else if(!isnull)
            {
                if(!is_set)
                {
                    max_value = tmp_value;
                    is_set = true;
                }
                else if(tmp_value > max_value)
                {
                    max_value = tmp_value;
                }
            }
        }
        saved->calculated_value = max_value;
    }
    return saved->calculated_value;
}
```

Výpis kódu 3.1: Vlastní *window* funkce využívající C API

3.2 PLV8

JavaScript extenze PL/v8 je druhý způsob, jak vytvořit *window* funkci v systému PostgreSQL. Stejné funkcionality jako u první varianty nebylo dosaženo vytvořením ekvivalentních funkcí, ale pomocí zapouzdření funkcí C API. Pro získání *window* objektu se používá syntaxe `plv8.get_window_object()`. Dvojice souborů je zodpovědná za práci s *window* funkcemi. Zatímco soubor *plv8.cc* indikuje volání *window* funkce a připraví funkční argumenty pro následné použití, soubor *plv8_func.cc* obsahuje funkce pro práci s analytickými funkcemi, jako je tomu u C API. Jelikož zde není podporována serializace, objekt uložený v paměti *window* funkce se uloží jako `String`. V adresáři s testy funkčnosti jsou vytvořeny základní analytické funkce sloužící také jako vzorové příklady [28].

```
CREATE FUNCTION my_plv8_sum(value numeric)
RETURNS numeric
AS
$$
var winobj = plv8.get_window_object();
var partition_count = winobj.get_partition_row_count();
var context = winobj.get_partition_local() || {};
var tmp_sum = 0;
if (!context.sum) {
    for (var i = 0; i < partition_count; i++) {
        tmp_sum += winobj.get_func_arg_in_partition(0, i,
            winobj.SEEK_HEAD, false);
    }
    context.sum = tmp_sum;
    winobj.set_partition_local(context);
}
return context.sum;
$$
LANGUAGE plv8 WINDOW;
```

Výpis kódu 3.2: Vlastní *window* funkce v procedurálním jazyce PLV8

Window funkce, zobrazená ve výpisu kódu 3.2, vrací sumu hodnot v dané *partition*. Funkce `get_func_arg_in_partition` vrací specifikovaný funkční argument řádků v *partition*. Pro získání počtu řádků, přes které se v cyklu iteruje, se používá funkce `get_partition_row_count`.

Kapitola 4

Implementace rozhraní

Po seznámení se s interními mechanismy v systému PostgreSQL nezbývalo než nově nabyté vědomosti využít v praxi. Praktická část byla vypracována ve virtualizované linuxové distribuci Ubuntu díky programu VirtualBox. Nejprve byly staženy zdrojové kódy systému PostgreSQL a v průběhu práce se doinstalovala (na několikátý pokus) extenze PL/v8. Zjednodušený postup instalace je k dispozici v souboru instalace.txt [29, 30, 31, 32].

4.1 Detekce window funkce

Pro vytváření funkce v systému PostgreSQL existuje přesně daná skupina parametrů a klauzulí, které určují, jaké vlastnosti bude mít výsledná funkce. Jedním z těchto parametrů je i `lang_name` určující jazyk, který funkce používá. Klauzule `WINDOW` značí, že se jedná o *window* funkci. Prvním krokem bylo ověření, zda komponenta jazyku PL/pgSQL zvaná *Parser* umožňuje kombinaci `LANGUAGE plpgsql` a `window`. V negativním případě by bylo nutné *Parser* pozměnit, jelikož by nebyl při zpracování funkce přístup k *window* objektu. Zmíněným *window* objektem se zde myslí ukazatel na strukturu `WindowObjectData`, který lze získat přetypováním `Node` ukazatele ve struktuře `FunctionCallInfoBaseData` (jedná se o součást každé funkce v PostgreSQL a její definice je k dispozici v souboru *src/include/fmgr.h*) [16].

V souboru *pl_exec.c*, který slouží jako interpret PL/pgSQL, existuje funkce `plpgsql_exec_function`, která se zavolá při každém zpracování funkce. Zápis do logu skutečně potvrdil, že v případě zavolání *window* funkce je přístup ke struktuře `WindowObjectData` k dispozici. Aby se činnosti výhradně spojené s *window* funkcemi neodehrávali i v ostatních případech, vytvořila se v rozhodovací konstrukci ve funkci `plpgsql_call_handler` náležící komponentě *Handler* možnost pro separování případu užití *window* funkce. Do C API se pro zjednodušení přidalo makro `CALLED_AS_WINDOW_OBJECT`, které ověřuje, že objekt `context` ve struktuře `fcinfo` lze použít jako validní `WindowObject`.

Do interpretu PL/pgSQL byla pro případ *window* funkce přidána kopie obecné funkce `plpgsql_exec_function`.

Na výpisu kódu 4.1 můžete vidět rozhodovací konstrukci v PL/pgSQL komponentě *Handler*, která detekuje případ *window* funkce pomocí makra `CALLED_AS_WINDOW_OBJECT`.

