Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Historické verzie v databázach časových radov

Diplomová práca

2019
Bc. Filip Janitor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Historické verzie v databázach časových radov

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	doc. RNDr. Robert Lukoťka PhD.

Bratislava, 2019
Bc. Filip Janitor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Filip Janitor
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Historické verzie v databázach časových radov
Storing historical data in time series databases

Anotácia: Časový rad je čiastočná funkcia, ktorá času priradí hodnotu. Na ukladanie časových radov je možné použiť ľubovoľnú databázu a nad ňou vytvoriť vlastný databázový model. Alternatívou sú špecializované databázy časových radov, ktoré poskytujú hotové riešenie. Práca s časovými radmi je charakteristická veľkým množstvom dát. Na zefektívnenie fungovania úložísk časových radov kladú mnohé systémy dodatočné podmienky na uchovávané časové rady. Jednou z takýchto podmienok je predpoklad nemennosti dát, t.j. že ak pre daný časový rad a pre daný čas je do databázy zapísaná hodnota, túto hodnotu už nemožno meniť. V prípade, že systém nemá predpoklad nemennosti dát, je žiadúce, aby bol schopný rekonštruovať stav časového radu pred zmenou, resp. v akomkoľvek čase v minulosti. O takomto systéme hovoríme, že podporuje historické verzie. Prieskum v bakalárskej práci, na ktorú táto diplomová práca nadväzuje ukázal, že nie je dostatok dostupných a aktívne podporovaných open-source systémov na správu časových radov, ktoré podporujú historické verzie a poskytujú vhodné rozhranie pre vykonávanie agregácií. Cieľom diplomovej práce je preskúmať niektoré existujúce systémy a do vybraného systému zakomponovať podporu historických verzií.

Vedúci: doc. RNDr. Robert Lukoťka, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 12.12.2017

Dátum schválenia: 13.12.2017 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Moje poďakovanie patrí rodine za prejavenú trpezlivosť a podporu a tiež môjmu školiteľovi doc. RNDr. Robertovi Lukoťkovi, PhD. za venovaný čas, odborné rady a ústretovosť.

Abstrakt

Táto diplomová práca je venovaná skúmaniu vlastností systémov na správu časových radov s ohľadom na ich rozšíriteľnosť o podporu uchovávania historických verzii. Popisujeme aj našu experimentálnu implementáciu tejto funkcionality do jedného zo systémov, pričom rozoberáme dôsledky takejto zmeny na garancie, poskytované dátovým modelom tohoto systému.

Kľúčové slová: časové rady, databáza časových radov

Abstract

This diploma thesis explores the properties of time series management systems with regard to their extensibility for support of preserving historical versions of data. We also describe our experimental implementation of this functionality in one of the studied systems, discussing the implications of such change on the guarantees provided by the data model of this system.

Keywords: time series, time series databases

Obsah

Zoznam obrázkov

Úvod

V súčasnej dobe prebiehajú systematické pozorovania a zhromažďovanie údajov o určitých javoch v čase v mnohých oblastiach ľudského pôsobenia. Tieto pozorovania môžu pochádzať zo senzorov používaných vo vede a výskume, monitorovacích systémov sledujúcich javy vo finančnom alebo reálnom svete okolo nás, alebo môžu vznikať napríklad aj v domácich spotrebičoch, ktorých úlohou je zjednodušovať a zefektívňovať ľudský život na základe prepojení v rámci internetu vecí. Výsledkom týchto pozorovaní sú dáta časových radov, ktoré je nutné uchovávať, organizovať a sprístupňovať, pretože nám umožňujú nahliadnuť do minulosti, nachádzať súvislosti medzi javmi, analyzovať trendy a vytvárať predpovede. Kvôli potenciálne veľkému množstvu spracúvaných dát a relatívne malému významu jednotlivých záznamov z dát je dôležité, aby systémy na správu časových radov poskytovali sadu silných agregácií a štatistických nástrojov, umožňujúcich prácu v takomto režime.

Najmä pri použití systémov v rámci vedeckého výskumu je dôležité, aby tiež bolo možné vykonávať úpravy a korekcie zhromaždených dát a zároveň z dôvodu potreby auditov či opakovaných výpočtov rekonštruovať predchádzajúce verzie upravených dát.

