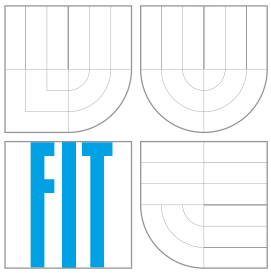
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DOPLNĚNÍ A OPTIMALIZACE TEMPORÁLNÍHO ROZŠÍŘENÍ PRO POSTGRESQL

COMPLETION AND OPTIMIZATION OF A TEMPORAL EXTENSION FOR POSTGRESQL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DOMINIKA KORONCZIOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

RNDr. MAREK RYCHLÝ, Ph.D.

BRNO 2016

Zadání diplomové práce/17908/2015/xkoron00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Koroncziová Dominika, Bc.**
Obor: Informační systémy
Téma: **Doplnění a optimalizace temporálního rozšíření pro PostgreSQL**
Completion and Optimization of a Temporal Extension for PostgreSQL
Kategorie: Databáze

Pokyny:

1. Seznamte se s PostgreSQL, možnostmi jeho rozšíření, s vlastnostmi temporálních dat, principy a implementacemi temporálních databází a s jazykem TSQL2.
2. Vyhodnoťte výsledky odkazované práce "Temporální rozšíření pro PostgreSQL". Soustřeďte se na složitost zadávání dotazů (oproti TSQL2) a na rychlost jejich provedení (prozkoumejte prováděcí plány).
3. Navrhněte řešení nedostatků zmiňované práce či rozšíření jejích výsledků, např. doplněním chybějící funkcionality (agregační funkce, odsávání, ref. integrita), zlepšením způsobu zadávání temp. dotazů či rychlosti jejich provádění. Můžete využít sys. sloupce (CTID), funkce vracející množiny (typ "record"), možnosti ovlivnit plánovač (FDW) atd.
4. Po konzultaci s vedoucím navržená rozšíření implementujte.
5. Výslednou implementaci důkladně otestujte, proveďte zhodnocení dosažených výsledků a diskutujte další možný vývoj projektu.

Literatura:

- Radek Jelínek. *Temporální rozšíření pro PostgreSQL*. Diplomová práce, FIT VUT v Brně, 2015. [http://www.fit.vutbr.cz/study/DP/DP.php?id=15982]
- Neil Conway. *Introduction to Hacking PostgreSQL*. 2007. [http://neilconway.org/talks/hacking/ottawa/ottawa_slides.pdf]
- Richard T. Snodgrass. Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann Publishers, Inc., San Francisco, July, 1999, 504+xxiii pages, ISBN 1-55860-436-7. [http://www.cs.arizona.edu/people/rts/tdbbook.pdf,http://www.cs.arizona.edu/people/rts/

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Táto práca sa zaoberá problematikou implementácie podpory práce s temporálnymi dátami v prostredí tradičného, relačného databázového systému PostgreSQL. Nadväzujem v nej na predchádzajúce výsledky Radka Jelínka a ním vyvinuté rozšírenie. Toto riešenie som analyzovala po stránkach funkčnosti, praktickosti a výkonu. Na základe týchto testov som navrhla výrazné zmeny v tomto rozšírení a tieto zmeny implementovala. Práca popisuje implementačné detaily ako aj výsledky výkonnostného zrovnania oproti pôvodnému rozšíreniu.

Abstract

This thesis focuses on a implemenation of a a temporal data support within traditional relational environment of PostgreSQL system. I pick up on Radek Jelínek's thesis and an extension developed by him. I've analyzed the extension from functional, practical and performance perspectives. Based on my results, I've designed and implemented changes to the original extension. The work also contains implementation details as well as performance comparison results between the new and the original extensions.

Kľúčové slová

Databázové systémy, temporálne databázy, relačné databázy, transakčný čas, čas platnosti, TSQL2, PostgreSQL, rozšírenie, výkonnosť

Keywords

Database systems, temporal databases, relational databases, transaction time, valid time, TSQL2, PostgreSQL, extension, performance

Citácia

KORONCZIOVÁ, Dominika. *Doplnění a optimalizace temporálního rozšíření pro PostgreSQL*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Rychlý Marek.

Doplnění a optimalizace temporálního rozšíření pro PostgreSQL

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracovala samostatne pod vedením pána RNDr. Mareka Rychlého Ph.D. Uviedla som všetky literárne pramene a publikácie, z ktorých som čerpala.

.....
Dominika Koroncziová
23. mája 2016

Poďakovanie

Týmto by som rada poďakovala RNDr. Marekovi Rychlému Ph.D. za nápady, rady a ochotu pomáhať mi počas tvorby celej diplomovej práce.

Obsah

Kapitola 1

Úvod

Čas vo svojej podstate je všade prítomný jav s ktorým sa stretáva každý z nás. Každá udalosť s ktorou sa stretneme sa odohráva v istý časový okamih. Ľudia si už v dávnych dobách začali uvedomovať pojem času a jeho plynutia. V dnešnej dobe už iba nemálo aplikácii nevyžaduje zaznamenávanie a meranie času.

V oblasti počítačovej vedy si ľudia v osemdesiatych rokoch začali uvedomovať že ukladanie dát je úzko späté s časom a chceli mať možnosť pristupovať rovnako ako k dátam súčasným tak aj k tým minulým či budúcim. Vo svojej podstate nám ale relačný databázový model umožňuje pohľad na dáta iba v jeden konkrétny časový okamih. Začali sa teda objavovať problémy späté s ukladaním a reprezentáciou temporálnych dát, teda dátami, ktoré sa môžu meniť v čase a tieto zmeny sú reflektované v databázovom modeli.

Každá zmena v databáze ako pridanie nového záznamu, vymazanie stávajúceho či úprava existujúceho zmení stav databázy a nie je možné zistiť stav databázi pred touto zmenou. V priebehu uplynulých rokov sa ľudia snažia vymyslieť riešenie ako s takýmito dátami naložiť a správne s nimi pracovať. Prvý krát s riešením tejto situácie prišiel Richard Snodgrass v roku 1992, ktorý inicioval prácu na prvom formálnom riešení tohto problému. Išlo o špecifikáciu jazyka TSQL2 [14], ktorej primárnym cieľom je možnosť podpory bitemporálnych dát v databázových systémoch. Neskôr koncom devedesiatych rokov bola snaha dostať niektoré časti špecifikácie TSQL2 do SQL:1999 štandardu, ktorý je známy aj pod menom SQL3. Nakoniec sa úspešne podarilo niektoré časti špecifikácie do tohto štandartu dostať. Ľudia si ďalej tým viac uvedomovali dôležitosť temporálnych dát a tak sa v roku 2011 dostala do štandardu SQL:2011 špecifikácia, ktorá hovorí o práci s tabuľkami obsahujúcimi čas platnosti, transakčný čas, či oba tieto časy zároveň [12]. V dnešnej dobe existuje niekoľko implementácii, ktoré sa snažia pokryť prácu s temporálnymi dátami. Microsoft predstavil prácu s temporálnymi tabuľkami ako jednu z nových funkcií SQL Serveru 2016. IBM DB2 databázový server vo verzii 10 pridáva funkčnosť, ktorá je založená na norme SQL:2011.

V tejto práci sa budem venovať možnostiam podpory temporálnych dát v jazyku PostgreSQL. Prvá kapitola sa bude venovať problematike temporálnych databázových systémov, ich histórií, vývoju a súčasnému stavu. Druhá kapitola je venovaná exisujúcemu rozšíreniu jazyka PostgreSQL vyvynutému Radkom Jelínkom a zrovnaniu tohto rozšírenia s pôvodným návrhom jazyka TSQL2. Nasledujúca kapitola popisuje návrh novej funkcionality riešenia. Rozoberá koncepčné otázky a prístup k jednotlivým problémom. V štvrtej kapitole popisujem konkrétne špecifiká návrhu a implementačné detaily. Posledná kapitola textu je venovaná testovaniu môjho rozšírenia, tak po sránke funkčnosti, ako aj po stránke výkonu. Pre ľahšie nasadenie tu uvádzam príklady rôznych typov príkazov demonšrujúcich riešenie realistických problémov. Zrovnanie výkonnosti je prevedené formou zrovnania na konkrét-

nych scenároch medzi originálnym a mnou vyvinutým rozšírením. V závere práce okrem zhrnutia výsledkov rozoberám problematiku dalšieho rozvoja rozšírenia a smery ktorými by sa mohlo uberať.

Kapitola 2

Databázové systémy

2.1 História a vývoj databázových systémov

Prvým historickým mílnikom vo vývoji databázových systémov je rok 1890, kedy Herman Hollerith vyrobil prvý automatický prístroj, ktorý pracoval na báze dierných štítkov. Následne v roku 1911 bol spoluzakladateľom firmy IBM (International Business Machine), ktorá stála pri zrode databázových systémov. Americká vláda v roku 1935 vydala Social Security Act, ktorý hovoril o tom, že údaje o približne 26 miliónoch ľudí musia byť niekde uložené. V tento okamih prestala technológia diernych štítkov byt postačujúcou a firma IBM prišla s novým počítačom, ktorý niesol názov UNIVAC I. V šesťdesiatych rokoch minulého storočia vzniklo konzorcium CODASYL (Conference on Data Systems Languages), ktorého účelom bolo viesť vývoj programovacích jazykov, ktoré by mohli byť využívane na rôznych počítačových systémoch. Ich úsilím vznikol programovací jazyk, ktorý bol primárne určený na business účely [17]. V roku 1961 Charles Bachmann predstavil prvý návrh integrovaného dátového skladu a pripojil sa k zoskupeniu CODASYL. Neboli však jediní, ktorí sa zaoberali myšlienkami týkajúcimi sa správou informácii. V roku 1968 IBM prišlo so systémom na správu informácii, ktorý vytvárali počas projektu Apollo. Od roku 1949 pracoval pre firmu IBM Ted Codd, ktorý bol nespokojný ako s prístupom a riešním zoskupenia CODASYL tak aj s riešením firmy IBM. Preto v roku 1970 publikoval článok „A relational Model of Data for large shared Data Banks“, v ktorom navrhoval riešenie pre správu databáz vo forme relačného modelu, ktoré sa stalo teoretickým základom relačných databází [3]. V práci hovoril aj o tom, že by sa mal používať zrozumitelný a jednoduchý jazyk podobný hovorovej angličtine. Z počiatku samotné IBM pre ktoré v tom čase pracoval bralo jeho článok na ľahkú váhu, nakoľko narušovalo ich doterajšie riešenia. Ted Codd však svoje riešenie predstavil verejnosti, a vyvolal radu diskusií, po ktorých si aj samotné IBM uvedomilo dôležitosť tejto práce a začalo pracovať na produkte System-R. V roku 1979 implementovali už plne funkčný systém, ktorý využíval jazyk SEQUEL (Structured English Language) [2]. V roku 1976 popísali jazyk SEQUEL2, ktorý bol následne premenovaný na SQL a prijatý ako štandard.

Paralene s vývojom vo firme IBM vznikol aj projekt Ingres na University of California at Berkley, ktorý sa snažil o spracovávanie geografických dát [16]. Na tomto projekte nikdy nepracovalo viac ako 6 ľudí súčasne, čo viedlo k tomu že v roku 1982 projekt Ingres skončil. Nebolo to však nadlho. Po troch rokoch sa vrátil znova a to pod menom Postgres, ktorý viedol pôvodný zakladateľ projektu Ingres Michael Stonebraker. Hlavným cieľom bolo vytvoriť objektovo relačný systém. Na systéme sa pracovalo až do roku 1994 kedy projekt znova skončil. Znova to však nebolo nadlho. Dvaja študenti Jolly Chen a Andrew Yu z Uni-

versity of California, pokračovali na projekte v rámci štúdia a premenovali ho na Posgres95 [7]. V roku 1996 sa z neho stal open source projekt a bol premenovaný na PostgreSQL, pod ktorým ho poznáme až dodnes.

Z množstva ďalších databázových systémov stoja niekoľké za zmienku z dôvodu ich rozšírenosti. Jednou z najpopulárnejších databází, najmä v komerčnom prostredí, je OracleDB od spoločnosti Oracle. Jej vývoj začal verziou 1 už v roku 1978 a k dnešnému dňu je aktuálnou verzia 12. Ďalším rozšíreným databázovým systémom je MySQL. Jeho vývoj začal v roku 1994 a po dlhom nezávislom vývoji bol odkúpený firmou Sun Microsystems v roku 2008. Následne, v roku 2010 táto prešla pod kontrolu spoločnosti Oracle. Posledným, z výrazne rozšírených databázových systémov je Microsoft SQL Server. Ako názov naznačuje, jedná sa o riešenie poskytované spoločnosťou Microsoft a dostupné je od svojej prvej verzie z roku 1989.

2.2 Temporálne databázové systémy

Relačné datábazové systémy ako Oracle, SQL Server, MySQL umožňújú ukladanie veľkého množstva dát. Tieto dáta reprezentujú aktuálny stav systému a väčšinou ide o dáta, ktoré sú platné v daný okamih. Nie je teda možné získať dáta s ktorými sme pracovali v minulosti, či dáta s ktorými budeme pracovať v budúcnosti. Každá zmena spôsobí presun databázového systému z jedného stavu do iného. Stav databázového systému, pred zmenou však nie je možné za normálnych okolností, teda po úspešnom prebehnutí transakcie, vrátiť do stavu pôvodného. Tento problém si ľudia začali uvedomovať už v osemdesiatych rokoch, kedy sa začína pracovať s pojmami ako temporálne databázy či temporálne dáta.

2.2.1 Relačný dátový model

Relačný dátový model je jeden z najrozšírenejších dátových modelov, na ktorom je založených mnoho súčasných databázových systémov. Vyvíja sa od začiatku sedemdesiatych rokov počas ktorých bol prvý krát definovaný.

Relačné datábazy pracujú s reláciami, ktoré sa však od tých matematických líšia. Pracujú s pojmami ako doména, skalárna hodnota či zložená hodnota.

- **Doména** je pomenovaná množina skalárnych hodnôt toho istého typu. Doménu si môžeme predstaviť ako výčet hodnôt jedného konkrétneho typu. Ako príklad si môžeme uviesť napríklad doménu farieb, kde ju budú tvorit farby biela, modra a červená.
- **Skalárna** hodnota je najmenšia sémanticka jednotka dát, ktorá je atomická, teda ďalej nedeliteľná.
- **Zložená doména** je doména zložená z niekoľko jednoduchých domén.
- **Relácia na domenách** $D_1, D_2, ..., D_n$ je dvojica $\boldsymbol{R} = (R, R^*)$, kde $R = (A_1 : D_1, A_2 : D_2, ..., A_n : D_n)$ je schéma relácie a $R^* \subseteq D_1 \times D_2 \times ... \times D_n$ je telo relácie. Počet atribútov n relácie sa označuje ako stupeň relácie, kardinalita tela relácie $m = |R^*|$ sa označuje ako kardinalita relácie.

Relácia je teda abstraktný pojem relačného modelu. Tabuľka je forma zobrazenia relácie. Jednotlivé n-tice relácie si môžeme predstaviť ako riadky tabuľky a atributy relácie ako stĺpce tabuľky.

2.2.2 Temporálny dátový model

Pomocou relačného dátoveho modelu je možné zachytiť stav systému v konkrétny časový okamih. Toto je z hľadiska mnohých aplikácii nepostačujúce a je potrebné zachytiť postupne meniace sa stavy systému v čase.

Temporálny dátový model sa teda pokúša rozšíriť tradičný relačný dátový model. Ku každej n-tici relácie je uložená informácia o tom od kedy do kedy je platná, ako aj informácia o tom kedy sme danú n-ticu v systéme modifikovali. Tu sa stretávame s pojmami ako čas platnosti a transakčný čas.

- **Čas platnosti** je čas, ktorý reprezentuje kedy je daná n-tica platná v reálnom svete, skladá sa z dvoch rôznych časových okamihov, ktorými sú začiatok a koniec času platnosti. Tabuľky, ktoré obsahujú čas platnosti nazývame tabuľky s časom platnosti.

  Predstavme si ako príklad zbierku zákonov. Čas platnosti jednotlivých zákonov začína v okamihu kedy daný zákon nadobudne platnosť. Koniec času platnosti je v prvotný okamih nastavený na nekonečno, nakoľko nevieme do kedy daný zákon bude platiť. Akonáhle zákon bude poupravený či zrušený koniec času platnosti sa nastavý na časový okamih tejto udalosti.

- **Transakčný čas** je čas, kedy je n-tica relácia vložená do relácie či modifikovaná v rámci tejto relácie. Tabuľky obsahujúce transakčný čas nazývame tabuľky s transakčným časom. Predstavme si teda znova príklad zo zákonmi spomenutý vyššie. Transakčným časom budeme rozumieť čas kedy je daný zákon vložený do databázy či modifikovaný.

- **Bitemporálne tabuľky** sú tabuľky, ktoré obsahuju súčasne čas platnosti a transakčný čas. Tieto dva časy sú na sebe nezávislé a musíme vždy pri práci s týmito dátami zvážiť ako budeme s ktorým pracovať.

Príklad: Predstavme si že zákon o odvoze odpadu bol prijatý 25.3.2014 a nadobúda platnosť 1.6.2014. Následne však rok na to 20.3.2015, mestské zastupiteľstvo potrebuje daný zákon upraviť a tento upravený zákon nadobudne platnosť 1.6.2015. Bitemporálna tabuľka zachytávajúca tieto udalosti by vyzerala nasledovne.