```
if(CALLED_AS_WINDOW_OBJECT(fcinfo))
{
    retval = plpgsql_exec_window_object(func,
      (WindowObject) fcinfo->context, fcinfo, NULL, NULL, !nonatomic);
}
else if (CALLED_AS_TRIGGER(fcinfo))
{
    retval = PointerGetDatum(plpgsql_exec_trigger(func,
        (TriggerData *) fcinfo->context));
}
else if (CALLED_AS_EVENT_TRIGGER(fcinfo))
{
    plpgsql_exec_event_trigger(func,
        (EventTriggerData *) fcinfo->context);
    retval = (Datum) 0;
}
else
    retval = plpgsql_exec_function(func, fcinfo, NULL,
        NULL,!nonatomic);
```

Výpis kódu 4.1: Rozhodovací konstrukce separující případ s *window* funkcí

4.2 Modifikace interpretu

Po detekci a separování do vlastní funkce v interpretu PL/pgSQL nastal čas vyřešit, jak `WindowObject` předat do těla PL/pgSQL funkce. První myšlenkou, která se ukázala jako chybná, bylo použití hashovací tabulky, která bude schopna udržet hodnotu po celou dobu exekuce funkce. Do tabulky se ukládají struktury obsahující unikátní klíč a hodnotu o libovolném datovém typu. Tento způsob ovšem neřeší předání *window* objektu do těla PL/pgSQL funkce, pouze její uložení po dobu provádění funkce.

Za správnou cestu se ukázalo použití automatické proměnné, která obsaženou informaci uchovává ve stavové proměnné volání funkce. Automatická proměnná je dostupná z těla PL/pgSQL, proto se v tomto případě hodí na předání *window* objektu. V kompilátoru se tato proměnná vytváří. Inspirace již existující automatickou proměnnou `found` usnadnila vytvoření vlastní varianty. Nutnost specifikování datatypu se nejprve vyřešila použitím `INT8OID` (odpovídající datovému typu `Integer` o velikosti 8 *byte*, do kterého adresa

ukazatele na objekt lze uložit) Později bylo vhodné z důvodu profesionality vytvořit vlastní datový typ pojmenovaný `WINDOWOBJECTOID`.

```
var = plpgsql_build_variable("winobj", 1,
    plpgsql_build_datatype(windowOBJECTOID,-1,InvalidOid,NULL),true);
var->dtype = PLPGSQL_DTYPE_PROMISE;
((PLpgSQL_var *) var)->promise = PLPGSQL_PROMISE_WO;
```

Výpis kódu 4.2: Vytvoření automatické proměnné *winobj*, která obsahovala *window* objekt

Vlastní datový typ se definuje v souboru *pg_type.dat*, který slouží jako zdroj hodnot pro generování SQL skriptu pro systémovou tabulku *pg_type*. Tento datatyp byl pro interní použití pojmenován jako `window_object`. Každý datatyp nutně potřebuje vstupní a výstupní soubor, aby ho bylo možné zaregistrovat. Tyto soubory se starají o to, jak se textový řetězec na vstupu přeloží na nově vzniklý datatyp v případě vstupního souboru `„datatyp_in"`, a při obráceném postupu v případě výstupního souboru `„datatyp_out"`. Uvedené soubory bylo nutné vytvořit, ale jelikož jsou pro účely práce zbytečné, jsou nastaveny na vrácení předdefinovaných údajů (při práci není potřeba zobrazovat textovou hodnotu tohoto datatypu).

```
{ oid => '270', oid_symbol => 'windowOBJECTOID', descr => 'Datovy
typ slouzici pro praci s windowObject', typname => 'window_object',
typlen => 'SIZEOF_POINTER', typbyval => 't', typcategory => 'P',
typinput => 'window_object_in', typoutput => 'window_object_out',
typreceive => '-', typsend => '-', typalign => 'i' }
```

Výpis kódu 4.3: Registrace datového typu pro práci s *window* objektem

Po vzoru *Triggeru* byla nastavena proměnná na vlastní typ *promise*. *Promise* je objekt spíše známý například z JavaScript, zde má však jiný význam. Redukuje se režie inicializace automatických proměnných, která by zpomalila start funkce, neboť inicializace proběhne až při prvním použití v kódu [33].

Když byl vytvořen způsob, jak předat objekt do těla funkce, bylo nutné při průchodu vlastní verzí `plpgsql_exec_function` někam do hlavní struktury `PLpgSQL_function` (struktura funkce v PL/pgSQL obsahující informace o parametrech funkce či návratovém typu) umístit vlastní proměnnou, která bude na omezenou dobu uchovávat `WindowObject`. Do vytvořené automatické proměnné se dosadí *window* objekt v interpretu PL/pgSQL ve funkci `plpgsql_fulfill_promise`. Ve funkci je odchycena varianta *promise* a do automatické proměnné se dosadí pomocí `assign_simple_var` ukazatel uložený v dočasné struktuře.