V tejto práci sa najskôr venujeme rozboru vplyvu pridania podpory uchovávania historických verzií do existujúcich systémov na správu časových radov. Ďalej detailne popisujeme fungovanie vybraného systému a rozoberáme úpravy, ktoré sme v ňom vykonali. V závere skúmame vplyv našich zmien na určité garancie poskytované uvedeným systémom.

Kapitola 1 obsahuje prehľad pojmov, ktoré v ďalších častiach práce používame. V kapitole 2 podrobnejšie rozoberáme praktické vlastnosti uchovávania historických verzií časových radov a skúmame rozšíriteľnosť funkcionality vybraných existujúcich systémov. Kapitola 3 obsahuje detailný a aktuálny popis funkcionality systému BTrDB, na čo nadväzuje obsah kapitoly 4, kde analyzujeme naše úpravy tohoto systému. V kapitole 5 rozoberáme dopad našich zmien na fungovanie systému a skúmame možné prístupy na vyriešenie vzniknutých obmedzení.

Kapitola 1

Základné pojmy

V tejto kapitole sú uvedené vysvetlenia základných pojmov, ktoré sú použité v práci, vzhľadom k skutočnosti, že rôzne publikácie používajú niektoré z týchto pojmov v iných významoch a zahŕňajú do nich rôzne rozširujúce predpoklady. Naväzujeme na pojmy zavedené v bakalárskej práci [14].

Na prácu s časovými radmi používame reprezentáciu časových údajov v počítači, ktorú nazývame *časová pečiatka*. Pre daný systém označujeme konečnú množinu časových pečiatok, ktorú používa, symbolom $\mathcal{T}$. Vo všeobecnosti platí, že nie každý čas je možné reprezentovať časovou pečiatkou. Pre daný systém teda existuje čiastočné zobrazenie $h\colon \mathcal{C} \to \mathcal{T}$, kde $\mathcal{C}$ je množina všetkých časov. Nad množinami $\mathcal{C}$ aj $\mathcal{T}$ existuje úplné usporiadanie. Platí pritom, že ak $a, b \in D(h)$ a zároveň $a < b$, tak aj $h(a) < h(b)$. Definujeme tiež pre $a, b \in D(h)$ nasledujúce symboly:

$$
\begin{aligned}
(-\infty, h(b))_h &= \{h(x) \mid x \in D(h), x < b\} \\
\langle h(a), h(b))_h &= \{h(x) \mid x \in D(h), x \in \langle a, b)\} \\
\langle h(a), h(b)\rangle_h &= \{h(x) \mid x \in D(h), x \in \langle a, b\rangle\} \\
(h(a), h(b))_h &= \{h(x) \mid x \in D(h), x \in (a, b)\} \\
\langle h(a), \infty)_h &= \{h(x) \mid x \in D(h), x \geq a)\}
\end{aligned}
$$

Tieto symboly nazývame *intervaly časových pečiatok*. Pre intervaly $\langle h(a), h(b))_h$, $\langle h(a), h(b)\rangle_h$ a $(h(a), h(b))_h$ nazývame $h(a)$ *začiatočnou časovou pečiatkou intervalu* a $h(b)$ *koncovou časovou pečiatkou intervalu*. Počet prvkov z intervalu $(h(a), h(b))_h$ nazývame *rozdielom časových pečiatok* $h(a)$ *a* $h(b)$. Ak rozdiel časových pečiatok $h(a)$ a $h(b)$ je rovný nule, hovoríme, že časové pečiatky $h(a)$ a $h(b)$ sú *susednými časovými pečiatkami*. Pre zjednodušenie zápisu budeme tam, kde to nie je nevyhnutné vynechávať označenie dolným indexom h.

V praxi platí, že vzory časových pečiatok vzhľadom na zobrazenie h sú na časovej osi rozmiestnené v určitom zmysle rovnomerne. To znamená, že pre ľubovoľné $t_a, t_b, t_c, t_d \in D(h)$ také, že $h(t_a)$ a $h(t_b)$ sú susednými časovými pečiatkami a tiež $h(t_c)$

a $h(t_d)$ sú susednými časovými pečiatkami, platí, že rozdiel časov t_a a t_b je rovnaký ako rozdiel časov t_c a t_d. V ďalších častiach predpokladáme, že používané množiny časových pečiatok majú takúto praktickú vlastnosť.