Id zákona	Zákon	Čas platnosti	Transakčný čas
253	Odvoz odpadu	1.6.2014 - ∞	25.3.2014 16:00:00
253	Odvoz odpadu	1.6.2014 - 30.5.2015	20.3.2015 13:30:00
253	Odvoz odpadu	1.6.2015 - ∞	20.3.2015 13:30:00

Tabuľka 2.1: Príklad bitemporálnej tabuľky

2.3 Temporálny jazyk TSQL2

Jazyk TSQL2 je prvou snahou o formalizáciu potreby práce s temporálnymi časovými údajmi. Jedná sa o návrh Richarda Snodgrass-a z roku 1992. Autor tento jazyk pojal ako rozšírenie existujúceho jazyka SQL o nové koncepty podporujúce práve špecifiká popísané v predošlej podkapitole. Rok 1992 bol zároveň rokom vydania tretej verzie štandardu jazyka SQL, známej ako SQL92 [8]. Táto však vo svojej finálnej variante neobsahovala zmienky o temporálnych dátach a štandardy sa preto rozchádzajú.

TSQL2 vystupuje po stránke syntaxe ako rozšírenie SQL, teda podporuje používanie tabuliek v podobe známej práve z jazyka SQL. Navyše zavádza nové typy tabuliek, ktoré reprezentujú jednotlivé typy temporálnych dát. K nim pridáva novú dotazovaciu syntax, ktorá skrýva pred užívateľom interné informácie a sprístupňuje temporálne dáta podľa potreby. Zvolený prístup spoliehajúci sa na veľké množstvá informácií ukrytých pred užívateľom je pre relačné databázy netradičný a bol veľmi kritizovaný [4]. TSQL2 následne pokračovalo ako relatívne samostatný štandard, zatiaľ čo zlomky podpory temporálnych dát pribúdali do SQL noriem. Pokusy zjednotiť štandard TSQL2 so štandardom SQL prebehli na prelome milénií, v rámci práce na štandarde SQL99 [9], avšak z dôvodu kritiky prístupu TSQL2 k transparentnosti dát táto snaha zlyhala. TSQL2 je teda napriek spoločnému pôvodu nezávislým štandardom, ktorý vychádza z mierne odlišnej filozofie. Zatiaľ čo štandard SQL si zakladá na transparentnosti a všetky dáta a metadáta prezentuje formou užívateľovi dostupných relácií, TSQL2 pracuje s konceptom skrytých informácií. Intervaly platnosti sú tu prezentované formou špeciálnej syntaxe a nie je s nimi možné manipulovať. Toto je nie len rozdiel vo filozofií, ale týka sa aj praktických situácií pri konfigurácií a používaní databázového systému.

Zásadným prídavkom v TSQL2 je možnosť špecifickej práce s temporálnymi sadami záznamov na ktoré sa vzťahuje špecifická syntax týkajúca sa spájania týchto záznamov a práce s ich časovými zložkami. Konkrétnej podobu tejto syntaxe sa venujem v ďalšej kapitole.

2.4 Temporálne štandardy v rámci SQL

Po neúspešnej snahe o zakomponovanie jazyka TSQL2 do štandardu SQL sa vývoj práce v tejto oblasti uberal samostatne. Realisticky sa prvky práce s temporálnymi dátami do štandardu SQL zaniesli jeho verziou SQL2011 [10]. Táto popisuje viaceré rozšírenia, ktoré sa jednotlivé databázové systémy môžu rozhodnúť implementovať. Tieto popisujú ekvivalentné druhy tabuliek ako používa TSQL2, avšak zaviedol im vlastné pomenovania. Tabuľky s časom platnosti sú označované ako *application-time period tables*, tabuľky s transakčným časom majú názov *system-versioned tables* a bitemporálne tabuľky sú pomenované *system-versioned application-time period tables*. S výnimkou týchto rozdielov v pomenovaní sa tieto typy tabuliek po sémantickej stránke nelíšia.

V čom sa temporálna norma SQL od TSQL2 líši je práve už spomínaná transparentnosť. SQL2011 totiž popisuje prácu s hodnotami reprezentujúcimi intervaly platnosti, alebo existencie, ako prácu so štandardnými SQL stĺpcami. Dodáva síce novú syntax ktorou je možné týmto stĺpcom ich špecifický význam priradiť, avšak tieto sú stále plne viditeľné a ich vlastnosti dohľadateľné. Za zmienku tiež stojí, že ako dátový typ pre tieto časové intervaly autori nepoužili typ reprezentujúci interval, ale 2 samostatné stĺpce.

Nezávisle na štandardizácií sa viacerí autori pokúšali v priebehu rokov rozšíriť možnosti jednotlivých databázových systémov o prvky súvisiace s temporálnosťou dát. Pre túto prácu je špecificky významné rozšírenie z roku 2008 [5], ktoré rozširuje systém PostgreSQL o dátový typ *PERIOD*. Tento po sémantickej stránke reprezentuje časový rozsah, ktorého hranice sú dané štandardnými čas určujúcimi typmi jazyka SQL (*DATE*, *TIMESTAMP* ...). Okrem dátového typu ako takého poskytuje rozšírenie rozsiahlu sadu funkcií pre pohodlnú prácu s ním. Sú medzi nimi tak funkcie logické (*overlaps*), ako aj časové (*intersection*) a pre typ špecifické (*period_out*, *first*).

Implementácie SQL2011

Viaceré databázové systémy sa od zverejnenia normy SQL2011 pokúsili o implementáciu niektorých jej častí. Za zmienku stojí, že v čase písania práce žiadna z významných implementácií nepokrýva plnú funkcionalitu temporálnej zložky SQL2011.

Databázový systém Oracle vo svojej aktuálnej verzií podporuje verziovacie tabuľky. Jedná sa teda o implementáciu tabuliek s transakčným časom. Nad databázovým systémom Oracle bol ďalej vyvinutý temporálny databázový systém TimeDB.

V prípade systému PotgreSQL boli temporálne prvky postupne pridávané vo verziách 9.1 a 9.2, avšak jedná sa znova len o čiastočnú podporu pri práci s časovými typmi.

Vo všeobecnosti sa dá povedať, že podpora štandardu SQL2011 po stránke temporálnej funkcionality nie je v žiadnom z hlavných systémov dotiahnutá a nedá sa očakávať, že by sa tak stalo v najbližších rokoch. Problematickosť takejto implementácie totiž zásadne komplikujú rozdielne nároky na výkonnosť v prípade rôznych typov prípadov použitia.

Kapitola 3

Zrovnanie predošlých výsledkov

Cieľom tejto kapitoly ja vyhodnotiť výsledky diplomovej práce Radka Jelínka s názvom Temporální rozšíření pro PostgreSQL [11]. Práca sa venovala problematike temporálnych dát a implementácii temporálneho rozšírenia pre PostgreSQL. Na koľko ide o problematiku, ktorá do dnešnej doby nemá v jazyku PostgreSQL riešenie, je potrebné nájsť problémy vyššie zmieňovanej práce a navrhnúť vhodné riešenia. V rámci kapitoly budú uvedené príklady, ktoré sa budú sústrediť na porovnanie niektorých prípadov použitia temporálnych databází. Pôjde hlavne o prípady použitia, ktoré je možné vykonať v jazyku TSQL2. Nakoniec bude zhodnotená zložitosť zadávania dotazov a rýchlosť ich prevádzania.

V tejto kapitole zrovnávam možnosti analyzovaného rozšírenia so vzorovými príkladmi ktoré sa dajú násť v dokumentácií od R. Snodgrass-a [15].

3.1 Vytvorenie temporálnych tabuliek

V tomto príklade sa zameriame na vytvorenie temporálnej tabuľky. Konkrétne ide o vytvorenie tabuľky s časom platnosti. Príklad, ako aj ďalšie prevzané pochádza z dokumentácie R. Songrass-a [15] a je praktickou ukážkou možností TSQL2.

- **Varianta TSQL2:**

```
CREATE TABLE NBCShows
(ShowName CHARACTER (30) NOT NULL,
InsertionLength INTERVAL SECOND,
Cost INTEGER)
AS VALID STATE YEAR (2) TO NBCSeason;
```

- **Varianta PostgreSQL:**

```
CREATE TABLE NBCShows
(ShowName TEXT,
InsertionLength INTERVAL HOUR TO SECOND,
Cost INTEGER);
SELECT createValidTimeTable('NBCShows');
```

- **Zrovnanie:** Po stránke praktickosti varianta vyvinutá pre PostgreSQL výrazne nezaostáva. Je sice nutné jedno dodatočné volanie funkcie, ale pri bežnej, nízkej frekventovanosti vytvárania tabuliek je toto zanedbateľné. Po stránke vyjadrovacích schopností

je tu však viditeľná prvá výrazná odchýlka. TSQL2 umožňuje definíciu „typov časových rozsahov“. Na demonštrovanom príklade je čas platnosti vymedzený na polročné intervaly. Každý záznam v tabuľke môže takto nadobúdať len obmedzené druhy časov platnosti. Druhou odchýlkou je možnosť definovania vlastných kalendárov. TSQL2 umožňuje definíciu kalendárov, ako je v našom príklade NBCSeason. Tento obsahuje pomenované hodnoty polročných intervalov a je potom možné sa na ne odkazovať ako „Spring season 2015“, na rozdiel od „1-1-2015:31-7-2015“, alebo podobne. Špecifické druhy intervalov a vlastné kalendáre budú použité aj v ďalších príkladoch. Ich neprítomnosť je síce citeľná, neobmedzuje však zásadne teoretické vyjadrovacie možnosti rozšírenia.

3.2 Jednoduché vloženie záznamu do tabuľky s časom platnosti

Vkladanie záznamov do tabuliek s časom platnosti je po syntaktickej stránke relatívne jednoduché.

Čas platnosti pridáva iba potrebu ho špecifikovať v rámci operácie. Po funkčnej stránke sa môže vloženie záznamu prejaviť komplikovaným sledom operácií, tým sa však budeme venovať v ďalších príkladoch.

- **Varianta TSQL2:**

```
INSERT INTO NBCShows
VALUES('Super Bowl', 60, 55100)
VALID PERIOD '[1/1/1998 - 30/5/1998]';
```

- **Varianta PostgreSQL:**

```
SELECT insertValidTime(
'INSERT INTO NBCShows VALUES (''Roseanne'', ''30 second'', 12000)',
period('1994-01-01', '1994-04-01')
);
```

- **Zrovnanie:** Vkladanie takýchto záznamov je podobne stručné v oboch jazykoch. Varianta zvolená v rozšírení pre PostgreSQL vyžaduje navyše zadanie samotného príkazu *INSERT* v rámci textového reťazca. Toto nás zbavuje možnosti statickej analýzy príkazu a bez jeho spustenia nie je možné overiť jeho funkčnosť. Ďalším negatívom je nutnosť escape-ovania reťazcov v rámci príkazu. Takisto väčšina editorov nie je schopná zvýraznovať SQL syntax v rámci reťazcov. Na druhej strane, TSQL2 zavádza novú syntax nad rámec štandardného jazyka SQL v podobe kľúčového slova *VALID PERIOD*.

3.3 Jednoduchý výber z tabuľky s časom platnosti

V tomto príklade je našim cieľom zobrazenie názvov všetkých záznamov v tabuľke spolu s ich prislúchajúcimi časmi platnosti.

- **Varianta TSQL2:**

```
SELECT SNAPSHOT ShowName,
CAST(VALID(N) TO INTERVAL DAY)
FROM NBCShows(ShowName) AS N
WHERE N.ShowName = 'Roseanne'
```

- **Varianta PostgreSQL:**

```
SELECT
ShowName, valid_time
FROM
sequencedNBCShows() WHERE ShowName = 'Roseanne';
```

- **Zrovnanie:** TSQL2 v tomto príklade po syntaktickej stránke používa nové kľúčové slová. *SNAHPSHOT* značí, že výsledkom operácie bude štandardná (tzn. netemporálna) relácia. Z pohľadu PostgreSQL je každá relácia štandardná a preto nie je táto špecifikácia nutná. Ďalšou zmenou je tu syntax „NBCShows(ShowName) AS N“. Touto formou vytvárame v TSQL2 *N* ako pomenovanie pre reláciu s časom platnosti obsahujúcu len hodnotý stĺpca ShowName. Následne *VALID(N)* extrahuje čas platnosti pre jednotlivé riadky tejto relácie. Operácia *CAST* umožňuje prevod intervalu na iný typ intervalu prípadne iný dátový typ. V tomto prípade nás zaujíma dĺžka trvania intervalu v dňoch. V prípade rozšírenia PostgreSQL je pre tieto účely použitá funkcia *sequencedNBCShows*. Táto nám sprístupní dáta tabuľky *NBCShows* v podobe obsahujúcej čas platnosti. Na rozdiel od TSQL2 tu nie je možné definovať podmnožinu stĺpcov definujúcich vrátenú reláciu. Tento rozdiel sa môže prejaviť na výsledných hodnotách, nakoľko varianta TSQL2 zjednotí susediace intervaly ktoré zdieľajú hodnotu *ShowName*, zatiaľ čo druhá varianta v prípade odlišnosti v inom zo stĺpcov vráti tieto záznamy rozdelené. Tento nedostatok je teoreticky možné obísť pomocou dodefinovania pomocných funkcií, avšak ich detailom sa budeme venovať neskôr.

Príklad rozdielu pri výbere menšej množiny stĺpcov:

1. Stav tabuľky

Name	Amount	Valid
'Name1'	10	1.1.1994 – 28.2.1994
'Name1'	20	1.3.1994 – 1.4.1994

Tabuľka 3.1: Pôvodný stav tabuľky

2. Výber platnosti stĺpca Name v TSQL2:

```
SELECT Name, Valid(N) FROM SomeTable(Name) N
```

Name	Valid
'Name1'	1.1.1994 - 1.4.1994

Tabuľka 3.2: Výsledok príkazu v TSQL2

3. Výber platnosti stĺpca Name v rozšírení PostgreSQL

```
SELECT Name, valid_time FROM sequencedSomeTable()
```

Name	Valid
'Name1'	1.1.1994 – 28.2.1994
'Name1'	1.3.1994 – 1.4.1994

Tabuľka 3.3: Výsledok príkazu v Rozšírení PostgreSQL

3.4 Komplikovanejší výber z tabuľky s časom platnosti

Tento príklad je rozšírením predošlého výberu z tabuľky NBCShows. Tentokrát nás okrem názvov relácií a ich súvislých dĺžok trvania v dňoch bude zaujímať ich cena, konkrétne intervaly v ktorých ich cena zostávala rovnakou.

- **Varianta TSQL2:**

```
SELECT SNAPSHOT ShowName,
CAST(VALID(N) TO INTERVAL DAY),
CAST(VALID(B) AS INTERVAL DAY)
FROM NBCShows(ShowName) AS N,
N(Cost)(PERIOD) AS B
WHERE N.ShowName = 'Roseanne'
```

- **Varianta PostgreSQL:**

```
SELECT AB.gamename, HGB.price,
t__restructualize(array_agg(valid_time)) AS days, val
FROM sequencedshn() AS AB
CROSS JOIN (
SELECT gamename, price, t__restructualize(array_agg(Valid_time)) AS
val
FROM sequencedshn(true)
GROUP BY gamename, price
) as HGB
GROUP BY AB.gamename, val, hgb.price
```

- **Zrovnanie:** Príklad sa oproti predošlému na strane TSQL2 líši len málo. Napriek tomu toto rozšírenie zásadne pridáva komplexnosti príkazu. Časť *N(Cost)(PERIOD) AS B* rozširuje nami vopred definovanú reláciu s časom platnosti *N*, a pridáva k nej stĺpec *Cost*. Novovzniknutá relácia *B* teda sprístupňuje jednotlivé kombinácie hodnôt *ShowName* a *Cost* spolu s ich časmi platnosti. Kľúčové slovo *PERIOD* určuje, že susediace časy platnosti pre reláciu *B* nebudú zjednocované.

  Na strane PostgreSQL je riešenie o poznanie komplikovanejšie. Nakoľko nami používané rozšírenie sprístupňuje prácu s časom platností len pre tabuľky a nie pre ľubovolné relácie, je nutné túto logiku zreplikovať dodatočne. Funkcia *t__restructualize* je jednou z funkcií interne používaných v rámci tohto rozšírenia a umožňuje zjednotiť pole intervalov do najväčších možných súvislých intervalov. Agregačná funkcia *array_agg* je dostupná v štandardnej distribúcií PostgreSQL a zjednotí hodnoty daného stĺpca do jednej hodnoty formou poľa.

Táto PostgreSQL varianta len aproximuje funkcionalitu príkazu TSQL2 tak, aby zvýraznila podstatné prvky. Použitie *CROSS JOIN* by sa stalo problematickým pre tabuľky s viacerými záznamami, jednoduché nahradenie podmieneným príkoazom *JOIN* by tu nestačilo a bolo by nutné komplexnejšie zachytit spájanie relácií *HGB* a *AB*.

3.5 Vytvorenie bitemporálnej tabuľky

V nasledujúcom príklade si ukážeme vytvorenie bitemporálnej tabuľky.