```
static void plpgsql_fulfill_promise(PLpgSQL_execstate *estate,
    PLpgSQL_var *var)
{
    MemoryContext oldcontext;
    if (var->promise == PLPGSQL_PROMISE_NONE)
        return;
    oldcontext = MemoryContextSwitchTo(estate->datum_context);
    switch (var->promise)
    {
        case PLPGSQL_PROMISE_WO:
            if (estate->winobj == NULL)
                assign_simple_var(estate, var,
                    PointerGetDatum(estate->winobj), false, false);
                break;
    ...
```

Výpis kódu 4.4: Dosazení do automatické proměnné

4.3 Obálkové funkce

Díky možnosti předání *window* objektu do těla PL/pgSQL funkce zbývalo pouze vytvořit funkce, které budou s novým datovým typem pracovat. Pro každou funkci ze C API byla vytvořena její pomocná funkce, která na vstupu přijímá spolu s ostatními argumenty *window* objekt, zavolá stávající funkci a vrátí výsledek této funkce. Každé z těchto „obálek“ bylo nutné doplnit zápis do *pg_proc.dat*, kde se funkce registruje, aby ji bylo možné zavolat za běhu systému. Nepoužité `OID`, symbolizující unikátní identifikátor funkce, lze získat pomocí skriptu *unused_oids* nacházejícím se v adresáři *src/include/catalog/*. Kromě `OID` se zde zadávají datové typy argumentů funkce a návratového typu, oficiální název funkce viditelný „zvenku“ či zdrojová funkce, kterou tento zápis reprezentuje. Jakmile se potřebné funkce zaregistrovaly, bylo je možné použít v nové extenzi, kde se prakticky vyzkoušelo, zda výsledná funkce vrací stejný výsledek, jako originál z *windowfuncs.c*.

Při psaní pomocných funkcí se objevil problém. Díky variabilitě struktury, která se používá ve funkci `WinGetPartitionLocalMemory` (získání či uložení obsahu do vnitřní paměti), nebylo nalezeno řešení, které by bylo univerzální. Dočasně vyřešit tento problém lze přidáním definice požadované struktury v souboru, kde je nové rozhraní uložené, a poupravit funkce pracující s uložištěm, aby se na základě podmínky v rozhodovací konstrukci vybral konkrétní případ. Pro účely testování byla vytvořena struktura obsahující 8 byte `float`.

Na výpisu kódu 4.5 je vidět postup při tvorbě rozhraní. Díky proměnné `winobj` bylo možné přistupovat k datům *window* funkce. Nejprve se testovalo předání proměnné `winobj` z těla PL/pgSQL funkce do funkce v jazyce C. Ve druhé funkci se již funkcionalita částečně přesunula z C funkce do PL/pgSQL funkce.

```
CREATE FUNCTION my_rownum()
RETURNS int
AS $$
BEGIN
    RETURN my_window_row_number(winobj);
END;
$$ LANGUAGE plpgsql WINDOW;

CREATE FUNCTION my_rownum2()
RETURNS int
AS $$
DECLARE
    curpos int8;
BEGIN
    IF window_object_is_valid_plpgsql(winobj) THEN
        curpos := win_get_current_position_plpgsql(winobj);
        PERFORM win_set_mark_position_plpgsql(winobj, curpos);
        RETURN curpos + 1;
    ELSE
        RETURN -1;
    END IF;
END;
$$ LANGUAGE plpgsql WINDOW;
```

Výpis kódu 4.5: Postup při vytváření *window* rozhraní pro PL/pgSQL

4.4 PL/pgSQL window funkce

Vlastních *window* funkcí, využívajících C API, existuje velmi málo. Jediný externí dohledatelný příklad se jmenuje `pg_median_utils`. S inspirací u definovaných funkcí z *windowfuncs.c* a tohoto příkladu byla vytvořena vlastní *window* funkce v PL/pgSQL. Aby se výsledek nepočítal znovu, pokud již byl vypočítán předchozím řádkem, zjistí se, zda uložiště pro *window* funkci neobsahuje již uložený výsledek. Pokud obsahuje, do výstupní proměnné se uloží vypočítaná hodnota a funkce se ukončí. Jedná-li se o první řádek v *partition*, bude nutné hodnotu vypočítat. Do proměnné se uloží počet řádků v rámci aktuální *partition*. Poté se v cyklu přistupuje ke všem řádkům v rámci *partition* a získává se hodnota z funkčního argumentu. Jednotlivé hodnoty se připočítávají k mezivýsledku. Po skončení cyklu se součet vydělí počtem řádků, hodnota se uloží pro následné použití a vrátí se výsledek [34].

Ve výpisu kódu 4.6 je představena *window* funkce napsaná v PL/pgSQL využívající nové *window* rozhraní. Nejprve se zjistí, zda v paměti není uložena již vypočítaná hodnota. V negativním případě se iteruje přes řádky v aktuální *partition* a získává se hodnota specifikovaného argumentu. Tyto hodnoty se sečtou a vydělí počtem řádků z *partition*. Výsledek se uloží pro případné další využití. Všechny funkce ve tvaru `"win_"` jsou z nově vzniklého rozhraní.