Reprezentáciu hodnôt v počítači, ktoré používame pri práci s časovými radmi, nazývame *údajmi*. Množinu prípustných údajov označujeme $\mathcal{U}$. Systémy na správu časových radov obyčajne množinu prípustných údajov obmedzujú na primitívne dátové typy, ako sú celé či reálne čísla alebo pravdivostné hodnoty, pričom niektoré systémy umožňujú uchovávať ako údaj aj reťazec znakov. V niektorých prípadoch je však vhodné, aby kvôli vzájomnej logickej súvislosti údaj obsahoval informácie o viacerých veličinách. Údaje časového radu zaznamenávajúceho informácie o výnimočných udalostiach v počítačovom systéme môžu napríklad obsahovať identifikátor používateľa, ktorý danú udalosť vyvolal, jej závažnosť, popis a podobné informácie. Údaj je v takomto prípade mapou a jednotlivé dvojice ⟨kľúč,hodnota⟩ tejto mapy nazývame *polia*. Ak systém umožňuje, aby údaj bol mapou, hovoríme, že takýto systém na správu časových radov *podporuje polia*.

Definícia. *Časovým radom* nazývame čiastočnú funkciu $f : \mathcal{T} \to \mathcal{U}$.

Definícia. *Záznam* časového radu f je dvojica $(c, f(c))$.

Zavádzame tiež pojem označujúci časové rady, ktorých definičný obor má špeciálnu vlastnosť.

Definícia. Časový rad f nazývame *periodickým* ak platí, že pre každé dve časové pečiatky $c, c' \in D(f)$ také, že $c < c'$, buď platí $(c, c') \cap D(f) \neq \emptyset$, alebo je rozdiel časových pečiatok c a c' rovný p.

Číslo p z predchádzajúcej definície nazývame *periódou časového radu*.

Ako sme uviedli, systémy môžu používať rôzne množiny $\mathcal{T}$, preto zavádzame pojem, umožňujúci v istom zmysle navzájom porovnať rôzne systémy používajúce rôzne druhy časových pečiatok.

Definícia. *Rozlíšenie záznamov* v systéme je rozdiel časov t_a a t_b pre ktoré platí, že $h(t_a)$ a $h(t_b)$ sú susedné časové pečiatky.

V rámci systému je potrebné vedieť jednotlivé časové rady dobre identifikovať a kategorizovať. Okrem názvu môže byť preto vhodné priradiť každému časovému radu množinu dvojíc ⟨kľúč,hodnota⟩. Jednotlivé dvojice z tejto množiny v takomto prípade nazývame *tagy*. Výhody použitia tagov ilustruje nasledujúci príklad.

Systém uchováva časové rady obsahujúce údaje o stave procesorov v clusterovej federácii. Konkrétny časový rad, zachytávajúci nejakú veličinu musí byť priraditeľný ku konkrétnemu procesoru, a teda je potrebné uchovať informáciu o polohe daného

procesoru v rámci federácie clusterov, identifikáciu serveru v clusteri a informáciu určujúcu o ktorý konkrétny procesor v rámci serveru ide. Taktiež musí byť zrejmé, akú veličinu časový rad zaznamenáva. Môže to byť teplota, záťaž procesora a pod. Ak na identifikáciu časového radu, obsahujúceho údaje o teplote procesoru 0 v serveri 6 v clusteri 1 použijeme iba názov, ako vhodné sa javí použitie niektorej z formátovacích konvencií, v ktorých sú jednotlivé identifikátory oddelené vhodným symbolom. Napríklad `cluster1.node6.cpu0.teplota`. Hoci je tento tvar pre používateľa zrozumiteľný, nie je vhodný v prípade, že je potrebné vytvárať dopyty na rôzne množiny časových radov, nachádzajúcich sa v systéme. Napríklad dopyt na množinu všetkých časových radov obsahujúcich údaje o teplote zo serverov 1 a 5 v clusteri číslo 4 nie je praktické vytvárať len pomocou reťazcových operácií. Naproti tomu, v systéme podporujúcom tagy môžeme pomenovať uvedený časový rad ako `teplota`, a priradiť mu množinu tagov $\{\langle \text{cluster,1} \rangle, \langle \text{node,6} \rangle, \langle \text{cpu,0} \rangle\}$. Uvedený dopyt je potom možné jednoducho vyjadriť v pseudodopytovacom jazyku podobnom SQL ako

```
SELECT teplota WHERE (node=1 OR node=5) AND cluster=4
```