- **Varianta TSQL2:**

```
CREATE TABLE NBC_FB_Insertion
(GameName CHARACTER ( 30 ),
InsertionWindow INTERVAL FootballSegment,
InsertionLength INTERVAL SECOND ( 3, 0 ),
CommercialID CHARACTER ( 30 ) )
AS VALID EVENT YEAR ( 2 ) TO HOUR AND TRANSACTION;
```

- **Varianta PostgreSQL:**

```
CREATE TABLE nbcfbinsertion
(GameName TEXT,
InsertionWindow INTERVAL HOUR TO SECOND,
InsertionLength INTERVAL HOUR TO SECOND,
CommercialId TEXT);
SELECT createBitemporalTable('nbcfbinsertion');
```

- **Zrovnanie:** Podobne ako pri tabuľke iba s časom planosti, oba príklady sú relatívne priamočiare. TSQL2 kľúčovým slovom TRANSACTION určuje, že sa bude jednať o tabuľku s transakčným časom. Druhé novopoužité kľúčové slovo je *EVENT*, označujúce fakt, že čas platnosti bude ukladaný formou okamihov (tzn. odkedy fakt platil), namiesto časových rozsahov. Na strane PostgreSQL rozšírenia táto možnosť dostupná nie je, preto je volaním funkcie *createBitemporalTable* vytvorená bitemporálna tabuľka používajúca rozsahy na určenie dôb platnosti.

3.6 Výber dát s neaktuálnym transakčným časom

Tento príklad demonštruje jednoduché použitie tabuľky s transakčným časom. Úlohou je zobraziť dáta, ktoré boli do systému vložené pred rokom 1990.

- **Varianta TSQL2:**

```
SELECT
GameName
FROM
NBC_FB_Insertion N
WHERE
TRANSACTION(N) PRECEDES DATE '1990-01-01'
```

- **Varianta PostgreSQL:**

```
SELECT
GameName
FROM
SnapshotNBC_FB_Insertion('1990-01-01'::timestamp)
```

- **Zrovnanie:** Na strane TSQL2 je v tomto príklade použitá nová vstavaná funkcia *TRANSACTION*. Jedná sa o funkciu ktorej vstupom je relácia ukladaná s pomocou transakčného času a výstupom je skalárna hodnota typu *DATE* určujúca čas v ktorom bola hodnota vložená do systému. Na strane rozšírenia PostgreSQL je použitá funkcia *SnapshotNBC_FB_Insertion*. Táto je vytvorená pre každú tabuľku s transakčným časom a umožňuje vybrať jej stav pre zvolený časový okamih. Po stránke prehľadnosti zápisu sú oboje varianty porovnateľné, avšak po funkčnej stránke demonštruje varianta v PostgreSQL zásadný problém. Nie je pri nej bohužiaľ dostupná iná zrovnávacia možnosť, než ekvivalencia časov, teda varianta „ľubovoľný okamih pred rokom 1990“ nie je realizovateľná.

3.7 Analýza zaznamenaných rozdielov

Vymenované zrovnania nám dávajú rozumný základ pre sformovanie jednoduchej analýzy funkčných medzier temporálneho rozšírenia PostgreSQL. Tabuľka nie je nijako prioritizovaná a jednotlivé medzery sú prezentované v poradí v akom sa v texte vyskytovali.

Funkčná medzera	Komentár
Užívateľom definované kalendáre	Funkcia ktorá spríjemňuje používanie, nemení však zásadne vyjadrovacie možnosti.
Špecifikácia typu intervalu pri tvorbe tabuľky s validným časom	
Nemožnosť vkladania dát s pomocou príkazu *INSERT*	Syntax prítomná v rozšírení PostgreSQL síce funkčne pokrýva možnosti TSQL2, je však ťažkopádna po viacerých stránkach. Vkladanie pomocou príkazu *SELECT* a zápis SQL príkazov formou textových literálov prácu znepríjemňujú a nenasledujú ani vzor SQL a ani TSQL2.
Možnosť zlievať podmnožinu atribútov tabuľky	Pomerne zásadný nedostatok výrazne limitujúci vyjadrovacie možnosti rozšírenia. Istá ručná aproximácia je možna s použitím nedokumentovaných, interných funkcií rozšírenia.
Možnosť zlievať rôzne podmnožiny atribútov pre jednu reláciu	Taktiež zásadný rozdiel oproti TSQL2. Syntax TSQL2 je v tomto prípade ale výrazne odlišná oproti štandardnému správaniu jazyka SQL. Ten jednotlivé relácie vstupujúce do príkazu *JOIN* oddelené čiarkou spojí formou kartézskeho súčinu. TSQL je schopná tieto výrazy spojiť na základe špecifickej syntaxe.
Možnosť rozlišovať medzi typmi tabuliek s transakčným časom (event vs. period)	Relatívne drobný rozdiel, period pokrýva funkcionalitu event čo sa funkčnosti týka

Tabuľka 3.4: Funkčné nedostatky rozšírenia PostgreSQL

3.8 Analýza výkonnosti rozšírenia PostgreSQL

V tejto časti sa zameriame na časovú výkonnosť nami testovaného rozšírenia PostgreSQL. Nakoľko nám nie je dostupná žiadna kanonická implementácia TSQL2, budeme výkonnosť jednotlivých dotazov porovnávať s očakávaniami, zameriavajúc sa najmä na využívanie indexov.

Taktiež budeme brať do úvahy len príkazy v ich jednoduchých variantách tak, ako sú popísané v dokumentácií. Tabuľky ktoré rozšírenie interne používa budeme používať len pre zrovnávanie výsledkov.

3.8.1 Analýza jednoduchého dotazu

Ako prvý sa pozrieme na jednoduchý dotaz vyberajúci dáta z tabuľky s validným časom a tieto dáta sa snaží filtrovať. V príklade tiež počítame s indexom vytvoreným na stĺpci na ktorom filtrujeme. Tabuľka obsahuje pár desiatok riadkov a výsledkom dotazu je jediný riadok.

- Príklad jednoduchého výberu z tabuľky

- **Príkaz:**

```
EXPLAIN ANALYZE
SELECT gamename FROM sequencedshn() where price > 3000000;
```

- **Výsledok:**

```
Function Scan on sequencedshn
(cost=0.25..12.75 rows=333 width=32)
(actual time=1.074..1.074 rows=1 loops=1)
Filter: (price > 3000000)
Rows Removed by Filter: 68
Planning time: 0.075 ms
Execution time: 1.236 ms
```

- **Diskusia:**
  Z popisu exekučného plánu môžeme vidieť prehľadávanie pomocou function scan metódy. To znamená, že sa podmienka aplikuje len na reláciu, ktorá je výsledkom volania funkcie *sequencedshn*. Toto je potenciálne pomalý krok, nakoľko PostgreSQL nie je schopný použiť index na prehľadanie dát. Toto môže byť extrémne limitujúcim faktorom pre vačšie obsahy tabuliek. Pre porovnanie som skúsila rovnakú podmienku pridať do dotazu *SELECT* ktorý je vykonaný v rámci volanej funkcie.

- Príklad komplexnejšieho výberu z tabuľky

  - **Príkaz:**

```
EXPLAIN ANALYZE
SELECT shn.*, t__shn__temp_data.valid_time
FROM shn
JOIN t__shn__temp_data ON shn.oid = t__shn__temp_data.row_oid
WHERE shn.price > 3000000;
```

  - **Výsledok:**

```
Nested Loop (cost=10000000000.14..10000000010.69 rows=1 width=31)
(actual time=0.085..0.087 rows=1 loops=1)
Join Filter: (shn.oid = t__shn__temp_data.row_oid)" " Rows Removed
by Join Filter: 68
-> Index Scan using priceshn on shn (cost=0.14..8.16 rows=1
width=19) (actual time=0.058..0.059 rows=1 loops=1)
        Index Cond: (price > 3000000)
-> Seq Scan on t__shn__temp_data
(cost=10000000000.00..10000000001.68 rows=68 width=20) (actual
time=0.012..0.019 rows=69 loops=1)
Planning time: 0.364 ms
Execution time: 0.244 ms
```

– **Diskusia:**
Ako je možné na prvý pohľad vidieť, tentokrát sa plánovaciemu algoritmu podarilo úspešne použiť index stĺpca *price*. Vďaka tomu bol schopný túto podmienku aplikovať ako prvú, čo ďalej môže urýchliť spájanie tabuliek. Efekt sa tu prejavil aj na výslednom čase, kde sa dá pozorovať niekoľkonásobné zlepšenie.

3.8.2 Analýza dotazu nad tabuľkou s transakčným časom

Dotazy nad tabuľkami s transakčným časom sú v analyzovanom rozšírení implementované spôsobom prakticky identickým s implementáciou výberu dát z tabuľky sčasom platnosti. Preto je možné závery z predošlého odstavca aplikovať aj v tejto situácií.

3.9 Zhrnutie nedostatkov rozšírenia PostgreSQL

Na nasledujúcich riadkoch zhrniem nedostatky prítomné v analyzovanom rozšírení PostgreSQL. Budem sa zaoberať tak nedostatkami po stránke návrhu/koncepcie, tak po stránke funkcionality ako aj stránke výkonnosti. Zoznam obsahuje len vybrané nedostatky ktoré boli zásadné, ako aj dostatočne všeobecné. Jazyk TSQL2 totiž obsahuje množstvo drobných konceptov a funkcií, ich výpis by však prekryl zásadnejšie problémy.

3.9.1 Návrhové nedostatky

- Prílišné používanie funkcií: S výnimkou vkladania dát do tabuliek s transakčnýn časom je väčšina funkcionality pokrytá použitím funkcií. Toto je zo stránky návrhu problematické z viacerých dôvodov. Prvým je nutnosť predávania SQL dotazov pomocou reťazcov. Toto obchádza statickú syntaktickú kontrolu a neumožňuje statickú analýzu kódu. Ďalším je nemožnosť priameho použitia nástrojov jazyka, ako operátory alebo agregačné funkcie. Použitie funkcií taktiež obchádza zaužívané idiómy, ako je napríklad vkladanie dát pomocou príkazu *SELECT*.

- Skryté informácie: dáta v tabuľkách sú používané netransparentne, tj. autor očakáva ich využitie funkciami, nie priamo užívateľom. Tým, že sa jedná o štandardnú nadstavbu PostgreSQL tieto dáta pri snahe dostupné sú, používaju však ťažko čitateľné identifikátory a menné konvencie.

3.9.2 Funkčné nedostatky

- Nemožnosť práce s temporálnymi sadami dát: Funkcie dostupné v rozšírení sú aplikovateľné len na tabuľky priamo. Pokiaľ však máme záujem temporálne dáta z tabuľky akýmkoľvek spôsobom predpripraviť (*sub-SELECT*, *VIEW*, *JOIN*,...), neexistuje pre tento medzivýsledok dostupná funkcia ani zdokumentovaný postup. Taktiež možnosť sekvencovania (tj. spájania rovnakých riadkov s prekrývajúcimi sa časmi platnosti) je dostupná len pre plnú sadu riadkov v tabuľke. Toto nepokrýva mnohé základné prípady použitia.

- Nemožnosť jemnejšej práce s transakčným časom: Výber z tzv. snapshot funkcií umožňuje len jednoduché vrátenie záznamov s časom transakcie obsahujúcim čas podaný ako parameter.

- Nemožnosť používať inú granularitu intervalov než timestamp: reálne systémy často operujú na báze dní, mesiacov prípadne rokov. Také správanie je simulovateľné s pomocou rozšírenia, ale zďaleka nedosahuje úrovne natívnej podpory.

3.9.3 Výkonnostné nedostatky

- Použitie funkcií už na úrovni tabuľky: Tým, že funkcia musí byť použitá na úrovni tabuľky, na rozdiel od úrovne sady riadkov, dochádza k zásadnému obmedzeniu možností plánovača. Tento je nútený najprv volanie funkcie vyhodnotiť s plnou sadou riadkov, napriek tomu že súčasťou dotazu je obmedzenie výrazne redukujúce počet riadkov.

Kapitola 4

Návrh novej funkcionality

Návrh mojich úprav rozšírenia bude pojatý primárne z pohľadu jednotlivých prípadov použitia. Popisy budú preto orientované na syntax a praktickosť jednotlivých príkazov. Výsledná forma vnútornej implementácie bude rozvedená až následne.

4.1 Hlavné koncepty

Pri zhrnutí nedostatkov mnou analyzovaného rozšírenia som vytipovala najkritickejšie nedostatky, rada by som sa preto v mojom návrhu zamerala na ich odstránenie. Hlavným viditeľným rozdielom by malo byť odstránenie veľkej závislosti na používaní funkcií, najmä pri dotazoch DML. Niektoré z funkcií môžu byť nahradené triggermi, niektore definíciami pohľadov a taktiež je možné funkcie odstrániť a nahradiť viditeľnými stĺpčekmi s ktorými môže užívateľ ďalej pracovať ako so štandardnými hodnotami. V prípade nutnosti použiť funkciu bude treba zvoliť rozumnú rovnováhu medzi jednoduchosťou hlavičky a možnosťou optimalizácie volaní v rámci funkcie s pomocou parametrov.

Ďalším rozdielom by mala byť dostupnosť, resp. čitateľnosť skrytých hodnôt. V existujúcom rozšírení boli zvolené menné konvencie ktoré síce zmenšujú možnosť kolízií, značne však zneprehľadňujú výsledné SQL dotazy v prípadoch keď je ich užívateľ nútený použiť priamo (čo je z vyššie popísaných dôvodov časté). Rozdiel nastane aj na strane dokumentácie, kde by tieto stĺpce/tabuľky mali byť výraznejšie popísané ako štandardná cesta pre prístup k dátam.

Iným pohľadom na plánované zmeny je rozšírenie poskytovanej funkcionality. Pri zrovnávacích testoch ktoré som vykonala som často narazila na funkčné medzery. Tieto sa vyskytli tak vo forme detailov, ako aj výrazne možnosti obmedzujúcich problémov. Tieto nedostatky neboli bohužiaľ zachytené ani v existujúcej dokumentácií, takže je niekedy ťažké predpokladať, či sa jedná o chybu, alebo zámer návrhu. Funkcionalitu ktorú v rámci práce vyviniem preto plánujem pokryť tak popisom, ako aj príkladmi.

4.2 DDL – príkazy pre definíciu dát

V kategórií príkazov pre definíciu dát sa obmedzíme na príkazy pre tvorbu jednotlivých typov tabuliek, nakoľko sa jedná o jediné bežne popisované DDL príkazy jazyka TSQL2. Po stránke výrazových možností dovoľuje TSQL2 dodefinovvať granularitu intervalov pre tabuľky s časom platnosti ako aj typ ukladaných časových dát pri tabuľkách s transakčným

časom. Oboje tieto možnosti nie sú po stránke funkčnosti nutné a prvotná implementácia ich môže bezpečne vynechať.

Po stránke syntaktickej realizácie ponúka PostgreSQL dve hlavné možnosti. Prvou je volanie funkcie pomocou ktorej sa tabuľka stane temporálnou. Druhou je možnosť tzv. *event trigger-u*, ktorý reaguje na vytvorenie tabuľky [6]. Tento by mohol z vhodne zvolených typov/mien atribútov tabuľku identifikovať ako temporálnu a vhodne s ňou naložiť.

Varianta s použitím *event trigger-u* vyžaduje o jedno volanie príkazu menej, na druhej strane však zavádza novú, skrytú, syntax v podobe kľúčových mien stĺpcov a neposkytuje možnosti budúcej rozšíriteľnosti porovnateľné s flexibilným volaním funcie. Oboje metódy by však mali byť v prvotnej variante prevoditeľné a malo by byť teoreticky možné ich používať spoločne.

Príklad vytvorenia temporálnej tabuľky s použitím funkcie(totožný s existujúcim rozšírením):

```
CREATE TABLE SomeTable
(Name TEXT PRIMARY KEY,
Amount INTEGER);
SELECT createValidTimeTable('SomeTable');
```

4.3 DML – príkazy pre manipuláciu s dátami

Príkazy pre manipuláciu s dátami z pohľadu užívania rozdelím na tie ktoré dáta v tabuľkách ovplyvňujú a tie ktoré sa na tieto dáta dotazujú. Funkčnosť pre jednotlivé typy temporálnych tabuliek zhrniem v jednotlivých podkapitolách zvlášť.

4.3.1 Príkazy pre zmenu dát

Vkladanie dát do tabuliek – INSERT

Ako prvé v jednoduchosti rozoberieme vkladanie dát do tabuliek s transakčným časom. Tu by nemala byť nutná žiadna ďalšia informácia od užívateľa, tj. do tabuľky by sa malo môcť dať vkladať dáta rovnako ako do netemporálnej tabuľky. Za zmienku stojí, že rovnaké riešenie bolo zvolené aj v nami analyzovanom rozšírení PostgreSQL.

Pre dáta s časom platnosti je situácia odlišná, nakoľko užívateľ musí vedome poskytnúť časy platnosti. Tu sa ponúkajú dva syntakticky odlišné prístupy. Vkladané dáta môžu byť poskytnuté ako parameter volania funkcie, ktorá by ich vložila, alebo môžu byť zadané formou štandardného SQL príkazu ako hodnota špecifického, kľúčového stĺpca. Varianta volania funkcie, zvolená aj v analyzovanom rozšírení, prináša do jazyka veľmi neintuitívny koncept vkladania dát pomocou volania príkazu *SELECT*. Naviac, táto metóda je potom jedinou možnou formou vloženia dát (príkaz *INSERT* je pre takúto tabuľku nevyužitý a zbytočný).

Varianta s použitím hodnoty špeciálne pomenovaného stĺpca je v tomto smere menej invazívna a uchováva hlavné koncepty jazyka SQL. Na druhej strane však vyžaduje nutnosť znalosti špeciálne pomenovaného stĺpca v tabuľke. Prípadnú nutnú verifikačnú a dodatočnú logiku implementovanú vo funkcií môže v tomto prípade replikovať trigger.