```
CREATE FUNCTION my_window_avg(int)
RETURNS float8
AS $$
DECLARE
    partition_count int := 0;
    cnt int := 0;
    sum float8 := 0;
    result float8 := 0;
    precalculated bool := false;
BEGIN
    precalculated := win_is_context_in_local_memory(winobj);
    IF precalculated THEN
        result := win_get_partition_local_memory(winobj);
    ELSE
        partition_count := win_get_partition_row_count(winobj);
        FOR cnt IN 0 .. partition_count - 1
        LOOP
            sum := sum + win_get_func_arg_in_partition(winobj,
                0, cnt, 1, false);
        END LOOP;
        result := sum / partition_count::float8;
        PERFORM win_set_partition_local_memory(winobj, result);
    END IF;
    RETURN result;
END
$$ LANGUAGE plpgsql WINDOW;
```

Výpis kódu 4.6: *window* funkce v PL/pgSQL počítající průměr

```
postgres=# select * from testTable;
 value | dep
-------+-----
     1 | A
     2 | B
     3 | B
     5 | C
     8 | C
     9 | C
    12 | A
    14 | B
(8 rows)

postgres=# SELECT dep, value, my_window_avg(value) OVER (PARTITION BY dep) FROM testTable;
 dep | value |   my_window_avg
-----+-------+-------------------
 A   |     1 |               6.5
 A   |    12 |               6.5
 B   |     3 | 6.333333333333333
 B   |    14 | 6.333333333333333
 B   |     2 | 6.333333333333333
 C   |     9 | 7.333333333333333
 C   |     8 | 7.333333333333333
 C   |     5 | 7.333333333333333
(8 rows)
```

Obrázek 4.1: Výstup funkce *my_window_avg* pro malou množinu dat

Kapitola 5

Testování

Ve chvíli, kdy bylo možné vytvořit *window* funkci v PL/pgSQL, nastal čas otestovat, zda je výstup funkce napsané v PL/pgSQL roven ekvivalentní funkci využívající C API. Po testu funkčnosti následoval test rychlosti zpracování dotazu, kde se spolu porovnaly všechny možnosti, jak v systému PostgreSQL vytvořit *window* funkci. Je důležité zmínit, že bylo dodrženo stejných podmínek při testování jednotlivých variant (virtualizované Ubuntu 64-bit s přidělenou pamětí 8192 MB a 2 CPU z procesoru).

5.1 Porovnání funkčnosti

Před testováním rychlosti zpracování dotazu bylo potřeba ověřit, že bude vlastní *window* funkce v PL/pgSQL vracet správný výsledek. Testovací tabulka obsahovala dva sloupce. První sloupec, pojmenovaný `value` s datovým typem `Integer`, obsahoval hodnotu, se kterou se v testovacích funkcích počítalo. Druhý sloupec `dep`, o datovém typu `text`, figuroval jako rozdělovač záznamů do jednotlivých *partition* pomocí příkazu `PARTITION BY`. Testy měly za cíl zkontrolovat, zda se neliší výsledek u funkce `firs_value` a u jejího PL/pgSQL funkčního ekvivalentu `my_window_first_value`. Nejprve se provedl test s prázdnou testovací tabulkou, následovaný testem NULL hodnot. Testování náhodnými daty zajistily tři skripty z *online* generátoru dat [35].

Funkce `my_window_first_value` má stejnou funkcionalitu jako funkce `first_value` ze souboru *windowfunc.c*. Pseudotyp `anyelement` v argumentu funkce značí skutečnost, že datový typ není přímo definovaný, ale funkce dokáže pracovat s jakýmkoliv typem. Příkaz `ALIAS FOR \$0` znamená, že proměnná `result` je datového typu jako pseudotyp z argumentu. Při běhu se datový typ automaticky přiřadí. Do proměnné `result` se dosadí hodnota prvního funkčního argumentu z prvního řádku ve *frame*. Obsah proměnné `result` se odešle jako návratová hodnota [36].

```
CREATE FUNCTION my_window_first_value(anyelement)
RETURNS anyelement
AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := win_get_func_arg_in_frame(winobj, 0, 0, 1, true);
    return result;
END
$$ LANGUAGE plpgsql WINDOW;
```

Výpis kódu 5.1: *window* funkce určená pro testování stejného výstupu

Pomocí dotazu (výpis kódu 5.2) vypsaného níže se ověřovalo, zda testy náhodných hodnot testovaná funkce splnila. Při použití stejné *window* u několika *window* funkcí lze zkrátit dotaz pomocí definice *window* na konci. SQL klauzule `CASE` vytváří rozhodovací konstrukci, kde podmínka je daná po klauzuli `WHEN`. Celkově dotaz vrací počet řádků, ve kterých se porovnané funkce nerovnají. U všech tří testů náhodných dat dotaz vrátil číslo nula.