V prípade, že systém umožňuje priradiť časovému radu množinu tagov hovoríme, že systém *podporuje tagy*. Vzhľadom na identifikačný účel tagov v takomto systéme na správu časových radov ďalej predpokladáme, že množina tagov, priradená konkrétnemu časovému radu, je mu priradená v momente jeho vytvorenia a ďalej sa nemení. Predpokladáme, že množina tagov spolu s názvom vytvárajú unikátny identifikátor časového radu v rámci systému.

Niektoré systémy umožňujú k jednotlivým časovým radom priradiť množinu dvojíc ⟨kľúč,hodnota⟩, ktorá sa v priebehu času môže meniť. Konkrétny časový rad vtedy býva identifikovateľný a vyhľadateľný aj bez použitia dvojíc z tejto množiny. V takom prípade dvojice ⟨kľúč,hodnota⟩ z tejto množiny nazývame *anotácie* a o systéme, ktorý ich použitie podporuje hovoríme, že systém *podporuje anotácie*.

Definícia. *Aktualizácia časového radu* je akákoľvek operácia, ktorá zmení hodnotu údaju resp. údajov časového radu, alebo pridá nový záznam, t.j. rozšíri definičný obor časového radu.

Špeciálnym druhom operácie, ktorá upravuje časový rad je odstraňovanie záznamov.

Definícia. *Odstránenie z časového radu* je operácia, ktorá zúži definičný obor časového radu tým, že z neho odstráni množinu časových pečiatok.

Pri práci s časovými radmi v systéme často používame agregácie, aby sme z uložených údajov získali pre nás podstatné informácie. Poznáme hodnotové a radové agregácie.

Definícia. *Hodnotová agregácia* je funkcia $g\colon (c_1, c_2, \ldots c_n) \to u$, kde $(c_1, c_2, \ldots c_n)$ je n-tica časových radov a u je údaj.

Príkladom hodnotovej agregácie môže byť napríklad zistenie priemernej teploty z nameraných hodnôt za posledný týždeň. Vzhľadom k zavedenému označeniu je vhodné pod hodnotové agregácie zahrnúť aj operácie, ktorých výsledok je svojou štruktúrou podobný údaju v časovom rade systému podporujúceho polia. Príkladom takejto agregácie je napríklad získanie histogramu početností.

Definícia. *Radová agregácia* je funkcia $g\colon (c_1, c_2, \ldots c_n) \to r$, kde $(c_1, c_2, \ldots c_n)$ je n-tica časových radov a r je časový rad.

Príkladom radovej agregácie je napríklad derivácia, časový posun alebo vývoj priemernej spotreby energie v mestskej časti, kde každý údaj výsledného radu je priemerom údajov v tom čase zo všetkých meračov v danej mestskej časti.

Agregácie delíme aj podľa počtu členov vstupnej n-tice časových radov.

Definícia. *Agregácia cez jeden časový rad* je agregácia, kde $n = 1$.

Definícia. *Agregácia cez viacero časových radov* je agregácia, kde $n > 1$.

V praxi sa používa ešte jeden špeciálny prípad radovej agregácie cez jeden časový rad. Takúto agregáciu nazývame *agregácia po skupinách (bucket alebo group aggregation)*. V prípade tejto operácie rozložíme množinu $\mathcal{T}$ na intervaly časových pečiatok. Každému z intervalov rozkladu priradíme časovú pečiatku (reprezentanta) tak, že reprezentant $y_i \in \langle x_i, x_{i+1})$. Reprezentantom jednotlivých intervalov priradíme údaje, ktoré vzniknú agregovaním zúženia vstupného časového radu na nimi reprezentované intervaly rozkladu hodnotovou agregáciou. Časový rad, ktorý týmto získame, je výsledkom agregácie po skupinách. Agregácia po skupinách sa používa napríklad pri normalizácii a resamplingu časových radov.