Z môjho pohľadu výhody práce s natívnym príkazom *INSERT* výrazne prevažujú výhody použitia špeciálnej funkcie a preto bude implementovaná práve táto varianta. Po funkčnej stránke vkladanie dát naráža na nesmierne kľúčový koncept identity záznamu. Je

nutné vedieť rozlíšiť kedy vkladaný záznam reprezentuje novú entitu a kedy reprezentuje potencionálne starú entitu, s prípadnou nutnosťou orezania existujúcich intervalov a tiež kedy je záznam úplne duplicitný a je ho nutné zamietnuť. Návrh riešenia identity záznamu je zdieľaný pre viaceré prípady použitia a preto sa mu venujem v samostatnej sekcii.

Po stránke zrovnania z originálnym rozšírením, najzásadnejším rozdielom je zmena prístupu k identite entít. V pôvodnom rozšírení nie je žiaden podobný koncept dostupný. A keďže zvolená schéma nedovoľuje zachovávať primárny kľúč, starostlivosť o nádväznosť a neprekrývanie intervalov platnosti pre jednotlivé fakty jednej entity ostáva úplne v rukách užívateľa. Nová implementácia umožní automatické upravenie prekrývaných intervalov a udrží tak ich náväznosť bez nutnosti ďalších zásahov.

Príklad vloženia dát do tabuľky s časom platnosti v existujúcom rozšírení(tabuľka z príkladu 4.3.1.1):

```
SELECT
    insertValidTime('
    INSERT INTO
        SomeTable
    VALUES (''A'', ''30'')', period('1994-01-01', '1994-04-01')
)
```

Návrh funkčne totožného vloženia dát:

```
INSERT INTO
    SomeTable (Name, Amount, valid_time)
VALUES
    ('A', '30', period('1994-01-01', '1994-04-01'))
```

Zmena dát v tabuľkách – UPDATE

Zmena hodnoty záznamu pre tabuľky s transakčným časom nepožaduje žiadne dodatočné informácie, ktoré by boli nutné pre prevedenie úpravy. Preto, podobne ako v prípade príkazu *INSERT*, je možné od užívateľa použitie tabuľky s temporálnym časom úplne odtieniť. Pracovať s tabuľkou teda bude môcť priamo pomocou príkazu *UPDATE* a o potrebnú logiku sa postarajú triggery.

Pri zmenách hodnôt pre tabuľky s časom platnosti musíme rozlišovať jednotlivé typy zmien dát. Prvým možným typom je zmena hodnoty štandardného stĺpca pre celú dĺžku platnosti záznamu. Táto by po stránke užívateľskej pohodlnosti nemala byť odlišná od štandardného príkazu *UPDATE*. Jedinou potencionálne očakávateľnou zmenou by tu bolo zliatie dvoch časovo susediacich záznamov s rovnakými hodnotami do jedného. Táto funkčnosť môže v prípade potreby byť obslúžená triggerom. Táto funkcionalita je po syntaktickej stránke rovnako riešená aj v analyzovanom rozšírení. Ďalším možným typom zmeny je zmena platnosti daného záznamu. Tu prichádzajú do úvahy dve varianty podľa toho, či je stĺpec s časom platnosti sprístupnený formou štandardného stĺpčeka alebo nejakej skrytej konštrukcie. V prípade štandardného stĺpca sa jedná o situáciu rovnakú ako pri zmene štandardnej hodnoty záznamu a úlohu je teda možné obslúžiť priamo príkazom *UPDATE*, v prípade potreby doplneným triggerom.

Príklad zmeny času platnosti v analyzovanom rozšírení:

```
SELECT
updateValidTime(‘SomeTable’, array(
    SELECT
        oid
    FROM
        ‘SomeTable’
    WHERE
        name = ‘A’),
’contains(valid_time, "2000-01-01"::timestamp)’,
period(’2001-01-01’::timestamp, now()::timestamp));
```

Návrh rovnakej zmeny dát:

```
UPDATE
    SomeTable
SET
    valid_time = period(’2001-01-01’::timestamp, now()::timestamp))
WHERE
    name = ‘A’ and
    contains(valid_time, "2000-01-01"::timestamp)
```

Poslednou variantou zmeny hodnoty je zmena hodnoty štandardného stĺpca platná len v nejakom intervale. Výsledkom zmeny teda musí byť väčšie množstvo riadkov. Pri tejto zmene dát je otázne akým spôsobom by mala byť do systému zavedená. Keďže výsledkom operácie má byť väčšie množstvo riadkov ako pred operáciou, štandardná sémantika by naznačovala použitie príkazu *INSERT*. Na druhej strane je výsledkom zmena existujúcich záznamov, čo by odpovedalo použitiu príkazu *UPDATE*. Jednou z možností by bolo tento prípad použitia nepodporovať a vynútit oba samostatné kroky *UPDATE* a *INSERT* od užívateľa. Druhou variantou ktorá sa ponúka je použitie špeciálne pripravenej funkcie pre tento účel pripravenej. Táto varianta však naráža na neintuitívne použitie príkazu *SELECT* na vkladanie dát. Poslednou variantou je podpora s pomocou jedného z volaní *UPDATE*/*INSERT*, kde sa trochu vhodnejšie pôsobí zrovna príkaz *INSERT*. Z dôvodu prevencie zbytočného rastu množstva kľúčových slov a konceptov zahrniem túto variantu pod funkcionalitu poskytovanú príkazom *INSERT*.

Príklad zmeny hodnoty atribútu vo vybranom intervale pomocou príkazu *INSERT*:

1. Prvotné vloženie dát

   ```
   INSERT INTO
       SomeTable (Name, Amount, valid_time)
   VALUES
       (‘A’, ‘30’, period(’1994-01-01’, ’1994-04-01’))
   ```

2. Zmena dát v určitom intervale

   ```
   INSERT INTO
       SomeTable (Name, Amount, valid_time)
   VALUES
       (‘A’, ‘40’, period(’1994-03-01’, ’1994-04-01’))
   ```

3. Výsledné dáta

Name	Amount	Valid Time
'A'	30	1994-01-01:1994-02-28
'A'	40	1994-03-01:1994-04-01

V zrovnaní s existujúcim rozšírením sa jedná o výraznú zmenu po stránke funkcionality v prípade tabuliek s časom platnosti. V pôvodnej variante je poskytnutá jediná špecifická metóda, ktorej účelom je zmena dĺžky intervalu platnosti. Ako som spomenula v predošlej kapitole, táto núti užívateľa obsluhovať prekryvy jednotlivých časových intervalov a výrazne obmedzuje pohodlnosť používania. Keďže pôvodné rozšírenie neobsahuje koncept identity entít, dovoľuje táto poskytovaná funkcia aj také zmeny intervalov platnosti, ktoré by spôsobili prekryv 2 nadväzujúcich faktov.

4.3.2 Mazanie temporálnych dát – DELETE

Pri mazaní temporálnych dát v tabuľkách s transakčným časom je zmazanie záznamu špecifickou formou úpravy. Stará hodnota je rovnakým spôsobom odložená s ukončeným časom platnosti, na rozdiel od pridania novej verzie záznamu však žiadna pridaná nie je. Keďže pre túto úlohu nie je potrebné dodať ďalšie informácie, je možné toto mazanie implementovať s pomocou triggeru a štandardného volania príkazu *DELETE*.

Na strane tabuliek s dátami s časom platnosti môže mazanie nadobúdať viacerých variánt. Buď je našim cieľom zmazať jeden fakt, tj. záznam s časom platnosti, úplne, alebo chceme zmazať len časť jeho platnosti. V prvom prípade sa jedná o relatívne jednoduchú operáciu ktorá nepotrebuje zvláštnu dodatočnú podporu. Užívateľ bude môcť použiť štandardný príkaz *DELETE* a jeho pomocou záznam zmazať. V prípade zmazania faktu len pre určitý časový úsek, tento úkon je problematické príkazom *DELETE* pokryť a po funkčnej stránke ho pokrýva príkaz *UPDATE*. Nakoľko sa po stránke sémantiky jedná o mazanie, bude tieto prípady použitia nutné vyskúšať a v prípade neintuitívnosti riešenia poskytnúť pomocnú funkciu.

4.3.3 Príkazy pre dotazovanie nad dátami

Problematika dotazovania nad temporálnymi dátami je objemovo rozsiahlejšia než problematika zadávania dát, preto bude táto sekcia hlbšie členená. Bude tu popísaný samostatne výber dát z tabuliek so systémovým a transakčným časom ako aj jednotlivé komplikovanejšie prípady.

Výber dát z tabuliek s transakčným časom

Pri tabuľkách s transakčným časom umožňuje jazyk TSQL2 klasickú formu dotazovania ako nad štandardnou tabuľkou. K nemu poskytuje ďalej funkciu *transaction*, ktorej vstupom je záznam a výstupom je rozsah reprezentujúci transakčnú platnosť daného záznamu. V prostredí štandardného SQL jazyka by bolo intuitívnejšie, keby transakčný čas záznamu bol jednoducho jednou z vlastností záznamu, rovnako ako iné atribúty. Druhým konceptom ktorý je nutné previesť je myšlienka, že kým nie je súčasťou dotazu volanie funkcie *transaction*, sú vrátené len záznamy ktoré sú aktuálne platné.

Táto funkčnosť nie je jednoducho prevoditeľná do jazyka SQL, je preto nutné ju pokryť konceptami ktoré PostreSQL podporuje. Máme tu na výber použitie volania funkcie, prípadne vytvorenie pohľadu. Nakoľko je varianta vracajúca len najaktuálnejšiu variantu

záznamov len statickým pohľadom na dané dáta, varianta použitia pohľadu pôsobí vhodnejšie. V situáciách, kde nás výber záznamu v špecifickom čase zaujíma môžeme dáta teda reprezentovať formou, kde je transakčný čas záznamu dostupný ako jednoduchá hodnota v rámci vráteného záznamu.

Príklad výberu dát v existujúcom rozšírení PostgreSQL (predpokladáme že testovacia *SomeTable* tabuľka je upravená na tabuľku s transakčným časom):

```
SELECT
    Name
FROM
    SomeTable_FB_Insertion('1990-01-01'::timestamp)
```

Príklad výberu dát podľa môjho návrhu:

```
SELECT
    Name
FROM
    SomeTableTransaction
WHERE
    contains(transaction_time, '1990-01-01'::timestamp)
```

V príklade použitý operátor *contains* je možné jednoducho zameniť za iný operátor, ako napríklad *overlaps*.

Výber dát z tabuliek s časom platnosti

Dáta obsiahnuté v tabuľkách s časom platnosti reprezentujú jednotlivé fakty, kde každý riadok obsahuje hodnoty atribútov a časový rozsah počas ktorého je množina hodnôt atribútov platná. Jeden z najzásadnejších konceptov, ktorý bol navyše v analyzovanom rozšírení obsiahnutý len okrajovo, je zlievanie susediacich intervalov s rovnakými hodnotami. Jazyk TSQL2 totiž umožňuje zvoliť podmnožinu stĺpcov tabuľky s ktorou chceme pracovať a spojiť prípadne sa prekrývajúce intervaly s rovnakými hodnotami stĺpcov do jedného. Koncept pre zjednodušenie ilustrujeme na príklade.

Podobne ako pri predchádzajúcom prípade, ako možné riešenia sa ponúkajú dve možnosti, ktorými sú pohľad na dáta alebo volanie pomocnej funkcie. Keďže v tejto situácií je nutné poskytnúť vstupný parameter, použitie pohľadu na dáta nie je postačujúce. Vstupným parametrom tejto funkcie bude musieť byť zoznam stĺpcov, ktoré má výsledná relácia obsahovať.

Príklad výberu dát pre zvolené obdobie platnosti s pomocou funkcie:

```
SELECT
    Name, valid_time
FROM
    sequenced('SomeTable',   '1990-01-01'::timestamp)
```

Toto riešenie bohužiaľ naráža na problémy spojené s použitím funkcií. Jedná sa o neflexibilné riešenie, ktoré je ťažko prispôsobiť v prípade potreby a podobne je komplikované riešenie prípadných výkonnostných problémov. Z týchto dôvodov by bolo vhodné poskytnúť aj dobre zdokumentovaný spôsob ako dosiahnúť tú istú funkcionalitu bez použitia funkcie vracajúcej kompletnú sadu záznamov. Z pohľadu klasického SQL sa jedná o aplikáciu akejsi agregačnej funkcie pracujúcej na časových rozsahoch. Finálny návrh teda obnáša implementáciu funkcie pokrývajúcej výber podmnožiny atribútov so zlievaním a implementáciu a dokumentáciu agregačnej funkcie zjednocujúcej intervaly.

Príklad výberu dát s pomocou agregačnej funkcie:

```
SELECT
    Name, sequenced(valid_time)
FROM
    SomeTable
GROUP BY
    Name
```

V sekcií 3.4 je ďalej ilustrovaná ďalšia zaujímavá funkcionalita týkajúca sa výberu dát z tabuliek s časom platnosti, a to automatické použitie príkazu *JOIN* pri spájaní dvoch rôznych pohľadov na tú istú tabuľku. Toto správanie pôsobí v priestore tradičných relačných databází veľmi netradične a neintuitívne a preto podobnú funkcionalitu v našom rozšírení implementovať nebudeme. V prípade, že užívateľ potrebuje spojiť dve tabuľky, tieto sú spojené formou kartézskeho súčinu a všetky iné formy spájania musia byť explicitné, použitím príkazu *JOIN*.

V porovnaní s pôvodným rozšírením ja zvolené riešenie relatívne podobné. Pôvodné riešenie rovnako pridáva stĺpec obsahujúci čas platnosti do vrátenej relácie a necháva užívateľa pracovať s ním podľa uváženia. Je v ňom však nutné použiť volanie funkcie, ktoré zakrýva reálnu formu uložených dát. Toto môj návrh odstraňuje a užívateľovi je sprístupňená priamo tabuľka s reálnymi dátami.

Filtrovanie temporálnych dát – klauzula WHERE

Filtrovanie dát v klauzuli *WHERE* sa týka nielen výberu dát, ale rovnako aj ich úpravy pomocou príkazov *UPDATE* a *DELETE*. Keďže úprava dát sa deje výlučne nad tabuľkami, musíme dbať na to, aby bola práca s výstupmi volaní funkcií, prípadne pohľadov s nimi konzistentná.

Je potrebné postarať sa o viacero druhov konzistencie. Prvým je typová konzistencia, chceli by sme teda, aby všetky hodnoty reprezentujúce temporálne časové rozsahy boli jednotného typu. Tento požiadavok je splnený už v existujúcom rozšírení použitím výlučne typu *PERIOD*.

Druhým požiadavkom je konzistencia na úrovni pomenovania temporálnych atribútov. Rozširenie je v tomto relatívne konzistentné, používajúc interne mená používajúce ako oddelovač symbol „_“. Možnou alternatívou k nemu by bolo použitie tzv. camel case zápisu, teda identifikátora *validTime* namiesto *valid_time*. Pokiaľ je používanie menných konvencií konzistentné, oba prístupy sú prípustné. V mojej práci som sa rozhodla ponechať pôvodné pomenovania, preto pracujem s oddelovačmi v podobe "_"v názvoch stĺpcov a funkcií.

Po stránke dostupných funkcií a operátorov je existujúce rozšírenie obsahujúce typ *PERIOD* pomerne vyčerpávajúce a obashuje všetky štandardné operácie dostupné nad intervalmi.

Klauzula *WHERE* je tiež najpodstatnejšou zložkou SQL dotazov týkajúca sa výkonnosti. Najzásadnejšiemu problému vystupujúcemu z analýzy, nutnosti použiť function scan pri výbere dát ako výsledku funkcie, sa vyhneme použitím výberu dát zo štandardnej tabuľky, resp. pohľadu. Vo chvíli ako sú dáta dostupné týmto spôsobom je štandardná optimalizácia v rukách PostgreSQL. Ďalšou témou týkajúcou sa optimalizácie je použitie indexov. V prípade regulárnych SQL tabuliek je tvorba indexov výlučne pod kontrolou užívateľa spravujúceho tabuľky. Naše rozšírenie však zavádza koncept špeciálnych stĺpcov, preto je na mieste možnosť generovať pre ne indexy automaticky. Rozšírenie obsahujúce dátový typ *PERIOD* možnosť indexu obsahuje, preto je len nutné zvoliť, či tento index generovať alebo nie. Z povahy temporálnych tabuliek vyplýva, že dotazy nad nimi budú vo veľkej miere obsahovať práve prácu s časom, čo nahráva variante s automatickým pridaním indexu. Nakoľko však táto varianta môže prekážať, v prípade priveľkého množstva záznamov, alebo častých zmien intervalov, je vo finálnej variante pridanie indexu ponechané na užívateľovi.

4.3.4 Identita entít v temporálnych reláciách

Na problematiku rozpoznania identity záznamov v temporálnych tabuľkách sme narazili vo viacerých častiach návrhu. Identita pre bežné relačné tabuľky je v databázových systémoch riešená formou primárnych kľúčov. Tieto spĺňajú dve hlavné úlohy. Ako prvé vynucujú unikátnosť svojej hodnoty v rámci relácie. Databázový systém preto automaticky zamietne operácie *INSERT* alebo *UPDATE* ktoré by túto unikátnosť porušili.

Druhou úlohou primárnych kľúčov je slúžiť ako cieľ odkazov v podobe cudzích kľúčov. Tento koncept označovaný ako referenčná integrita zabezpečí, že pokiaľ by užívateľova operácia spôsobila, že sa záznam odkazuje na neexistujúci záznam, táto operácia skončí chybou.