```
SELECT COUNT(*) FROM (
    SELECT CASE WHEN x.native = x.mine THEN '1' ELSE '0' END AS result
        FROM (
                SELECT first_value(value) OVER W AS native,
                my_window_first_value(value) OVER W AS mine FROM
                uncertainTable window W AS (PARTITION BY dep
                ORDER BY value DESC)
        ) x
 ) y WHERE y.result = '0';
```

Výpis kódu 5.2: SQL dotaz ověřující, zda se výstupy funkcí shodují

5.2 Porovnání rychlosti zpracování SQL dotazu

Pro účely měření bylo potřeba pracovat s většími daty. Pro svou jednoduchost byl použit *online* generátor záznamů. Vzorová tabulka obsahovala dva sloupce. První sloupec, pojmenovaný `value` s datovým typem `Integer`, obsahoval hodnotu, se kterou se v testovacích funkcích počítalo. Druhý sloupec `dep`, o datovém typu `text`, figuroval jako rozdělovač záznamů do jednotlivých *partition* pomocí příkazu `PARTITION BY`. Do tabulky pojmenované `testTable` byl vložen pomocí příkazu `psql -f "názevSQLskriptu.sql"` nagenerovaný počet řádků. Testování mezi sebou porovnávalo čtyři soupeře ve třech různých kategoriích (tisíc, sto tisíc a milion záznamů v tabulce) při deseti pokusech v kategorii, kdy se následně zprůměroval konečný výsledek.

5.2.1 Funkce využívající rozhraní napsané v jazyce C

Při seznamování s agregačními funkcemi bylo žádoucí si zkusit vytvořit vlastní agregaci pojmenovanou `myAVG`. Této skutečnosti se využilo v testování, jelikož se kromě již existující funkce `avg`, použila i vlastní agregace, která je funkčně ekvivalentní existující verzi. Díky tomu, že každou agregační funkci lze pomocí příkazu `OVER` použít jako *window* funkci, bylo možné zařadit tyto dvě funkce do testování.

Testovaná funkce - definovaná agregační funkce avg (C API)			
Test	Řádků 1.000 [ms]	Řádků 100.000 [ms]	Řádků 1.000.000 [ms]
1	2,5	180	1 915
2	1,7	177	1.998
3	1,7	170	1.992
4	1,7	164	1.913
5	1,6	177	1.848
6	1,7	171	1.928
7	1,7	176	1.926
8	1,7	167	1.836
9	1,6	176	1.894
10	1,7	170	1.997

Tabulka 5.1: Výsledky testování - definovaná agregační funkce (C API)

Zatímco ukázka vytvoření agregační funkce využívala PL/pgSQL, testovaná funkce je napsaná v jazyce C, aby mohla využít C API. Složení je nicméně podobné. Akumulátor si nashromáždí hodnoty, konkrétně do pole typu `float8` ukládá mezisoučet a celkový počet záznamů. Finální funkce získá z pole hodnoty, vydělí je a vrátí výsledek.

```
CREATE FUNCTION accumulator(anyarray, float8)
RETURNS anyarray
AS 'MODULE_PATHNAME'
LANGUAGE C IMMUTABLE STRICT PARALLEL SAFE;
```

Výpis kódu 5.3: Deklarace akumulátoru v extenzi

```
Datum accumulator(PG_FUNCTION_ARGS)
{
ArrayType   *transarray = PG_GETARG_ARRAYTYPE_P(0);
float8  newval = PG_GETARG_FLOAT8(1);
float8  *transvalues;
float8  counter, total;

if (ARR_NDIM(transarray) != 1 || ARR_DIMS(transarray)[0] != 2 ||
ARR_HASNULL(transarray) || ARR_ELEMTYPE(transarray) != FLOAT8OID)
    elog(ERROR, "accumulator: expected 2-element float8 array");
transvalues = (float8 *) ARR_DATA_PTR(transarray);
counter = transvalues[0];
total = transvalues[1];

counter += 1.0;
total += newval;
transvalues[0] = counter;
transvalues[1] = total;
PG_RETURN_ARRAYTYPE_P(transarray);
}
```

Výpis kódu 5.4: Definice akumulátoru pro agregační funkci určenou k testování rychlosti zpracování SQL dotazu

Testovaná funkce - vlastní agregační funkce myAVG (C API)			
Test	Řádků 1.000 [ms]	Řádků 100.000 [ms]	Řádků 1.000.000 [ms]
1	2,0	184	2.039
2	1,7	183	2.016
3	1,7	181	1.948
4	1,7	173	1.952
5	1,9	177	1.953
6	1,8	169	1. 955
7	1,7	168	1.952
8	1,7	168	1.874
9	1,8	173	1.953
10	1,7	184	1.946

Tabulka 5.2: Výsledky testování - vlastní agregační funkce (C API)

5.2.2 Funkce využívající PL/v8 rozhraní

Při vytváření *window* funkce využívající rozhraní napsané v procedurálním jazyce PL/v8 nebylo k dispozici taktéž málo vzorových příkladů. S pomocí ukázkových příkladů definovaných ve zdrojových kódech procedurálního jazyka PL/v8 (*plv8/sql/window.sql*) byla zhotovena *window* funkce, která počítá průměr. Pracuje na stejném principu jako vlastní *window* funkce v novém PL/pgSQL rozhraní. Nejprve rozpozná, zda je nutné počítat výsledek, či zda již byl vypočítán. V případě absence výsledku se získá počet řádků v aktuální *partition*, sečtou se postupně hodnoty uložené ve funkčních argumentech a po vydělení se získá výsledek, který je následně uložen pro následné použití.