1.1 Rozšírenie definície

Dáta časových radov sú uchovávané pre rôzne použitia a v rôznych prostrediach. Mnohé z týchto prostredí sa vyznačujú potrebou uchovávať korekcie a aj pôvodné verzie údajov s ktorými pracujú, prípadne potrebou uchovávať informáciu potrebnú na rekonštrukciu stavu časového radu k určitému času, aby bolo možné vykonávať audity či opakovať výpočty vykonávané na uchovaných dátach. Niektoré všeobecné databázové systémy [23, 15] reagujú na požiadavky týchto prostredí a umožňujú natívnu podporu uchovávania historických verzií uložených dát.

Systém na správu časových radov nemusí mať predpoklad nemenných existujúcich záznamov a teda môže umožňovať modifikovať údaj v uložených záznamoch časového radu. Požiadavka uchovávania historických verzií znamená, že takáto úprava nenahradí pôvodnú hodnotu, ale obe budú naďalej uchovávané, pričom o pôvodnej hodnote bude systém vedieť, že bola modifikovaná.

V prípade systémov na správu časových radov pridáva požiadavka na uchovávanie historických verzií do dát tretiu dimenziu. Dimenziami sú v takýchto systémoch identifikátory časových radov, čas a verzie. Je teda nutné do záznamu časového radu pridať ďalšiu položku - identifikátor verzie. Pre daný systém označme množinu podporovaných identifikátorov verzií symbolom $\mathcal{I}$. Nad touto množinou je definované úplné usporiadanie. Pre $i, j \in \mathcal{I}$ nasledujúci symbol definujeme ako interval *identifikátorov verzií*:

$$\langle i, j)_{\mathcal{I}} = \{x \in \mathcal{I} \mid i \leq x < j\}$$

Ostatné druhy intervalov, podobne ako je uvedené pri intervaloch časových pečiatok, sú pre identifikátory verzií definované podobne prirodzeným spôsobom.

Vlastnosť uchovávania historických verzií a nové pojmy s ňou súvisiace zavádzame v definíciách nachádzajúcich sa v nasledujúcich častiach. Pre zjednodušenie teraz uvažujeme iba o práci s jedným časovým radom a nie o systéme, v ktorom je potrebné identifikovať viacero časových radov.

Systémy, ktoré podporujú uchovávanie historických verzií časových radov môžu poskytovať rôzne rozhrania na prácu s nimi. Táto vlastnosť spôsobuje, že vznikajú dva konceptuálne odlišné spôsoby, ako systém funguje a aké operácie v ňom majú význam. Zadefinujeme teda dva pojmy, ktoré odzrkadľujú rozdielnosť týchto dvoch pohľadov a popíšeme rozdiely medzi systémami, fungujúcimi jednotlivým spôsobom.

Definícia. *Časovým radom s jednoduchými historickými verziami* nazývame čiastočnú funkciu $f : \mathcal{I} \times \mathcal{T} \to \mathcal{U}$, kde $\mathcal{T}$ je množina časových pečiatok, $\mathcal{U}$ množina údajov a $\mathcal{I}$ množina identifikátorov verzií.

Definícia. *Časovým radom so špeciálnymi hodnotami* nazývame čiastočnú funkciu $f : \mathcal{I} \times \mathcal{T} \to \mathcal{U} \cup \{Nil\}$, kde $\mathcal{T}$ je množina časových pečiatok, $\mathcal{U}$ množina údajov, $\mathcal{I}$ množina identifikátorov verzií a Nil je špeciálna hodnota, pre ktorú platí, že $Nil \notin \mathcal{U}$.

Tam, kde majú zavádzané pojmy zmysel pre oba druhy pohľadov používame na označenie oboch predchádzajúcich pojmov spoločný pojem *časový rad s historickými verziami*.

Definícia. *Záznam časového radu s historickými verziami* f je trojica $(i, c, f(i, c))$, kde i je identifikátor verzie a c časová pečiatka.