V kontexte temporálnych databází je potrebné tento koncept identity jemne poupraviť. Je možné predpokladať, že užívatelia budú od neho očakávať oboje vyššie spomenuté funkcie, jedinečnosť a referenčnú integritu.

Unikátnosť

Unikátnosť záznamov má pre nás v oboch typoch temporálnych tabuliek obdobný význam. Očakávané správanie v oboch je, že neexistujú dva časové záznamy popisujúce tú istú entitu s prekrývajúcimi sa intervalmi platnosti. V prípade transakčného času je toto správanie automaticky zabezpečené jeho návrhom. Užívateľ totiž nemôže priamo meniť transakčné intervaly a tieto systém dopočítava tak aby na seba nadväzovali a končili, resp. začínali, v časoch zmien. V prípade tabuliek s časom táto funkcionalita nie je samozrejmá a pôvodné rozšírenie ju ani neposkytuje. Prvým problémom je tu práve operácia nedostupná v prípade transakčného času a tou je manuálna úprava intervalu platnosti. Tento môže užívateľ zmeniť tak, že vznikne prienik s iným intervalom týkajúcim sa tej istej entity. Druhou takouto operáciou je vloženie nového faktu do temporálnej relácie, ktorý bude popisovať entitu v čase, pre ktorý tam už táto záznam má.

V oboch prípadoch sa môžeme s touto situáciou, na rozdiel od tradičných relácií, vysporiadať dvomi spôsobmi. Prvým je vyhlásenie chyby a zamietnutie operácie. Druhým, špecifickým pre temporálne dáta je ustúpenie pôvodných intervalov. Pre oboje je možné si predstaviť situácie v ktorých by toto správanie bolo žiadané. V prípade nesprávnej úpravy intervalu platnosti by pravdepodobne bolo správne užívateľa upozorniť a ukončiť operáciu chybou. Na druhej strane si môžeme predstaviť situáciu, keď máme v databáze záznam o zamestnancovi a chceme do nej vložiť fakt, že počas obmedzeného obdobia bol alebo bude

preradený do iného týmu, ale eventuálne sa vráti do svojho pôvodného. Ak by sme tento vklad ukončili chybou, bol by užívateľ nútený pôvodný záznam rozdvojiť a dva novovzniknuté záznamy patrične skrátiť. Je teda pravdepodobné, že v tomto prípade by očakával automatické ustúpenie dát.

V mojej práci som zvolila kombinované riešenie. Pokiaľ by temporálnu unikátnosť narušila operácia *UPDATE* ukončím túto operáciu chybou. Pokiaľ však tento stav nastane pri vložení nového záznamu, existujúce záznamy sa upravia automaticky tak, aby k prekryvom nedošlo. Túto voľbu som urobila na základe testov s relatívne realistickými dátami a vlastnými očakávaniami. Je však možné, že v iných situáciách by bolo vhodné iné správanie. Toto je preto jednou z možných oblastí budúceho vývoja.

Referenčná integrita

Referenčná integrita v temporálnych dátach je koncept o niečo komplikovanejší ako ich jedinečnosť. Je totiž nutné najprv rozobrať scenáre rôznych typov entít odkazujúcich sa na typy iné. Tabuľky s transakčným časom nám tu situáciu avšak znovu zjednodušujú, keďže zmeny v nich sa môžu týkať jedine ich aktuálne platnej časti. Vzťahy medzi tabuľkami bežnými (medzi ktoré sa teda radí aj atuálne platná časť dát s transakčným časom) nám zabezpečuje databázový systém formou cudzích kľúčov. Vzťah medzi bežnou tabuľkou a tabuľkou s časom platnosti je v oboch možných smeroch podobný tradičnému. Ostáva nám teda vzťah medzi dvoma tabuľkami s časmi platnosti (potencionálne bitemporálnymi). Pri týchto je možné nájsť neplatný odkaz vo forme odkazovania sa na entitu v čase platnosti, pre ktorý aspoň v nejakom bode neexistuje. Tieto by malo byť možné kontrolovať pre užívateľa automaticky a v prípade porušenia vzťahu vrátiť chybu. Technickým detailom tejto operácie sa budem venovať v samostatnej kapitole v popise implementácie.

Podoba identifikátorov

Na základe blízkosti identifikátorov entít v temporálnych tabuľkách a štandardných primárnych kľúčov považujem za ideálny návrh, kde tieto identifikátory budú z primárnych kľúčov vychádzať. Ideálne, pokiaľ dochádza ku konverzií z bežnej tabuľky na tabuľku s časom platnosti, mal by byť systém automaticky schopný na základe pôvodného primárneho kľúča vygenerovať nový identifikátor slúžiaci pre temporálnu tabuľku. Takto vzniknutý identifikátor musí byť umožnené užívateľovi následne zmazať, prípadne pridať.

Kapitola 5

Rozbor implementácie existujúceho rozšírenia a návrh implementačných zmien

V tejto kapitole sa zameriam na interné detaily implementácie existujúceho rozšírenia. Budem sa zameriavať na jednotlivé konkrétne návrhové rozhodnutia, nimi spôsobené problémy a možnosti ich zmeny. Poradie v ktorom sa budem jednotlivým prvkom venovať nereflektuje ich vážnosť.

5.1 Stručný popis implementácie pôvodného rozšírenia

Analyzované rozšírenie je distribuované formou jedného SQL súboru. Tento po zavedení pridáva výlučne sadu funkcií, všetky pomocné tabuľky sú vytvorené až v prípade potreby. Tento koncept sám o sebe neprekáža, ale zanáša množstvo zbytočného kódu do tiel funkcií a je zbytočne netransparentný. Medzi pridanými funkciami nájdeme funkcie pre zmenu tabuľky na temporálnu, pre vloženie dát s časom platnosti a pre výber dát tak s transakčným časom, ako aj časom platnosti. Autor zvolil jemne nekonzistentné pomenovanie funkcií, kde časť funkcií je špecifická pre jednotlivé temporálne tabuľky a sú vyrobené podľa potreby. Čo sa menných konvencií pri objektoch ktoré by mali ostať bežnému užívateľovi skryté zvolil autor notáciu s prefixom *t___*.

Za samostatnú zmienku stojí dizajn pomocných tabuliek v prípade pridávania transakčného času, resp. času platnosti. V oboch prípadoch zvolil autor rovnakú štruktúru. K pôvodnej tabuľke je pridaný OID stĺpec, ktorý je následne zvolený ako primárny kľúč. Ďalej je vytvorená pomocná tabuľka ktorá obsahuje odkazy na jednotlivé OID a k nim prislúchajúce hodnoty transakčného času alebo času platnosti. Tento návrh je síce funkčný avšak prináša niekoľko zvláštných vlastností. V prvom rade je táto druhá tabuľka z pohľadu relácií zbytočná, nakoľko sa viaže k originálnej striktne vzťahom 1:1. Tento krok by mohol byť odôvodnený snahou nezasahovať do originálnej tabuľky nad rámec potreby. Tu je však nutné podotknúť, že po úprave na temporálnu je originálna tabuľka po stránke jej obsahu relatívne zbytočná. V prípade existencie viacerých variant jednotlivých záznamov je nemožné bez pomocnej tabuľky rozoznať aktuálne údaje od historických. Táto forma návrhu však prináša zjednodušenia na strane implementácie referenčnej integrity. Interná implementácia tabuliek je bližšie ilustrovaná na diagramoch priložených v ďalšom odstavci.

5.2 Návrh zmien rozšírenia

V mojom návrhu implementácie by som sa chcela sústrediť na čistotu implementácie v súlade s tradičnými zásadami práce s relačnými databázami. Ako prvé nepovažujem za šťastné schovávať pred užívateľom implementačné detaily. V prvom rade toho samotná relačná databáza nie je schopná (ak má človek môcť využivať tabuľky musí k nim mať prístup) a v rade druhom sa takto systém ochudobňuje o potencionálne neštandardné využitia. Preto by som chcela zvoliť relačnú schému, ktorá bude:

- Mať jednouché a intuitívne menné konvencie
- Každá tabuľka bude obsahovať zmysluplné dáta. To znamená, že *SELECT * FROM X* vráti sadu dát s ktorou sa dá ďalej pracovať

Rovnako by som rada pozmenila prístup k procesu úpravy tabuľky na jej temporálnu variantu. V originálnej implementácií sa autor obmedzil na pridanie *OID* stĺpca a zmenu primárneho kľúča. Toto riešenie je v súlade s originálnym návrhom schémy ale obsahuje niekoľko nedostatkov. V prvom rade zmena primárneho kľúča neponecháva pôvodnú unikátnosť primárneho kľúča. Tento problém autor obchádza nutnosťou používať funkcie pre zmenu dát pri tabuľkách s časom platnosti, toto riešenie je však obmedzujúce, nezaberie v prípade že užívateľ z nejakého dôvodu zmení dáta priamym spôsobom a hlavne je neprítomné v prípade tabuliek s transakčným časom. Tam je užívateľovi dovolené vložiť nový riadok, porušujúc originálne primárne kľúče. Túto konzistentnosť by bolo ideálne udržovať aj po úprave tabuľky na tabuľku temporálnu, avšak je nutné dbať na jej správnu formu: v bežnej relácií sa nemôžu vyskytnúť dva záznamy s rovnakým primárnym kľúčom, zatiaľ čo v temporálnej tabuľke sa nesmú vyskytnúť dva záznamy s rovnakým primárnym kľúčom, ktorých rozsahy času platnosti alebo transakčného času by sa prekrývali.

5.3 Návrh relačnej schémy v novej revízií rozšírenia

V tejto kapitole stručne popíšem nový návrh databázovej schémy, ktorú následne budem implementovať ako základ rozšírenia. Ako som vyššie popísala, jednou z hlavných myšlienok bude, aby výber dát z hlavnej tabuľky dával konzistentné výsledky. V prípade tabuľky s časmi platnosti sa môže ako sľubná javiť varianta s hlavnou tabuľkou obsahujúcou aktuálne platné dáta. Túto formu je však nesmierne obtiažne implementovať, nakoľko sa to ktoré dáta sú aktuálne platné mení plynule s časom, ale je to aj nepraktické riešenie, keďže pri práci s časmi platnosti sa očakáva, že užívateľ bude chcieť pravidelne zasahovať aj do minulých či budúcich dát. Z tohto dôvodu som zvolila variantu, kde hlavná tabuľka obsahuje všetky dáta ktoré obsahovala pôvodná tabuľka spolu s časmi ich platnosti. Jedná sa teda o zjednotenie oboch tabuliek použitých v pôvodnom rozšírení do jednej.

Tabuľku reprezentujúcu aktuálne platné dáta bude rozšírenie sprístupňovať pomocou náhľadu, nakoľko je potrebná len na získavanie dát a jej obsah je dynamický. Rozdiel v návrhu je zachytený v nasledovnom príklade. Príklad zachytáva zamestnanca a jeho výšku mzdy, ktorá sa mu od roku 2015 zvýšila.

- **Príklad tabuľky s časom platnosti v pôvodnom rozšírení:**
  - **tabuľka s pôvodnými dátami**

OID	Employee	Salary
1	John Doe	10 000
2	John Doe	20 000

Tabuľka 5.1: Dátová tabuľka v pôvodnom rozšírení

– **tabuľka s časmi platnosti**

OID	Valid_time
1	[1.1.2014 - 1.1.2015]
2	[1.1.2015 - 1.1.2017]

Tabuľka 5.2: Časová tabuľka v pôvodnom rozšírení

- **Príklad tých istých dát v mnou navrhovanom rozšírení:**

  – **tabuľka osahujúca všetky dáta**

Employee	Salary	Valid_time
John Doe	10 000	[1.1.2014 - 1.1.2015]
John Doe	20 000	[1.1.2015 - 1.1.2017]

Tabuľka 5.3: Súhrnná tabuľka v novom rozšírení

Na zvolenom príklade vhodne vidieť nepraktickosť originálneho návrhu. Pôvodná tabuľka obsahuje dva záznamy, ktorých význam nie je možné uchopiť bez pridania dát z druhej tabuľky, ktoré sú s nimi vo vzťahu 1:1.

V prípade tabuliek s transakčným časom je situácia zásadne iná. Tu je možné staticky zvoliť dáta ktoré sú aktuálne platné (význam platnosti je samozrejme iný, než pri tabuľkách s časom platnosti). Takisto tu existuje obmedzenie ohľadne dát ktoré môže užívateľ meniť. Konkrétne môže meniť len dáta aktuálne platné. Taktiež je pravdepodobné že najčastejším požiadavkom pri tabuľkách s transakčným časom bude výber aktuálne platných dát. Historické dáta užívatelia potrebujú prevažne z dôvodu auditu alebo obnovenia dát v prípade chybných operácií. Ďalším rozdielom oproti tabuľkám s časom platnosti je, že definícia aktuálnych dát je z pohľadu transakčného času statická. Pre tieto dôvody som zvolila schému reflektujúcu prirodzené rozloženie dát. Tabuľke, ktorá má byť označená za tabuľku s transakčným časom, bude pridaný stĺpec typu *PERIOD*, ktorý bude obsahovať interval od momentu kedy boli dáta vložené a bude z prava neobmedzený. K tejto tabuľke pribudne druhá, tzv. historická tabuľka. Na podobné riešenie je možné naraziť aj v iných projektoch venujúcich sa podobným problémom [1]. Táto bude obsahovať dáta ktoré do nej budú vložené pri každej zmene v originálnej tabuľke. Pri tomto rozložení je jednoduché nepovoliť užívateľovi omylom zasiahnuť do historických dát. Keďže schéma oboch tabuliek je rovnaká, je tiež triviálne vytvoriť na tieto dáta zjednotený pohľad. Rozdiel v návrhoch zachytím pomocou ďalšieho príkladu. Jedná sa o dáta na prvý pohľad podobné dátam z prvého príkladu, majú však inú sémantiku. Príklad zachytáva samotné zvýšenie mzdy zamestnanca v priebehu januára 2015.

- **Príklad tabuľky s transakčným časom v pôvodnom rozšírení**

  – **tabuľka s pôvodnými dátami**

OID	Employee	Salary
1	John Doe	10 000
2	John Doe	20 000

Tabuľka 5.4: Dátová tabuľka v pôvodnom rozšírení

– **tabuľka s transakčnými časmi**

OID	Valid_time
1	[1.1.2014 - 10.1.2015]
2	[10.1.2015 - ∞]

Tabuľka 5.5: Časová tabuľka v pôvodnom rozšírení

- **Príklad tých istých dát v mnou navrhovanom rozšírení:**

  – **tabuľka s aktuálnymi dátami**

Employee	Salary	Transaction_time
John Doe	20 000	[1.1.2015 - ∞]

Tabuľka 5.6: Tabuľka s aktuálnymi dátami v novom rozšírení

  – **tabuľka s historickými dátami**

Employee	Salary	Transaction_time
John Doe	10 000	[1.1.2014 – 10.1.2015]

Tabuľka 5.7: Tabuľka s historickými dátami v novom rozšírení

Na tomto príklade znova vidieť lepšiu čitateľnosť mnou zvolenej schémy. Je jednoduché pristúpiť k dátam ktoré sú aktuálne platné a teda je možné ich meniť. Takisto nie je prítomná tabuľka ktorej dáta sú samy o sebe nepoužiteľné.

Ostáva nám teda problematika bitemporálnych tabuliek. Implementácia týchto je pri mnou navrhovanom riešení relatívne priamočiara. Keďže relácia s časom platnosti sa v mojom návrhu ukladá do jedinej tabuľky, je na túto možné aplikovať rovnakú procedúru ako pri tvorbe tabuľky s transakčným časom. Rovnako tu bude platiť žiadané obmedzenie, kde užívateľ nebude môcť zasahovať do historických dát. Takúto schému znovu ilustrujem na príklade.