```
CREATE FUNCTION my_plv8_avg(value numeric)
RETURNS float AS $$
var winobj = plv8.get_window_object();
var context = winobj.get_partition_local() || {};
var tmp_sum = 0;
if (!context.sum) {
    var partition_count = winobj.get_partition_row_count();
    for (var i = 0; i < partition_count; i++) {
        tmp_sum += winobj.get_func_arg_in_partition(0, i,
            winobj.SEEK_HEAD, false);
    }
    context.sum = tmp_sum / partition_count;
    winobj.set_partition_local(context);
}
return context.sum;
$$ LANGUAGE plv8 window;
```

Výpis kódu 5.5: *Window* funkce využívající PL/v8 rozhraní

Testovaná funkce - vlastní window funkce (PL/v8 rozhraní)			
Test	Řádků 1.000 [ms]	Řádků 100.000 [ms]	Řádků 1.000.000 [ms]
1	10,1	1.121	10.094
2	11,4	1.108	10.117
3	11,2	1.085	10.065
4	9,7	1.107	9.985
5	9,3	1.089	9.954
6	9,2	1.092	9.908
7	9,0	1.095	9.948
8	9,1	1.082	9.980
9	9,8	1.085	9.920
10	9,6	1.079	9.937

Tabulka 5.3: Výsledky testování - vlastní window funkce (PL/v8)

5.2.3 Funkce využívající PL/pgSQL rozhraní

Jako poslední se testoval „vyzyvatel“ v podobě funkce využívající PL/pgSQL rozhraní. Aby byl splněn druhý cíl praktické části práce, bylo nutné dosáhnout minimálně stejných časů, jako u *window* funkce využívající PLV8 rozhraní. Definici použité *window* funkce je možné vidět v podkapitole PL/pgSQL *window* funkce.

Testovaná funkce - vlastní window funkce (PL/pgSQL rozhraní)			
Test	Řádků 1.000 [ms]	Řádků 100.000 [ms]	Řádků 1.000.000 [ms]
1	5,7	430	4.545
2	4,2	410	4.642
3	4,3	418	4.609
4	4,2	415	4.546
5	4,3	410	4.544
6	4,2	420	4.514
7	4,3	416	4.499
8	4,6	408	4.530
9	4,2	410	4.620
10	4,3	420	4.498

Tabulka 5.4: Výsledky testování - vlastní window funkce (PL/pgSQL)

5.2.4 Shrnutí výsledků

Výsledky testu splnily očekávání. Nejrychlejší byla vestavěná agregační funkce, kterou těsně stíhala vlastní agregační funkce. Nově vytvořené API bylo přibližně dvakrát pomalejší než vestavěná agregační funkce, ale drtivě předstihlo již existující PLV8 analytickou funkci, která zaostala o další více než dvojnásobný čas.

Průměrné hodnoty			
Technologie	Řádků 1.000	Řádků 100.000	Řádků 1.000.000
C API	1,8 ms	173 ms	1.925 ms
C API 2	1,8 ms	176 ms	1.959 ms
PL/v8	9,7 ms	1.094 ms	9.991 ms
PL/pgSQL	4,4 ms	416 ms	4.555 ms

Tabulka 5.5: Celkové výsledky testování - průměrné hodnoty

Závěr a zhodnocení výsledků

Pro naplnění prvního cíle praktické části bylo nutné nastudovat vnitřní strukturu a identifikovat místa, jejichž modifikace zpřístupnila zpracování *window* funkcí. Modifikování interpretu PL/pgSQL umožnilo předat odkaz na *window* objekt funkci volané z těla PL/pgSQL funkce. Interním funkcím, které byly zpřístupněny pro volání z SQL prostředí, bylo poté možno předat zmíněný objekt. Po této skutečnosti bylo možné provádět funkce přistupující k vnitřní paměti *window* funkce či vracející pozici v rámci partition.

Až na výjimky lze vytvořit jakoukoliv *window* funkci z předpřipravených funkcí a při doplnění drobných funkcionalit do důležitého souboru by mělo být možné splnit jakýkoliv požadavek potencionálního zákazníka, který po vývojáři bude požadovat vlastní analytickou funkci využívající PL/pgSQL. Existence vlastní funkce, jejíž funkčnost zajišťuje nově vytvořené rozhraní, podporuje splnění prvního vytyčeného cíle praktické části. Jediný větší nedostatek se nachází v tom, že pokud je nutné používat ve funkci specifickou strukturu, musí být deklarace této struktury doplněna do souboru, kde se nacházejí ostatní funkce dohromady tvořící rozhraní. Pro účely práce byl jeden druh struktury vytvořen, ale obecné řešení nebylo nalezeno. Po splnění, až na drobné výhrady, prvního praktického cíle se mohlo začít s testováním. V sérii testů bylo ověřeno, že *window* funkce vytvořená v PL/pgSQL má stejné výstupy jako předdefinovaná funkce. Při testu rychlosti zpracování požadavku v testovací tabulce dosáhla vytvořená vlastní *window* funkce vytoužených výsledků. V porovnání s funkcí vytvořenou v jiném jazyce (PLV8), jehož implementace *window* funkcí se považuje za jedinou další oficiálně podporovanou možnost kromě tradičního řešení, se dosáhlo dokonce více než dvakrát rychlejšího průměrného času ve všech kategoriích testování.