Špeciálnym druhom úpravy v prípade časového radu s historickými verziami je vymazanie údaju bez náhrady novou hodnotou. Rozdiel medzi operáciou odstránenie pri časovom rade a vymazaním pri časových radoch s historickými verziami je ten, že v druhom prípade, hoci nastáva z logického hľadiska vymazanie, definičný obor časového radu s historickými verziami sa rozšíri, nie zúži. V praktickom použití je

jediná používaná úprava časového radu rozšírenie definičného oboru a to, aké záznamy sú pridané určuje logický význam tejto operácie.

Systém fungujúci prvým spôsobom používa časové rady s jednoduchými historickými verziami. Takýto systém pri každej aktualizácii dopĺňa do novovytvorenej verzie aj záznamy pre časové pečiatky s údajmi z predchádzajúcej verzie, ktoré sa aktualizáciou nemenili a v rozhraní na čítanie podporuje iba dopyt na konkrétne verzie spolu s poslednou. Ak je v určitej verzii záznam pre nejakú časovú pečiatku nedefinovaný, hoci bol v danej časovej pečiatke definovaný v predchádzajúcej verzii, znamená to, že v aktualizácii, ktorá vytvorila túto verziu bol jeho údaj odstránený. V prípade, že klient vymaže všetky existujúce záznamy si systém musí poznačiť, že jednu verziu musí preskočiť, keďže aktualizáciou sa v skutočnosti nič nemôže pridať.

Hovoríme, že časový rad f' je *verziou j časového radu s historickými verziami f* ak pre každú časovú pečiatku c je $f'(c) = f(j, c)$ alebo je $f'(c)$ nedefinované, v prípade že $f(j, c)$ nie je definované. Dopyty na nejakú verziu j časového radu s historickými verziami f nazývame aj dopytmi na *konkrétnu verziu časového radu s historickými verziami f*.

Posledný identifikátor verzie časového radu s jednoduchými historickými verziami je identifikátor i taký, že buď i je maximálny identifikátor verzie, pre ktorý platí, že existuje časová pečiatka c, pre ktorú je $f(i, c)$ definované, alebo bola naposledy vykonaná operácia, ktorá vymazala údaje všetkých záznamov, ktorých identifikátor verzie bol posledným identifikátorom verzie pred touto operáciou. Vtedy je posledný identifikátor i taký, že ak j je maximálny identifikátor verzie, pre ktorý platí, že existuje časová pečiatka c pre ktorú je $f(j, c)$ tak i je najmenší z identifikátorov, pre ktorý platí $j < i$.

Posledná verzia časového radu s jednoduchými historickými verziami f je časový rad f', ktorý je verziou j časového radu s jednoduchými historickými verziami f, pričom j je posledný identifikátor verzie pre tento rad.

Uvedená definícia časového radu s jednoduchými historickými verziami môže byť pre niektoré praktické využitia priveľmi voľná, najmä ak uchovávanie verzií neslúži na umožnenie modifikácie existujúcich záznamov ale len na spomínanú rekonštrukciu stavov časového radu v systéme. Uvádzame preto aj jej reštriktívne formy.

Definícia. *Časovým radom s verziami* nazývame taký časový rad s jednoduchými historickými verziami f, pre ktorý platí, že ak f je definované na dvojici (i, c), tak potom pre všetky j také, že $j > i$ platí, že $f(j, c)$ je definované a $f(j, c) = f(i, c)$

Definícia. *Časovým radom so súvislými verziami* nazývame taký časový rad s jednoduchými historickými verziami f, pre ktorý platí, že ak f je definované na dvojici (i, c) a v je identifikátor verzie taký, že $i < v$, $f(v, c)$ je nedefinované a pre všetky identifikátory j z intervalu identifikátorov $\langle i, v)$ je $f(j, c)$ definované, tak potom $f(j, c) = f(i, c)$.

Systém pracujúci s časovými radmi s verziami neumožňuje používateľovi mazať ani modifikovať uložené záznamy, časové rady so súvislými verziami zas zabraňujú modifikácii bez predchádzajúceho vymazania.