- **Príklad bitemporálnej tabuľky v originálnom rozšírení**

  – **tabuľka s pôvodnými dátami**

OID	Employee	Salary
1	John Doe	10 000
2	John Doe	10 000
3	John Doe	20 000

Tabuľka 5.8: Dátová tabuľka v pôvodnom rozšírení

– **tabuľka s transakčným časom a časom platnosti**

OID	Valid_Time	Transaction_Time
1	[1.1.2014 – 1.1.2017]	[1.1.2014 – 10.1.2015]
2	[1.1.2014 – 1.1.2015]	[10.1.2015 - ∞]
3	[1.1.2015 – 1.1.2017]	[10.1.2015 - ∞]

Tabuľka 5.9: Časová tabuľka v pôvodnom rozšírení

- **Príklad tých istých dát v mnou navrhovanom rozšírení:**

  – **tabuľka s aktuálnymi dátami a ich časmi platnosti**

Employee	Salary	Valid_time	Transaction_time
John Doe	10 000	[1.1.2014 – 1.1.2015]	[10.1.2015 - ∞]
John Doe	20 000	[1.1.2015 -1.1.2017]	[10.1.2015 - ∞]

Tabuľka 5.10: Tabuľka s aktuálnymi dátami v novom rozšírení

  – **Tabuľka s historickými dátami a ich časmi platnosti**

Employee	Salary	Valid_time	Transaction_time
John Doe	10 000	[1.1.2014 – 1.1.2017]	[1.1.2014 - 1.1.2015]

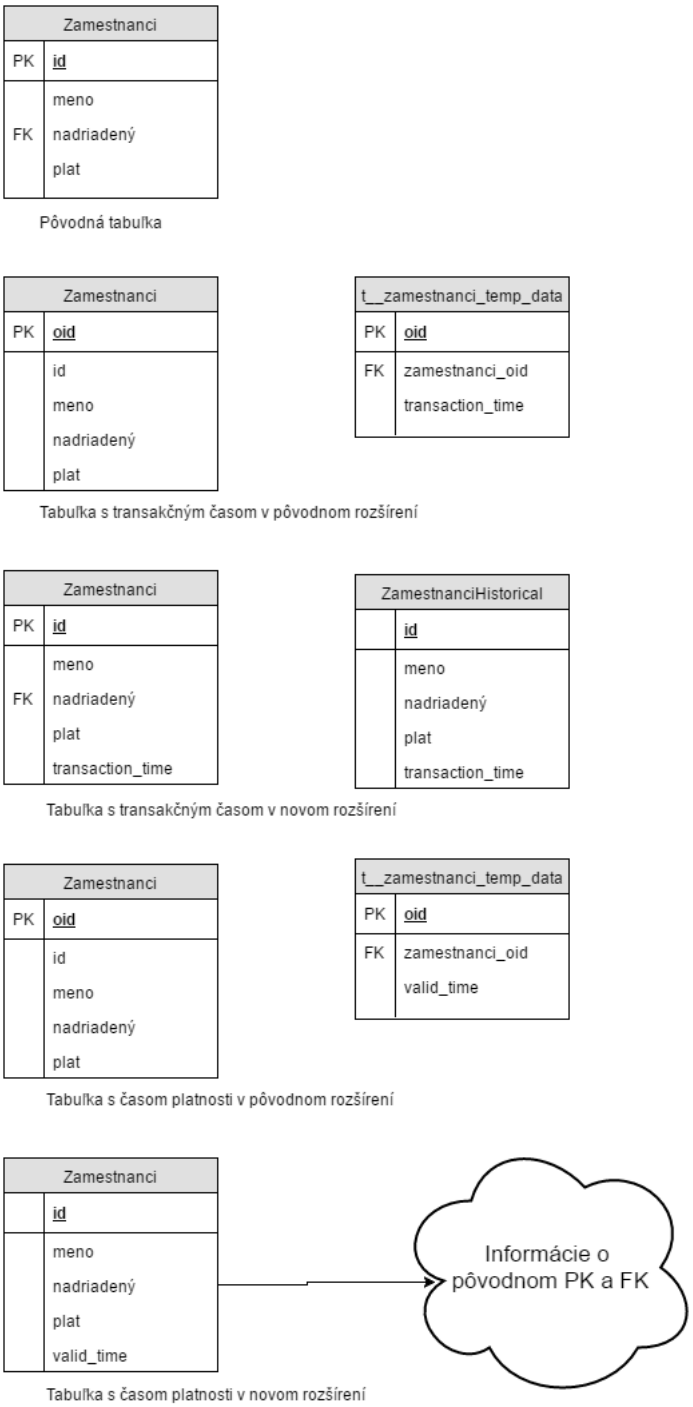
Tabuľka 5.11: Tabuľka s historickými dátami v novom rozšírení

Na záver popisu mojich zmien v schéme interne používaných tabuliek prikladám diagram pre lepšie predstavenie si rozdielov 5.1.

5.4 Referenčná integrita

Odlišný návrh popísaný v predošlých odstavcoch má zásadný vplyv aj na implementáciu podpory referenčnej integrity. Významne sa zmena týka najmä referenčnej integrity pri tabuľkách s transakčným časom. V mojej implementácií s pomocou reálnej a historickej tabuľky je možné zabezpečiť referenčnú integritu medzi tabuľkou s transakčným časom a štandardnou tabuľkou, a to obojsmerne. V prípade, že užívateľ zmaže údaj z tabuľky ktorá podporuje transakčný čas, je tento riadok zmazaný vo svojej pôvodnej tabuľke a je presunutý do historickej. Túto zmenu PostgrreSQL štandardne zaregistruje a mazanie buď zakáže, alebo kaskádovito zmaže odkazujúce sa riadky podľa nastavenia.

Pri tabuľke s časmi platnosti situácia nie je tak jednoduchá a vyžaduje bližší pohľad. Prvá varianta, tabuľka s časom platnosti odkazujúca sa na štandardnú tabuľku, je relatívne bezproblémová, štandardná obsluha referenčnej integrity sa o túto situáciu postará. Druhá situácia, kedy sa štandardná tabuľka odkazuje na tabuľku s časom platnosti je o niečo viac problematickou, keďže na strane odkazovanej tabuľky neexistuje originálny primárny kľúč. Z tohto dôvodu by teda bolo nutné takúto podporu doimplementovať. Keď sa však zamyslíme nad povahou tohto vzťahu, je ťažko uchopiteľné čo by tento vzťah mal reflektovať. Ak považujeme dáta v klasickej tabuľke za dáta, ktoré sú vždy platné bolo by nutné vynucovať odkazy len na také záznamy, ktoré majú spojený čas platnosti neobmedzený. Toto je však extrémne obmedzujúce riešenie a ako také ho považujem za zbytočné implementovať. V prípade potreby takéhoto odkazu je ho samozrejme možné pridať samotným užívateľom na základe konkrétneho požiadavku. Najproblematickejším a zároveň hlavným problémom sú



Obr. 5.1: Schéma demonštrujúca rozdiely v schéme oproti pôvodnému rozšíreniu

však odkazy z tabuľky s časom platnosti do tabuľky s časom platnosti. Tu je totiž ich sémantika priamočiara bez nejakých zásadne iných možností implementácie. Referenčná intergrita v prípade odkazov z tabuľky s časom platnosti do tabuľky rovnakého typu je platná, pokiaľ spojená platnosť záznamov odkazovanej enity pokrýva spojenú platnosť záznamu ktorý sa odkazuje. Tento koncept bližšie priblížim na príklade.

- Tabuľka zamestnancov s časmi platnosti

Employee	Salary	Valid_time
John Doe	10 000	[1.1.2014 – 1.1.2015]
John Doe	20 000	[1.1.2015 -1.1.2017]

Tabuľka 5.12: Odkazovaná tabuľka s časom platnosti

- Tabuľka priradení zamestnancov na jednotlivé pracoviská, odkazujúca sa na tabuľku zamestnancov.

Department	Employee	Valid_time
Marketing	John Doe	[1.1.2014 – 31.11.2015]
Sales	John Doe	[1.8.2015 -1.1.2017]

Tabuľka 5.13: Odkazujúca sa tabuľka s časom platnosti

Tabuľky sú v tomto prípade vo validnom stave, nakoľko časové rozsahy v ktorých sa druhá tabuľka odkazuje na prvú sú pokryté časovými rozsahmi prítomnými v prvej tabuľke. Môžeme teraz rozobrať jednotlivé situácie, kedy by táto integrita mohla byť narušená [13].

- Vloženie nového riadku do odkazujúcej sa tabuľky – užívateľ by mohol pridať riadok odkazujúci sa na rodičovskú tabuľku s neplatným intervalom, napr.:

Department	Employee	Valid_Time
Marketing	John Doe	[1.1.2013 – 31.11.2015]

Tabuľka 5.14: Príklad na porušenie temporálnej referenčnej integrity

- Zmena hodnoty riadku v odkazujúcej sa tabuľke - užívateľ by mohol príkazom *UPDATE* zmeniť ľubovoľný z riadkov tabuľky na riadok podobný riadku v predošlom príklade.
- Zmazanie riadku v odkazovanej (rodičovskej) tabuľke – v prípade zmazania ľubovoľného z riadkov v našej tabuľke zamestnancov by spôsobil nevalidnosť tabuľky oddelení.
- Zmena času platnosti v odkazovanej tabuľke - obdobne, ak by sme ktorýkoľvek z intervalov skrátili alebo presunuli, dáta v tabuľke oddelení by sa stali nevalidnými.

Tieto situácie sú jedinými, ktoré môžu nevalidnosť referenčnej integrity spôsobiť a z ich povahy vyplýva, že je možné ich implementovať pomocou niekoľkých triggerov.

Pri problematike referenčnej integrity je okrem udržovania korektnosti stavu databázy podstatná aj forma výberu dát z nej. Jazyk SQL pre tento účel poskytuje konštrukciu *JOIN*. V prípade temporálneho rozšírenia poskytol autor relatívne komplikovanú konštrukciu pre výber dát zo spojených tabuliek. V mojom rozšírení by som sa rada vyhla zbytočným konštrukciám, ktoré prácu s jazykom výrazne nezjednodušujú. Na príkladoch si ukážeme, že tieto formy spojenia je možné použiť priamo, bez pomocných funkcií s pomocou štandardnej a dobre čitateľnej syntaxe.

- **Príklad spojenia tabuliek z predchádzajúceho príkladu s pomocou existujúceho rozšírenia:**

```
SELECT addTemporalJoin(array('Employees', 'Departments'));
SELECT Department, Employee, Salary, valid_time FROM
JoinEmployeesDepartmentsSequenced(true);
```

- **Príklad spojenia tabuliek z predchádzajúceho príkladu v mojom návrhu**

```
SELECT Department, Employee, Salary,
PERIOD_INTERSECT(e.valid_time, d.valid_time)
FROM Employee e JOIN Department d ON e.Employee = d.Employee
AND OVERLAPS(e.valid_time, d.valid_time)
```

Tento príklad ilustruje zásadný rozdiel medzi pôvodným a navrhovaným rozšírením. Zatiaľ čo sa samotná dĺžka zápisu nezmenila, nový zápis sa skladá len z jediného príkazu *SELECT* a dovoľuje užívateľovi meniť jednotlivé podmienky spojenia, napríklad kontrolu na presnú zhodu rozsahov, prípadne pokrytie jedného druhým. Taktiež, ako bolo spomenuté v analýze výkonnosti príkazov, vo variante bez volania funkcie je PostgreSQL schopné použiť indexy, prípadne si výber dát zjednodušiť vhodným poradím aplikácie podmienok.

5.5 Agregačné funkcie

5.5.1 Implementácia agregačnej funkcie sequenced

Pri popise môjho návrhu rozšírenia som spomenula existenciu agregačnej funkcie *sequenced*. Táto je nutná pre prácu s dátami s časom platnosti tak, ako to umožňuje jazyk TSQL2. Malo by ísť o funkciu ktorá pre zvolené stĺpce podľa ktorých je výsledok zgrupovaný agreguje ich časy platnosti do poľa zliatych intervalov. Ideálne by sa toto pole malo ďalej rozpadnúť do riadkov podľa jeho jednotlivých elementov. Nakoľko sa jedná o komplikovanejší koncept, priblížim ho na príklade.

- **Vzorová tabuľka s dátami:**

Employee	Salary	Manager	Valid_time
John Doe	10 000	False	[1.1.2014 -1.1.2015]
John Doe	20 000	False	[1.2.2015 -1.1.2016]
John Doe	20 000	True	[1.1.2016 -1.1.2017]

Tabuľka 5.15: Plný obsah tabuľky s časom platnosti

- **Výber mena a platu z tabuľky**

```
SELECT
 Employee, Salary, sequenced(Valid_time)
FROM
 Employees
GROUP BY
 Employee, Salary
```

Employee	Salary	Valid_time
John Doe	10 000	[1.1.2014 -1.1.2015]
John Doe	20 000	[1.2.2015 -1.1.2017]

Tabuľka 5.16: Príklad na výber dát s použitím agregačnej funkcie

- **Výber mena a pozície z tabuľky**

```
SELECT
 Employee, Manager, sequenced(Valid_time)
FROM
 Employees
GROUP BY
 Employee, Manager
```

Employee	Manager	Valid_time
John Doe	False	[1.1.2014 -1.1.2015]
John Doe	False	[1.2.2015 -1.1.2016]
John Doe	True	[1.1.2016 -1.1.2017]

Tabuľka 5.17: Príklad na výber špecifickejších dát s použitím agregačnej funkcie

Na zvolenom príklade je vidieť rozdiel v situácií, keď je možné intervaly spojiť do jedného a keď toto možné nie je.

Realizácia tejto funkcie je možná s pomocou už existujúcich, vstavaných, funkcií jazyka PostgreSQL a funkcie *t___restructualize* ktorá je súčasťou originálneho rozšírenia. Táto funkcia berie ako vstup pole intervalov a jej výstupom je pole intervalov a výstupom je pole intervalov vzniknutých postupným zjednocovaním prekrývajúcich sa intervalov.

Táto funkcia v kombinácií so vstavanou funkciou *array_agg* umožňuje dosiahnuť želané správanie. Funkcia *array_agg* je agregačnou funkciou, ktorá všetky agregované hodnoty spojí do jednej hodnoty formou poľa týchto prvkov. Týmto spôsobom sme schopní pokryť situácie analogické s príkladom 5.16.

V príklade 5.17 však potrebujeme pole vzniknuté zlievaním intervalov rozdeliť do viacerých riadkov. Pre tento účel poskytuje PostgreSQL funkciu *unnest*, táto je schopná rozložiť riadok na viaceré riadky práve na základe hodnôt prvkov poľa.

5.5.2 Ďalšie agregačné funkcie

V rámci práce na rozšírení pridávam aj ďalšie agregačné funkcie, nakoľko tieto nie sú súčasťou používaného rozšírenia pridávajúceho dátový typ *PERIOD*. Okrem funkcie popísanej v predchádzajúcom odstavci, ktorá je nutná pre samotnú funkcionalitu rozšírenia v súlade s funkčnosťou TSQL2 je pridaná dvojica funkcií vracajúcich prienik všetkých agregovaných intervalov, resp. ich obal. Koncept prieniku je intuitívny a funkciou obalu sa tu myslí najmenší taký interval, ktorý obsahuje všetky agregované intervaly. Ďalej pridávam funkcie ktoré reprezentujú obe intuitívne varianty konceptu maxima a minima, teda funkciu ktorá vracia interval s najmenším časom začiatku, funkciu ktorá vracia interval s najmenším časom konca a rovnakú sadu funkcií pre maximálne hodnoty. Tieto funkcie označujem ako *min_start*, *min_end*, *max_start* a *max_end* .

Nasleduje pár príkladov použitia týchto funkcií.

- Kedy sme prvý krát najali manažéra

```
SELECT
 start(min_start(valid_time))
FROM
 Employees;
```

- Kedy prebehla prvá zmena zamestnanca v systéme

```
SELECT
 end(min_end(valid_time))
FROM
 Employees;
```

5.6 Inštalácia a spustenie

Rozšírenie PostgreSQL ktoré som v rámci práca vyvinula je funkcionalitou nezávislé na rozšírení Radka Jelínka. Toto riešenie som použila pri zrovnávaní a počiatočnom návrhu. Funkčne sa však rovnako ako pôvodný autor spolieham na rozšírenie pridávajúce dátový typ *PERIOD* [5]. Do systému kde je toto rozšírenie nainštalované sa moje rozšírenie zavedie jednoduchým spustením súboru *temporal-setup.sql*. Po inštalácií je možne funkcionalitu rozšírenia overiť spustením testovacieho súboru *examples.sql*, ktorý demonštruje použitie jednotlivých prvkov rozšírenia na realistických príkladoch. V rámci projektu je priložený aj súbor s dokumentáciou: *doc.html*.

5.7 Dostupnosť a licencovanie rozšírenia

Zdrojové kódy ako aj príklady jednotlivých príkazov som sa rozhodla zverejniť na platforme BitBucket ako verejne dostupný projekt. V prípade záujmu je dostupný na adrese: https://bitbucket.org/account/user/dominikakoroncziova/projects/TP. Ako licenciu som zvolila open source licenciu GPLv3, nakoľko je bežne používaná a populárna v podobných projektoch.

Kapitola 6

Testovanie temporálneho rozšírenia

V tejto kapitole sa zameriam na praktické použitie novej verzie temporálneho rozšírenia. Kapitola by mala slúžiť viacerým účelom. V prvom rade by mala slúžiť ako súhrnné vizuálne zrovnanie používania pôvodného a nového temporálneho rozšírenia. V druhom rade by mali tabuľky a schémy v nej použité slúžiť ako základ pre testovanie výkonu v prípade, keď to bude možné (pôvodné rozšírenie nepokrýva niektoré prípady použitia). V poslednom rade by som rada koncipovala testovacie príkazy ako základ eventuálnej dokumentácie, nakoľko by mali demonštrovať jednotlivé možnosti práce spolu so vzorovými riešeniami tradičných úloh.

6.1 Príprava temporálnej schémy

V prvom rade bude nutné pripraviť používanú schému. Aby boli príklady jednoducho predstaviteľné, zameriame sa na zjednodušenú situáciu informačného systému starajúceho sa o správu zamestnancov. V prvom rade bude náš systém obsahovať tabuľku zamestnancov. Táto bude tabuľkou bitemporálnou, keďže má zmysel tieto dáta tak verzovať (transakčný čas - audit) ako aj ukladať informácie o minulých a budúcich zmenách (čas platnosti – zmena platu od budúceho mesiaca a pod.). Tabuľka zamestnancov bude obsahovať vzťah na seba samú, reprezentujúc vzťah podriadený-nadriadený, bude teda demonštrovať referenčnú integritu pri odkazovaní tabuľky s časom platnosti na tabuľku s časom platnosti.

Títo zamestnanci budú môcť byť zadelení do týmov. Týmy budú reprezentované jednoduchou tabuľkou v podobe číselníku a bude existovať vzťah m-n medzi entitami zamestnanec a tým. Tento vzťah bude mať svoj vlastný čas platnosti a bude reprezentovaný samostatnou mapovacou tabuľkou. Zatriedenie ľudí do týmov sa bežne v čase mení a nás zaujíma ako vyzerali alebo budú vyzerať jednotlivé týmy v ľubovoľnom čase. Už však nejde o kritickú tabuľku z pohľadu HR a preto nie je nutné udržovať jej transakčný čas.