Vzhledem k funkčnímu řešení splňující základní potřeby při tvorbě analytické funkce a příznivému času dosaženého v testech nezbývá než konstatovat, že nové rozhraní má smysl realizovat a v praxi je použitelné.

Literatura

[1] VERTICA. *Analytic Functions Versus Aggregate Functions*. Vertica.com [online]. [cit. 2020-03-01]. Dostupné z: https://www.vertica.com/docs/9.2.x/HTML/Content/Authoring/AnalyzingData/SQLAnalytics/AnalyticFunctionsVersusAggregateFunctions.htm.

[2] RAHUL, Batra. *SQL Primer: An Accelerated Introduction to SQL Basics*. Berkeley: Apress, ©2018. ISBN 978-1-4842-3575-1. Dostupné také z: https://link-springer-com.ezproxy.techlib.cz/book/10.1007%2F978-1-4842-3576-8.

[3] CHRISTENSEN, Jack. *Custom Aggregates in PostgreSQL*. In: Blogger [online]. 2016-02-15 [cit. 2020-04-06]. Dostupné z: https://hashrocket.com/blog/posts/custom-aggregates-in-postgresql.

[4] Postgres OnLine Journal. *Dollar-quoting for escaping single quotes*. In: Blogger [online]. 2017-06-17 [cit. 2020-04-17]. Dostupné z: https://www.postgresonline.com/journal/archives/376-Dollar-quoting-for-escaping-single-quotes.html.

[5] The PostgreSQL Global Development Group. *Zdrojové kódy databázového systému PostgreSQL*. Verze: 20.04. [online] © 1996-2020. [Verze aktuální k datu: 2020-02-07]. Dostupné z: https://github.com/postgres/postgres/

[6] BISSO, Ignacio L. *SQL Window Function Example With Explanations*. [online]. 2017-08-4 [cit. 2020-04-27]. Dostupné z: https://learnsql.com/blog/sql-window-functions-examples/.

[7] WENZEL, Kris. *SQL Window Functions [Visual Explanation]*. EssentialSQL. In: Blogger [online]. [cit. 2020-01-05]. Dostupné z: https://www.essentialsql.com/sql-window-functions/.

[8] The PostgreSQL Global Development Group. *window Functions*. Postgresql.org [online]. © 1996-2020 [cit. 2020-04-18]. Dostupné z: https://www.postgresql.org/docs/9.0/tutorial-window.html.

[9] LASKOWSKI, Jacek. *Standard Aggregate Functions*. GitBook [online]. [cit. 2020-04-15]. Dostupné z: https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-aggregate-functions.html

[10] LASKOWSKI, Jacek. *Standard Functions for Window Aggregation*. GitBook [online]. [cit. 2020-04-15]. Dostupné z: https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-functions-windows.html

[11] PostgreSQL Tutorial. *What is PostgreSQL?* [online]. ©2020. [cit. 2020-04-05]. Dostupné z: https://www.postgresqltutorial.com/what-is-postgresql/.

[12] Open Source Initiative. *The MIT License*. [online]. [cit. 2020-05-30]. Dostupné z: https://opensource.org/licenses/mit-license.php.

[13] STĚHULE, Pavel. *C a PostgreSQL - interní mechanismy*. [online] 2011 [cit. 2020-04-20] Dostupné z: https://postgres.cz/wiki/C_a_PostgreSQL_-_intern%C3%AD_mechanismy.

[14] DAR, U., H. Krosing, J. Mlodgenski a K. Roybal. *PostgreSQL Server Programming - Second Edition*. ©2015. ISBN 978-1-7839-8058-1. Dostupné také z: https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781783980581

[15] severalnines. *Creating New Modules using PostgreSQL Create Extension*. In: Blogger [online]. 2018-08-30 [cit. 2020-04-22] Dostupné z: https://severalnines.com/database-blog/creating-new-modules-using-postgresql-create-extension.

[16] The PostgreSQL Global Development Group. *CREATE FUNCTION*. Postgresql.org [online]. © 1996-2020 [cit. 2020-04-18]. Dostupné z: https://www.postgresql.org/docs/9.6/sql-createfunction.html.

[17] The PostgreSQL Global Development Group. *Extension Files*. Postgresql.org [online]. © 1996-2020 [cit. 2020-04-20]. Dostupné z: https://www.postgresql.org/docs/current/extend-extensions.html.

[18] The PostgreSQL Global Development Group. *C-Language Functions*. Postgresql.org [online]. © 1996-2020 [cit. 2020-04-20]. Dostupné z: https://www.postgresql.org/docs/8.2/xfunc-c.html.

[19] STĚHULE, Pavel. *Re: Rozpracovaná bakalářská práce – Pavel Špecht* [elektronická pošta]. Zasláno z: pavel.stehule@gmail.com. 2020-05-30. [cit. 2020-05-30].