Systém pracujúci druhým spôsobom používa časové rady so špeciálnymi hodnotami. Takýto systém využíva usporiadanie identifikátorov verzií, aby vedel odlíšiť, ktoré hodnoty údajov sú aktuálne platné a pri aktualizáciách nekopíruje nezmenené záznamy do novej verzie. Namiesto toho rekonštruuje stavy v systéme pomocou záznamov v rôznych verziách. Ako rozhranie poskytuje obyčajne dopyty na aktuálnu verziu a na časový rad do verzie, pričom význam týchto pojmov definujeme v nasledujúcich odsekoch. Ak takýto systém umožňuje operáciu vymazávania záznamov, je nutné explicitne vyznačiť, že sa v určitej verzii údaj pre konkrétnu časovú pečiatku stal nedefinovaný, inak sa vymazanie v odpovedi na dopyt čítajúci časový rad do konkrétnej verzie neprejaví. Na toto vyznačenie sa používa práve špeciálna hodnota Nil, ktorá ma osobitnú sémantiku.

Definícia. *Aktuálna verzia záznamu časového radu so špeciálnymi hodnotami f s časovou pečiatkou c* je taký záznam $(i, c, f(i, c))$, že pre všetky ostatné záznamy $(i', c, f(i', c))$ platí, že $i' < i$. Hodnota aktuálnej verzie záznamu je $f(i, c)$

Definícia. Nech f je časový rad so špeciálnymi hodnotami, M je množina časových pečiatok taká, že $c \in M$ práve vtedy keď existuje j také, že (j, c) patrí do definičného oboru f a zároveň hodnota aktuálnej verzie záznamu pre f s časovou pečiatkou c nie je Nil. Hovoríme, že časový rad f' je *aktuálna verzia časového radu s špeciálnymi hodnotami f*, ak definičný obor f' je M a pre všetky $c \in M$ platí $f'(c) = f(i, c)$ pričom $(i, c, f(i, c))$ je aktuálna verzia záznamu s časovou pečiatkou c.

Aktuálny identifikátor verzie časového radu so špeciálnymi hodnotami je maximálny identifikátor verzie i, pre ktorý platí, že existuje časová pečiatka c, pre ktorú je $f(i, c)$ definované.

Hovoríme, že časový rad f' je *časovým radom do verzie j časového radu so špeciálnymi hodnotami f* ak pre každú časovú pečiatku c je $f'(c)$ definované práve vtedy keď existuje identifikátor verzie $i \leq j$ taký, že $f(i, c)$ je definované a zároveň pre najväčší identifikátor verzie k spĺňajúci túto podmienku platí $f(k, c) \neq Nil$. Vtedy $f'(c) = f(k, c)$.

Hovoríme, že časový rad f' je *časovým radom do verzie j časového radu so špeciálnymi hodnotami f so zachovaním špeciálnej hodnoty* ak pre každú časovú pečiatku c je $f'(c)$ definované práve vtedy keď existuje identifikátor verzie $i \leq j$ taký, že $f(i, c)$ je definované. Označme k najväčší identifikátor verzie spĺňajúci túto podmienku pre časovú pečiatku c. Potom $f'(c) = f(k, c)$.

Systém pracujúci s časovými radmi so špeciálnymi verziami tiež môže podporovať dopyt na konkrétnu verziu časového radu so špeciálnymi hodnotami f. Ak systém

podporuje takúto operáciu, umožňuje klientovi sledovať, aké zmeny nastávali pri aktualizáciách, ktoré vytvárali jednotlivé verzie. V takomto prípade je výsledkom tejto operácie časový rad f' pre ktorého množinu údajov $\mathcal{U}_{f'}$ a špeciálnu hodnotu Nil radu f platí, že $Nil \in \mathcal{U}_{f'}$.

Uvádzame aj pojmy, ktoré vhodne popisujú praktické reštrikcie používania časových radov so špeciálnou hodnotou.

Definícia. *Časovým radom s obmedzenými verziami* nazývame taký časový rad s so špeciálnymi hodnotami f, pre ktorý platí, že ak f je definované na dvojici (i, c) tak pre všetky identifikátory verzií $j \neq i$ platí, že $f(j, c)$ nie je definované.

Definícia. *Časovým radom so špeciálnymi hodnotami a oddeľovačmi* nazývame taký časový rad so špeciálnymi hodnotami f, pre ktorý platí, že ak f je definované na dvojici (i, c) a $f(i, c) \neq Nil$ tak potom buď existuje identifikátor verzie j taký, že $i < j$ a $f(j, c) = Nil$, pričom platí, že pre všetky identifikátory verzií v z intervalu (i, j) je $f(v, c)$ nedefinované, alebo $(i, c, f(i, c))$ je aktuálna verzia záznamu s časovou pečiatkou c.