Poslednou tabuľkou použitou v demonštrácií bude tabuľka reprezentujúca dovolenku ktorú zamestnanci čerpajú. Táto tabuľka bude obsahovať dátum, odkaz na zamestnanca, dĺžku trvania a komentár. Tabuľka nebude obsahovať čas platnosti, nakoľko nemá zásadný zmysel zamýšlať sa nad stavom dovoleniek v čase. Naopak, bude podporovať transakčný čas z dôvodu auditovateľnosti, keďže na dovolenke závisia mzdy.

6.1.1 Zápis definície schémy v novom rozšírení

```
CREATE TABLE Employees (
Name TEXT PRIMARY KEY,
```

```
Salary INTEGER,
Supervisor TEXT REFERENCES Employees
);

SELECT makeTableBitemporal('employees');

CREATE TABLE Teams (
Name TEXT PRIMARY KEY
);

-- Spôsob tvorby m:n vzťahu v novej verzii rozšírenia
CREATE TABLE TeamEmployeeMap (
Team TEXT REFERENCES Teams,
-- Tabuľka employees je už temporálna, nemôžeme preto použiť štandardný odkaz
Employee TEXT
);

SELECT addValidTime('teamemployeemap');

SELECT addValidTimeReference('teamemployeemap', 'employees', '{employee}');

--Tvorba tabuľky s transakčným časom je obdobná
CREATE TABLE TimeOffs (
Employee TEXT,
TimeOffDate DATE,
Comment TEXT,
Hours int
);

SELECT Add ('timeoffs');
```

Na uvedených príkladoch nie sú viditeľné zásadné rozdiely v zápise, dobre však ilustrujú problematickosť definície vzťahov. Pôvodné rozšírenie vyžaduje aby obe strany vzťahu boli rovnakého typu, čo je v praxi zásadne obmedzujúce. Tiež je možné pozorovať nemožnosť použitia pôvodných primárnych kľúčov, nakoľko tieto sú nahradené OID (prípadne kompozitnými kľúčmi).

6.1.2 Zápisy DML príkazov

```
-- Vloženie zamestnanca do bitemporálnej tabuľky v novom rozšírení
INSERT INTO
    Employee
VALUES
    ('John Doe', 10000, NULL,
     PERIOD('1-1-2015', 'infinity'::TIMESTAMP));

-- Vloženie zamestnanca do bitemporálnej tabuľky v originálnom rozšírení
SELECT
```

```
    *
FROM
    insertValidTime(
            'INSERT INTO
                zamestnanec
            VALUES
                ('John Doe',10000, NULL)',
                 PERIOD('1-1-2015', 'infinity'::TIMESTAMP));

-- Vloženie zamestnanca s odkazom na iného zamestnanca,
garantujúc temporálnu referenčnú integritu
INSERT INTO
    Employee
VALUES
    ('John Minion', 10000, 'John Doe',
     PERIOD('1-1-2016', 'infinity'::TIMESTAMP));

-- Vloženie zamestnanca v originálnom rozšírení,
neobsahuje garanciu temporálnej referenčnej integrity
SELECT
    *
FROM
    insertValidTime('
        INSERT INTO
            zamestnanec
        VALUES
            ('John Minion', 10000, 'John Doe')',
             PERIOD('1-1-2016', 'infinity'::TIMESTAMP));

-- Zmena hodnoty atribútu pre konkrétny interval v oboch rozšíreniach
INSERT INTO
    Employee
VALUES
    ('John Doe', 20000, NULL, PERIOD('1-2-2016', 'infinity'::TIMESTAMP));

SELECT
    *
FROM
    insertValidTime('
INSERT INTO
    zamestnanec
VALUES
    ('John Doe',20000, NULL)', PERIOD('1-2-2016', 'infinity'::TIMESTAMP));

-- Vytvorenie temporálneho vzťahu: priradenie zamestnanca do týmu
INSERT INTO
    Teams
VALUES
```

```
    ('Team A');

INSERT INTO
    TeamEmployeeMap
VALUES
    ('Team A', 'John Doe', PERIOD('1-1-2015', 'infinity'::TIMESTAMP));

-- Úprava platnosti temporálneho vzťahu
UPDATE
    TeamEmployeeMap
SET
    valid_time = PERIOD('1-3-2016', '1-4-2016')
WHERE
    Team = 'Team A' AND
    Employee = 'John Expensive',

-- Vloženie dát do tabuľky s temporálnym časom pre oboje varianty rozšírenia
INSERT INTO
    TimeOffs
VALUES
    ('John Doe', '2016-01-05'::DATE, 'Dovolenka', 8);

-- Zmena hodnoty atribútu v tabuľke s transakčným časom
UPDATE
    TimeOffs
SET
    Note = 'Choroba'
WHERE
    Employee = 'John Doe' AND
    TimeOffDate = '2016-01-05'::DATE;
```

Na príkladoch DML príkazov je znovu vidieť najmä problémy pôvodného rozšírenia v oblasti temporálnej referenčnej integrity. Samotný zápis príkazov, ako je možné vidieť, je dĺžkou podobný, avšak zbavil sa viecerých nadbytočných volaní funkcií.

6.2 Príkazy pre dotazovanie dát

Nakoľko sú príkazy pre dotazovanie dát o niečo komplikovanejšie než ostatok testovaných príkazov, budem v tejto kapitole komentovať jednotlivé príklady bližšie. Na úvod prejdeme základné výbery dát z jednotlivých typov tabuliek.

1. **Stav tabuľky s transakčným časom ku konkrétnemu dátumu – stav tabuľky dovoleniek ku koncu kalendárneho roka**

   ```
   -- Nové rozšírenie
   SELECT
       Employee, TimeOffDate, Hours
   FROM
   ```

```
    TimeOffsHistory
WHERE
    contains(transaction_time, '2016-01-01'::DATE);

-- Pôvodné rozšírenie
SELECT
    Employee, TimeOffDate, Hours
FROM
    sequencedTimeOff(true, '2016-01-01'::DATE);
```

Na príklade je vidieť že aj pri absencii špecializovanej funkcie ostáva príkaz príjemne čitateľný. Na oplátku získava väčšiu flexibilitu pri zadávaní podmienok a umožňuje použitie indexov v prípade potreby. Viac si o tomto prípade povieme pri analýze výkonnosti.

2. **Výber dát z bitemporálnej tabuľky pomocou obmedzenia oboch časov – zoznam zamestnancov ktorý mali plánovaný nástup v priebehu roku 2016 už na začiatku roka**

```
-- Nové rozšírenie
SELECT
    DISTINCT Name
FROM
    Employee
WHERE
    first(transaction_time) < '2016-01-01'::DATE AND
    first(valid_time) > '2016-01-01'::DATE;

-- Pôvodné rozšírenie
-- Nepostačujúce vyjadrovacie možnosti
```

Tento relatívne jednoduchý dotaz s priamočiarou sémantikou ilustruje funkčné limity pôvodného rozšírenia, kde funkcie umožňujú len kontrolu prítomnosti dátumu v danom rozsahu. Porovnania tohto druhu však v realistických prípadoch vystupujú.

3. **Použitie temporálnych spojení – výber ľudí ktorí nastúpili a nemali zaradenie do týmu**

```
-- Nové rozšírenie
SELECT
    DISTINCT Name
FROM
    (SELECT
        Name, SEQUENCED(valid_time) valid_time
    FROM
        Employee
    GROUP BY
        Name) e
```

```
JOIN
    (SELECT
        Employee, SEQUENCED(valid_time) valid_time
    FROM
        TeamEmployeeMap
    GROUP BY
        Employee) tem
WHERE
    e.Name = tem.Employee AND
    first(e.valid_time) < first(tem.valid_time)

-- Pôvodné rozšírenie znovu nedovoľuje zachytiť takýto dotaz
```

4. **Komplikovanejší dotaz s temporálnymi dátami - Všetky intervaly kedy niekto pracoval v jednom týme so svojim manažérom**

```
-- Nové rozšírenie
SELECT
    e.Name, e.Supervisor, tem.Team,
    PERIOD_INTERSECT(tem.valid_time, tsm.valid_time)
FROM
    (SELECT
        Name, Supervisor, SEQUENCED(valid_time)
    FROM
        Employee
    GROUP BY
        Name, Supervisor) e
JOIN
    (SELECT
        Employee, SEQUENCED(valid_time)
    FROM
        TeamEmployeeMap
    GROUP BY
        Employee) tem
ON
    e.Name = tem.Employee
JOIN
    (SELECT
        Employee, SEQUENCED(valid_time)
    FROM
        TeamEmployeeMap
    GROUP BY
        Employee
    ) tsm
ON
    e.Supervisor = tsm.Employee
WHERE
    tsm.Team = tem.Team AND
```

```
    OVERLAPS(tem.valid_time, tsm.valid_time)

-- Pôvodné rozšírenie znovu nedovoľuje zachytiť takýto dotaz
```

Na príkladoch 3 a 4 je okrem ťažkopádnosti pôvodného rozšírenia vidieť opakujúci sa vzor vo formáte *SELECT DISTINCT a,b,..., sequenced(valid_time) FROM X GROUP BY a,b,....* Toto je výber podmnožiny stĺpcov v podobe temporálnej relácie. Prostredie PostgreSQL nám nijako významne neuľahčuje prevedenie tejto redundancie do kratšej podoby. Konštrukcie *VIEW* nie sú parametrizovateľné a funkcie musia ako návratový typ definovať konkrétny typ záznamu (tj. konkrétne stĺpce prítomné vo výsledku). Pôvodné rozšírenie tento zásadný problém nijak nerieši, poskytuje len funkcie pre vrátenie všetkých stĺpcov zároveň. Táto funkčnosť je však výrazne nepostačujúca pre praktické príklady.

5. **Iné operácie s časovými úsekmi – priemerné platy v jednotlivých týmoch ku dnešnému dátumu**

```
-- Nové rozšírenie
SELECT
    Team, AVG(salary)
FROM (
    SELECT
        e.Name, e.Salary, tem.Team,
        PERIOD_INTERSECT(e.valid_time, tem.valid_time) valid_time
    FROM
        Employee e
    JOIN
        TeamEmployeeMap tem
    ON
        e.Name = tem.Employee AND
        OVERLAPS(e.valid_time, tem.valid_time)
    WHERE
        contains(PERIOD_INTERSECT(e.valid_time, tem.valid_time),
        now()::DATE)
)
GROUP BY
    Team

-- Pôvodné rozšírenie znovu nedovoľuje zachytiť takýto dotaz
```

Tento príkaz je ukážkou použitia odlišných operátorov pre prácu s časovými intervalmi. Dotaz s tou istou sémantikou je pravdepodobne možné zapísať aj iným spôsobom, tento som však zvolila kvôli ilustratívnosti.

6.3 Zrovnanie výkonnosti temporálnych rozšírení

Táto časť textu bude venovaná zrovnaniu výkonnosti mnou vyvinutého rozšírenia s výkonnosťou rozšírenia ktoré vyvinul Radek Jelínek. Začnem popisom metodológie, nakoľko tento je nutný pre korektnú interpretáciu výsledkov.

Všetky testy som vykonávala na svojom osobnom počítači (procesor Intel i7, 8GB RAM), pod operačným systémom Mac OS X. Pri behu dotazov nebežali žiadne ďalšie významne výkon vyžadujúce aplikácie. Pre spustenie príkazov som použila prostredie *pg_admin*. Všetky merania som opakovala a zaznamenávala priemerné hodnoty. Pri všetkých príkazoch typu *SELECT* som tiež ignorovala prvé spustenie príkazu, nakoľko si v rámci neho databázový systém prepočítava exekučný plán a tento čas ním môže byť skreslený.

Ďalšou dôležitou súčasťou je voľba scenárov. Tie budem bližšie popisovať pri jednotlivých meraniach, ale vo všeobecnosti som sa tu zamerala na minimalistické situácie s minimom nutných premenných aby tieto zbytočne neovplyvňovali výsledky. Pri scenároch bolo tiež nutné prihliadať na výrazné funkčné nedostatky pôvodného rozšírenia. V niekoľkých oblastiach výrazne zaostávali za možnosťami novej implementácie.

6.3.1 Výkonnosť pri práci s tabuľkami s transakčným časom

Práca s tabuľkami s transakčným časom prebieha v oboch rozšíreniach pre užívateľa relatívne podobným spôsobom. Pri vkladaní a úprave dát sa totiž pracuje s tabuľkou rovnakým spôsobom ako s klasickou SQL relačnou tabuľkou. Tabuľka s ktorou som pracovala mala len 2 stĺpce, jeden reprezentujúci primárny kľúč, tj. id, a druhý, ktorý reprezentoval číselnú hodnotu ktorá sa v čase môže meniť. Tento som pre ľahkú prirovnateľnosť k reálnej situácií nazvala plat.

Príkazy pre zmenu dát

Prvým testovateľným krokom bolo naplnenie tabuľky prvotnými dátami. Ako základnú dátovú sadu som zvolila 5 000 riadkov reprezentujúcich 5 000 unikátnych entít. Tieto som do tabuľky vložila pomocou 5 000 samostatných príkazov typu *INSERT*.

	Pôvodné rozšírenie	Nové rozšírenie
Vložnie 5 000 záznamov	2 646 ms	2 508 ms

Tabuľka 6.1: Dĺžky trvania vkladu dát to tabuliek s transakčným časom

Ako je na dátach možné vidieť, vloženie dát do vopred prázdnej tabuľky je porovnateľne rýchle. Toto je v súlade s očakávaniami, nakoľko žiadna nadštandardná logika by sa počas týchto krokov nemala vykonávať.

Ďalšou operáciou ktorú som zvolila je postupná zmena hodnoty atribútu, ktorá spôsobí uloženie existujúceho vzťahu. Každý z riadkov som zmenila 4 krát, čo v konečnom výsledku znamená 20 000 operácií typu *UPDATE*. Tento scenár sa však ukázal byť nerealizovateľným pre pôvodnú verziu rozšírenia, ktoré v prípade, že jedna databázová transakcia vykoná 2 zmeny v jednom zázname skončí výnimkou. Preto som bola scenár nútená upraviť do podoby, keď mení každý z riadkov práve raz.

	Pôvodné rozšírenie	Nové rozšírenie
Update 5 000 riadkov	20 198 ms	5 747 ms

Tabuľka 6.2: Dĺžky trvania zmien dát v tabuľkách s transakčným časom

Tak ako v predošlom príklade sa jedná o priemernú hodnotu nameranú počas piatich behov. Hodnoty sa v rámci jedného typu líšili maximálne v stovkách milisekúnd, čo naznačuje skutočne výrazný rozdiel v spracovaní dát. Najpravdepodobnejším dôvodom takto

výrazného rozdielu je architektúra tabuliek, kde uložením všetkých entít v jednej tabuľke nútime databázový systém túto tabuľku zbytočne prehľadávať. Pokúšala som sa otestovať aj rýchlosť mazania dát, avšak v tejto situácií znovu pôvodné riešenie skončilo chybovým stavom.

Vo všeobecnosti je možné moje zmeny v príkazoch pre zmenu dát tabuľky s temporálnym časom prehlásiť za úspešné. Nielenže je nový systém schopný vykonávať operácie ktoré v pôvodnom systéme končili chybovými stavmi, ale aj namerané 4-násobné zrýchlenie pri zmene dát je extrémne pozitívnou zmenou.

Príkazy pre dotazovanie dát

Pri testovaní príkazov pre dotazovanie dát som ešte ďalej upravila testovaciu sadu z predošlej kapitoly a postupne pridala pôvodne zamýšľaných 20 000 zmien riadkov. Toto som docielila falošným prepísaním databázovej funkcie *now()*. Vo výledku teda temporálna relácia obsahovala 5 000 aktuálne platných informácií a 20 000 v minulosti platných informácií. Čo sa testovaných dotazov týka, zvolila som nasledovnú kombináciu príkazov. V prvom rade som skúšala výber nejakej agregovanej hodnoty pre fakty platné niekde v minulosti (konkrétne maximálnu hodnotu atribútu). Ako druhú som skúsila voľbu historickej informácie v spojení s nejakým obmedzením na nečasový stĺpec (id mod 3 == 0).

Posledným testovaným príkazom bola voľba rovnakej informácie, ale z aktuálnych namiesto historických dát. Práve výber z aktuálnych dát je podľa očakávaní najbežnejšou variantou práce stabuľkou s transakčným časom. Pre všetky príkazy som znovu namerala 5 hodnôt, z ktorých tu uvádzam priemernú hodnotu. Pre zber všetkých hodnôt som neuvažovala prvý beh, ktorý bol podľa očakávaní pomalší (nakoľko je v ňom započítaná tvorba exekučného plánu).

	Pôvodné rozšírenie	Nové rozšírenie
Výber agregácie historických dát	81 ms	51 ms
Agregácia historických dát s podmienkou	57 ms	36 ms
Výber agregácie aktuálnych dát	55 ms	42 ms

Tabuľka 6.3: Dĺžky trvania jednotlivých príkladov na výber dát

Letmý pohľad na namerané hodnoty naznačuje, že nová implementácia výkonnosť pôvodnej implementácie definitívne nezhoršila a s najväčšou pravdepodobnosťou ju jemne vylepšila. Je však nutné podotknúť, že tieto merania sú do značnej miery ovplyvnené zvolenou metodológiou a je možné, že pri iných špecificky zvolených dátach bude rozdiel väčší, prípadne zanikne. Dáta som sa však snažila voliť čo najviac minimalistické, takže je možné predpokladať že reflektujú skutočnú povahu implementácií.