[20] The PostgreSQL Global Development Group. *psql*. Postgresql.org [online]. © 1996-2020 [cit. 2020-04-20]. Dostupné z: `https://www.postgresql.org/docs/9.2/app-psql.html`.

[21] The PostgreSQL Global Development Group. *Executor*. Postgresql.org [online]. © 1996-2020 [cit. 2020-04-07]. Dostupné z: `https://www.postgresql.org/docs/8.2/executor.html`.

[22] The PostgreSQL Global Development Group. *Overview*. Postgresql.org [online]. © 1996-2020 [cit. 2020-04-18]. Dostupné z: `https://www.postgresql.org/docs/9.6/plpgsql-overview.html`.

[23] The PostgreSQL Global Development Group. *Procedural Languages*. Postgresql.org [online]. © 1996-2020 [cit. 2020-04-25]. Dostupné z: `https://www.postgresql.org/docs/9.1/xplang.html`

[24] STĚHULE, Pavel. *PL/pgSQL*. [online] [cit. 2020-04-22] Dostupné z: `https://postgres.cz/wiki/PL/pgSQL#P.C5.99edstaven.C3.AD_jazyka_PL.2FpgSQL`.

[25] BITNINE GLOBAL INC. *PL/pgSQL Internals*. In: Blogger [online]. 2016-06-14 [cit. 2020-04-07]. Dostupné z: `https://bitnine.net/blog-postgresql/plpgsql-internals/`.

[26] *How to create a custom windowing function for PostgreSQL*. Stackoverflow.com [online]. 2012-12-09. [cit. 2020-04-17]. Dostupné z: `https://stackoverflow.com/questions/13790028/how-to-create-a-custom-windowing-function-for-postgresql-running-average-examp`

[27] PGCon. Introducing windowing Functions. *Pgcon.org*. [online]. 2009 [cit. 2020-04-07]. Dostupné z: `https://www.pgcon.org/2009/schedule/attachments/98_windowing%20Functions.pdf`.

[28] *PLV8*. [online]. [cit. 2020-04-07]. Dostupné z: `https://plv8.github.io/#window-function-api`.

[29] ORACLE CORPORTATION. *Oracle VM VirtualBox 6.0.16*. [software]. Dostupné z: `https://www.virtualbox.org/`. Minimální požadavky na systém: x86 hardware, 512MB RAM, 30MB místa na disku, podporuje většinu operačních systémů.

[30] Canonical Ltd. *Ubuntu image 20.04 LTS*. [software]. Dostupné z: `https://ubuntu.com/download/desktop`. Minimální požadavky na systém: 5GB místa na disku.

[31] The PostgreSQL Global Development Group. *PostgreSQL 20.04*. [software]. Dostupné například z: `https://www.postgresql.org/download/`. Minimální požadavky na systém: 64bit CPU, 64bit operační systém, 2GB místa na disku, dvě jádra CPU, RAID 1.

[32] SIEVERT, Jerry. *PL/v8 2.3.10*. [software]. Dostupné z: `https://github.com/plv8/plv8`. Požadavky na běh pro linuxové distribuce: Git, g++/clang++, Python, pkg-config, libc++dev, libc++abi-dev.

[33] STĚHULE, Pavel. *Re: Rozpracovaná bakalářská práce – Pavel Špecht* [elektronická pošta]. Zasláno z: pavel.stehule@gmail.com. 2020-05-15. [cit. 2020-05-15].

[34] GRAY, Jonathan. *Pg_median_utils*. GreenApe. [online] [cit. 2020-04-20] Dostupné z: `https://github.com/greenape/pg_median_utils/blob/master/src/median_filter.c`.

[35] WHITEBOARD technologies. *Online test data generator*. onlinedatagenerator.com [online]. ©2020 [cit. 2020-03-15]. Dostupné z: `https://www.onlinedatagenerator.com/`

[36] The PostgreSQL Global Development Group. *Declarations*. Postgresql.org [online]. © 1996-2020 [cit. 2020-04-18]. Dostupné z: `https://www.postgresql.org/docs/9.4/plpgsql-declarations.html`.

Příloha **A**

Seznam zkratek a pojmů

SQL
: Structured Query Language neboli strukturovaný dotazovací jazyk

PostgreSQL
: objektově-relační databázový systém

PL/pgSQL
: Procedural Language/PostgreSQL (jedná se o procedurální jazyk systému PostgreSQL)

PL/v8
: procedurální JavaScript jazyk systému PostgreSQL

Partition
: skupina řádků ve *window* funkci určená pomocí klauzule `PARTITION BY`

Frame
: skupina řádků v *partition* určená pomocí klauzule `ORDER BY`

Datum
: abstraktní datový typ v systému PostgreSQL

API
: Application Programming Interface

C API
: skupina maker a funkcí zajišťující funkčnost *window* funkcí v jazyce C v systému PostgreSQL

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... Popis obsahu média
- `instalace.txt` ..... Seznam instrukcí pro instalaci
- `patch.zip` ..... Vybrané pozměněné soubory
- `postgres.zip` ..... Zdrojové soubory PostgreSQL včetně změn
- `prace.pdf` ..... Bakalářská práce ve formátu PDF
- `prace.zip` ..... Zdrojové kódy k bakalářské práci ve formátu TEX