Zaoberala som sa aj možnosťou skúmať vytvorené exekučné plány oboch rozšírení, avšak kvôli nutnosti používať volania funkcií v pôvodnom rozšírení je ťažké tieto priamo porovnávať. Analýza exekučného plánu pôvodného rozšírenia je popísaná v odseku 3.8. Vychádzajúc z jej záverov by som však výsledky tohto merania označila za zodpovedajúce očakávaniam.

6.3.2 Výkonnosť pri práci s tabuľkami s časom platnosti

Pri práci s tabuľkami s časom platnosti je nutné na začiatok popísať funkčné rozdiely pri práci s jednotlivými rozšíreniami. Zatiaľ čo sa mnou implementované nové rozšírenie stará o priamu nadväznosť jednotlivých dát, v pôvodnom rozšírení je táto starosť ponechaná

na užívateľovi. Jediná špeciálna funkcionalita ktorú pôvodné rozšírenie ponúka je zmena času platnosti nejakej podmnožiny riadkov. Užívateľ si teda pri vložení nového faktu musí najprv sám overiť či sa tento interval neprekrýva s existujúcimi intervalmi a podľa potreby si ich upraviť. Toto môže byť v niektorých situáciách výrazne komplikované, napr. v prípade rozdelenia existujúceho intervalu na dva.

V mojej implementácií som do práce s časom platnosti zaviedla koncept „primárneho kľúča“, ktorý slúži ako identifikátor jednotlivých entít obsiahnutých v tabuľke. Na jeho základe je možné automaticky detekovať riadky ktoré je treba upraviť aby umožnili vloženie nového údaju bez porušenia časovej línie. Bez tejto vlastnosti je užívateľ nutný robiť úpravy v tabuľke platnosti v rámci troch samostatných krokov. Ako prvé si musí sám dohľadať dáta ktoré novo vkladaný záznam ovplyvňuje, následne musí tieto zánamy vhodne upraviť aby neprekážali novo vkladanému záznamu, čo ako spomínam vyššie nemusí byť triviálne. Ako posledné musí skutočne vložiť žiadaný záznam.

Príkazy pre zmenu dát

Vyššie popísaný zásadný rozdiel v možnostiach jednotlivých rozšírení je podstatný práve pri príkazoch pre zmenu dát. Mnou pôvodne zamýšľaný plán pre porovnanie výkonnosti týchto rozšírení pri práci s dátami s časom platnosti obsahoval postupné zadávanie zmien v časoch platnosti tak, aby sa demonštrovalo postupné skracovanie, posúvanie a prípadné ignorovanie pôvodných časových rozsahov. Bohužiaľ v rámci funkcionality poskytovanej pôvodným rozšírením je toto prakticky nemožné, preto som zvolila jednoduchšiu variantu, kde v prvom kroku vložíme do tabuľky s časom platnosti 15 000 záznamov rovnakého typu a následne im zmeníme hodnotu v intervale ktorý sa s pôvodnými intervalmi prekrýva. Toto v našom rozšírení znamená postupné volanie príkazov *INSERT* a použitia primárneho kľúča. V prípade pôvodného rozšírenia sa jedná o vloženie záznamov, následné upravenie existujúcich záznamov a vloženie nových záznamov. V reálnych aplikáciách by medzi tymito krokmi musela existovať sekcia starajúca sa o nájdenie správneho rozdelenia intervalu a prípadné ošetrenie komplikovanejších delení. Z tohto dôvodu budú tu uvedené výsledky jemne skreslené v prospech pôvodného rozšírenia. Začneme podobne ako v prípade transakčného času vložením záznamov, v tomto prípade pracujeme s 15 000 záznamami.

	Pôvodné rozšírenie	Nové rozšírenie
Vloženie 15 000 záznamov s časom platnosti	35 642 ms	9 427 ms

Tabuľka 6.4: Dĺžky trvania vkladu dát to tabuliek s časom platnosti

Na rozdiel od transakčného času môžeme vidieť zásadný rozdiel v nameraných hodnotách už pri prvotnom vkladaní záznamov. Potrebná doba je v prípade nového rozšírenia až takmer 4 krát nižšia. Najpravdepodobnejším zdrojom tohto rozdielu je zbavenie sa nutnosti používať volania funkcií, a obsluha volaní pomocou jednoduchých triggerov.

Ako druhý údaj sa pozrieme na čas nutný na vykonanie zmien, ktoré spočívajú v posunutí času platnosti každého záznamu a vložení nového záznamu. Ako som vyššie spomínala, na strane pôvodného rozšírenia vynechávam zisťovanie hraníc intervalov na skrátenie.

	Pôvodné rozšírenie	Nové rozšírenie
Pridanie 15 000 nových záznamov ktoré menia časy platnosti pôvodných	226 666 ms	13 113 ms

Tabuľka 6.5: Dĺžky trvania zmien dát v tabuľkách s časom platnosti

Ako môžeme na nameraných hodnotách vidieť, jedná sa o najvýraznejšiu nameranú zmenu. Tento výrazný rozdiel bude pravdepodobne spôsobený viacerými faktormi. Jedným je už spomínaná nutnosť opakovaného volania funkcií. Druhým vplyvom bude pravdepodobne fakt, že originálne riešenie na prácu používa dve samostatné tabuľky, čím sa znásobuje potrebný počet operácií.

Posledným možným faktorom je veľmi špecifický návrh funkcie pre zmenu intervalu platnosti. Táto je ako keby pripravená na jednotnú zmenu intervalu platnosti pre viaceré záznamy v tabuľke. Nie je z pôvodnej práce zjavné prečo autor použil práve takýto prístup, po stránke výkonu však značne zaostáva a je aj pomerne ťažké predstaviť si situácie kde by bol potrebný (napr. firma naraz prepustí sadu zamestnancov ktorú naraz najala, čo znie veľmi nepravdepodobne).

Namerané hodnoty ukazujú, že na stránke manipulácie s dátami s časom platnosti došlo k výraznému zlepšeniu nielen v technických a výrazových možnostiach, ale aj vo výkone pri operáciách so stredne veľkým množstvom záznamov.

Príkazy pre dotazovanie dát

V prípade dotazov nad dátami s časmi platnosti som zvolila podobný prístup ako v prípade tabuliek s transakčným časom, s malými odchýlkami. Ako prvý som skúšala jednoduchý agregačný dotaz nad dátami platnými v konkrétny časový okamih. Následne som vyskúšala variantu toho istého dotazu ktorá do podmienky pridáva obmedzujúci výraz nesúvisiaci s časom platnosti. Poslednou variantou ktorú som skúšala bolo spustenie predošlej varianty po vytvorení indexu pre obmedzovaný stĺpec.

	Pôvodné rozšírenie	Nové rozšírenie
Dotaz na historické dáta	60 ms	61 ms
Dotaz na historické dáta s obmedzením	55 ms	22 ms
Dotaz na historické dáta s obmedzením s existujúcim indexom	54 ms	16 ms

Tabuľka 6.6: Dĺžky trvania jednotlivých príkladov na výber dát

Namerané hodnoty korešpondujú absolútne s očakávaniami vyplývajúcimi z vopred uskutočnenej analýzy výkonnosti existujúceho riešenia. Podľa očakávaní je filtrovanie výlučne podľa časových dát prakticky totožne výkonné v porovnaní s novým rozšírením. Avšak pri pridaní nových obmedzení pôvodné riešenie trpí nemožnosťou zbaviť sa „function scan“ postupu ako prvého, počas ktorého nie je možné použiť vhodnejšie podmienky pridané v klauzule *WHERE*. Použitie indexu ešte viac stupňuje tieto rozdiely, nakoľko nový postup umožňuje začať „index scan“-om a pracovať tak s výrazne obmedzenou dátovou sadou.

6.3.3 Zhrnutie analýzy výkonnosti

Vo všetkých mnou testovaných prípadoch sa mi podarilo výkonnosť oproti pôvodnému rozšíreniu zlepšiť alebo minimálne zachovať. Prípady v ktorých bolo zlepšenie najzásadnejšie boli hlavne činnosti výpočetne náročné, ako zmeny v temporálnych tabuľkách. Z týchto dôvodov považujem zmeny ktoré som v novom temporálnom rozšírení urobila za výrazne prínosné po stránke výkonnosti. V zmenách v prospech výkonnosti rozšírenia by bolo možné aj naďalej pokračovať, avšak často pravdepodobne za cenu všeobecnosti. Jednou z možností by mohlo byť aj proaktívne pridávanie indexov na nami vytvorené tabuľky alebo stĺpce, ale

tieto operácie by nemuseli byť v reálnom nasadení žiadúce a preto považujem riešenie bez nich sa univerzálnejšie. V prípade potreby ich užívatelia môžu sami pridať.

Kapitola 7

Záver

Zadaním mojej diplomovej práce bola analýza a rozšírenie existujúceho temporálneho rozšírenia pre platformu PosgreSQL, ktoré bolo vyvinuté Radkom Jelínkom. Ako som spomenula v úvodných kapitolách práce, temporálne databázy sú dynamicky sa vyvíjajúca oblasť a existujúce databázové systémy stále vo svojich distribúciách temporálnu podporu neobsahujú alebo ju obsahujú v obmedzenej miere.

Analýza existujúceho rozšírenia odhalila viaceré problémy v oblasti funkčnosti, ako aj v oblasti výkonnosti. Jedným z prvých problémov bolo prílišné používanie funckcií vracajúcich sady záznamov. Tie boli problematické tak po stránke použiteľnosti, ako aj po stránke výkonnosti. Čo sa použiteľnosti týka, volanie funkcií v podobe v akej boli naimplementované núti užívateľa opustiť syntax jazyka SQL a podávať množstvo informácií pomocou reťazcov. Toto obchádza typovú kontrolu a je len ťažko rozšíriteľné. Výkonnostné problémy spôsobené prílišným používaním funkcií sú spojené s funkcionalitou plánovača. Ten vynucuje kompletné vyhodnotenie funkcie pred aplikáciou vonkajších podmienok, čím výrazne znižuje efektivitu dotazovania.

Ďalšou z oblastí, kde som identifikovala problémy bol návrh samotnej relačnej schémy reflektujúcej temporálnosť dát. Tento nereflektoval rozdiely v prístupe k dátam s transakčným časom a k dátam s časom platnosti. Toto samostatné ukladanie časov rovnakým spôsobom spôsobovalo výrazné výkonnostné problémy, najmä pri zmenách dát. Takto zvolený návrh spôsoboval aj problémy s funkcionalitou. Vyriešiť sa mi ich podarilo použitím samostatného prístupu k jednotlivým druhom temporálnych dát.

Okrem výkonnostných nedostatkov som sa vo svojej práci zaoberala aj nedostakami vo funkčnosti pôvodného rozšírenia. Aby som nedostatky tohto druhu našla, vypracovala som sadu testovacích príkazov, ktoré som sa snažila vykonať s pomocou existujúceho rozšírenia. Časť týchto príkladov vychádzala s existujúcich dokumentov popisujúcich TSQL2, časť som pridala na základe mojich predstáv. Výsledkom tohto testovania bola sada funkčných nedostatkov ako aj nedostatkov v oblasti návrhu užívateľského rozhrania. Po spracovaní mojich zmien som k práci priložila testovací skript demonštrujúci použitie rozšírenia na vzorovej sade dát. Toto môže byť použité ako prvotný krok pri ďalšej práci v tejto oblasti. Nezávislým prínosom bolo tiež pridanie sady agregačných funkcií, ktoré prácu s temporálnym rozšírením uľahčujú.

Poslednou významnou časťou práce bolo zrovnanie výkonnosti jednotlivých rozšírení na sade testovacích SQL skriptov. V tejto sa mi podarilo demonštrovať situácie, v ktorých moje zmeny výkonnosť výrazne zlepšili. Bohužiaľ nebolo možné otestovať viaceré zaujímavé scenáre, keďže som pri ich testovaní narazila na funkčné nedostatky pôvodného rozšírenia.

Po stránke ďalších rozšírení mojej práce by bolo pravdepodobne najvhodnejšou cestou

zapojenie riešenia do nejakej reálnej aplikácie, ktorá by temporálnu podporu vyžadovala. Dôvodom je, že ďalšie zlepšenia v oblasti výkonu budú pravdepodobne obsahovať zapojenie indexov alebo iných pomocných štruktúr. Tieto však prinášajú kompromisy tak v množstve potrebného miesta ako aj vo výkonnosti pri iných operáciách. V rámci môjho rozšírenia som sa s týmto vysporiadala transparentnejším prístupom v návrhu tabuliek, kde používam jednoducho pomenované a užívateľovi dostupné názvy pomocných atribútov a tabuliek. Vďaka tomu má užívateľ v prípade nutnosti možnosť pridať index v miestach kde to bude nutné.

Vo všeobecnosti sú výsledky mojej práce v súlade so zadaním. V oblasti výkonu som výsledky predošlej práce výrazne zlepšila, čo som demonštrovala pomocou konkrétnych meraní a popísala po stránke návrhu. Z pohľadu rozšírenia funkcionality som pridala a zmenila možnosti práce s temporálnym rozšírením tak aby lepšie zodpovedali realistickým aplikáciám ako aj jazyku TSQL2.

Literatúra

[1] Arkhipov, V.: *Temporal Tables Extension*. PostgreSQL Extension Network, Srpen 2015, [Online; navštíveno 16.5.2016].
URL http://pgxn.org/dist/temporal_tables/

[2] Chamberlin, D. D.; Astrahan, M. M.; Blasgen, M. W.; aj.: A History and Evaluation of System R. *Commun. ACM*, ročník 24, č. 10, Říjen 1981: s. 632–646, ISSN 0001-0782, doi:10.1145/358769.358784.
URL http://doi.acm.org/10.1145/358769.358784

[3] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, ročník 13, č. 6, Červen 1970: s. 377–387, ISSN 0001-0782, doi:10.1145/362384.362685.
URL http://doi.acm.org/10.1145/362384.362685

[4] Darwen, H.; Date, C.: An Overview and Analysis of Proposals Based on the TSQL 2 Approach. 2005.

[5] Deckelmann, S.; Davis, J.: *Temporal PostgreSQL*. pgfoundry, 2008, [Online; navštíveno 16.5.2016].
URL http://pgfoundry.org/projects/temporal/

[6] Group, T. P. G. D.: *PostgreSQL 9.5.2 Documentation*. 2016, [Online; navštíveno 16.5.2016].
URL http://www.postgresql.org/files/documentation/pdf/9.5/postgresql-9.5-A4.pdf/

[7] Group, T. P. G. D.: *PostgreSQL History*. 2016, [Online; navštíveno 16.5.2016].
URL http://www.postgresql.org/about/history/

[8] International Organization for Standardization: *ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL*. 1992, 587 s., available in English only.
URL http://www.iso.ch/cate/d16663.html

[9] International Organization for Standardization: *ISO/IEC 9075-1:1999: Information technology — Database languages — SQL (SQL/Framework)*. 1999.
URL http://www.iso.ch/cate/d26196.html

[10] ISO: *ISO/IEC 9075-1:2011 Information technology — Database languages — SQL (SQL/Framework)*. Prosinec 2011.
URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681

[11] Jelínek, R.: *Temporální rozšíření pro PostgreSQL, diplomová práce*. Diplomová práce, FIT VUT v Brně, Brno„ 2015.

[12] Kulkarni, K.; Michels, J.-E.: Temporal Features in SQL:2011. *SIGMOD Rec.*, ročník 41, č. 3, Říjen 2012: s. 34–43, ISSN 0163-5808, doi:10.1145/2380776.2380786.
URL http://doi.acm.org/10.1145/2380776.2380786

[13] Nicola, M.; Sommerlandt, M.: *Managing time in DB2 with temporal consistency*. IBM developerWorks, Červenec 2012, [Online; navštíveno 16.5.2016].
URL http://www.ibm.com/developerworks/data/ibrary/techarticle/dm-1207db2temporalintegrity/index.html

[14] Snodgrass, R. T.: *The TSQL2 Temporal Query Language*. Norwell, MA, USA: Kluwer Academic Publishers, 1995, ISBN 0792396146.

[15] Snodgrass, R. T.; Ahn, I.; Ariav, G.; aj.: A TSQL2 Tutorial. Technická zpráva.
URL http://www.cs.aau.dk/~{}csj/Thesis/pdf/chapter9.pdf

[16] Stonebraker, M.; Rowe, L. A.: The Design of POSTGRES. *SIGMOD Rec.*, ročník 15, č. 2, Červen 1986: s. 340–355, ISSN 0163-5808, doi:10.1145/16856.16888.
URL http://doi.acm.org/10.1145/16856.16888

[17] Taylor, R. W.; Frank, R. L.: CODASYL Data-Base Management Systems. *ACM Comput. Surv.*, ročník 8, č. 1, Březen 1976: s. 67–103, ISSN 0360-0300, doi:10.1145/356662.356666.
URL http://doi.acm.org/10.1145/356662.356666

Prílohy

Zoznam príloh

Príloha A

Obsah CD

Súčasťou práce je priložené DVD, obsahujúce všetky dokumenty týkajúce sa práce. Celý text práce je možné nájsť v hlavnom adresári na DVD vo formáte pdf. Adresárová štruktúra priloženého DVD je nasledujúca:

- **thesis.pdf** - text diplomovej práce
- **temporal-setup.sql** - inštalačný súbor rozšírenia
- **examples.sql** - súbor so sadou vzorových príkladov
- **doc.html** - dokumentácia rozšírenia