Pojem časový rad s obmedzenými verziami je v istom zmysle analogický pojmu časový rad s verziami v rámci systému používajúceho časové rady s jednoduchými verziami. Podobný vzťah je aj medzi časovým radom so špeciálnymi hodnotami a oddeľovačmi a časovým radom so súvislými verziami.

Z praktického hľadiska je vhodné zaviesť aj pomocné pojmy, ktoré popisujú určité špeciálne druhy aktualizácie časových radov s historickými verziami.

Nech k označuje posledný identifikátor verzie v prípade časového radu s jednoduchými verziami a identifikátor aktuálnej verzie v prípade časového radu so špeciálnymi hodnotami. *Novým identifikátorom verzie* nazývame identifikátor verzie i, ktorý je najmenším z identifikátorov, pre ktoré platí $i > k$.

Vytvorením novej verzie označujeme aktualizáciu, ktorá rozšíri definičný obor časového radu s historickými verziami o dvojice, v ktorých identifikátor verzie je novým identifikátorom verzie pre daný rad. V prípade časových radov s jednoduchými historickými verziami týmto pojmom označujeme špeciálne aj operáciu, pri ktorej sú vymazané údaje všetkých záznamov, ktorých identifikátor verzie bol posledným identifikátorom verzie pred touto operáciou. Táto operácia definičný obor časového radu nerozšíri, ale zmení identifikátor poslednej verzie.

Ak systém pracuje s časovými radmi s jednoduchými historickými verziami, *vkladaním* alebo *vkladaním nových záznamov* označujeme vytvorenie novej verzie časového radu s jednoduchými historickými verziami f, pri ktorej pre každú z dvojíc (i, c), ktorými bol rozšírený definičný obor platí, že buď posledná verzia f pred aktualizáciou nebola definovaná v c alebo $f(i, c) = f(j, c)$, kde j je identifikátor poslednej verzie pred

aktualizáciou. Zároveň, pre každú časovú pečiatku c' platí, že ak bol f definovaný na (j, c'), tak je definovaný aj na (i, c'), kde i označuje nový identifikátor verzie.

V prípade, že systém pracuje s časovými radmi so špeciálnymi hodnotami, *vkladaním* alebo *vkladaním nových záznamov* označujeme vytvorenie novej verzie s nasledujúcimi vlastnosťami. Označme f' aktuálnu verziu časového radu so špeciálnymi hodnotami f, ktorý upravujeme, pred vkladaním. Pre každú z dvojíc (i, c), ktorými je rozširovaný definičný obor f platí, že f' je v c nedefinovaný a zároveň po tejto aktualizácii platí, že $f(i, c) \neq Nil$.

Ak systém pracuje s časovými radmi s jednoduchými verziami, *úpravou existujúcich záznamov* označujeme vytvorenie novej verzie, pri ktorej každá z dvojíc, ktorými bol rozšírený definičný obor obsahuje časovú pečiatku, v ktorej je definovaná posledná verzia upravovaného časového radu pred aktualizáciou a zároveň, pre každú časovú pečiatku c' platí, že ak bol f definovaný na (j, c') kde j je identifikátor poslednej verzie pred aktualizáciou, tak je definovaný aj na (i, c), kde i označuje nový identifikátor verzie.

Ak systém pracuje s časovými radmi so špeciálnymi hodnotami *úpravou existujúcich záznamov* označujeme vytvorenie novej verzie, s nasledujúcimi vlastnosťami. Označme f' aktuálnu verziu časového radu so špeciálnymi hodnotami f, ktorý upravujeme, pred vkladaním. Pre každú z dvojíc (i, c), ktorými je rozširovaný definičný obor f platí, že f' je v c definovaný a zároveň po tejto aktualizácii platí, že $f(i, c) \neq Nil$.

Pojmom *najnovšia verzia* spoločne označujeme poslednú verziu pri prvom spôsobe fungovania a aktuálnu verziu pri druhom spôsobe fungovania.

Pri uchovávaní časových radov s historickými verziami je dôležité určiť, aký logický význam majú aktualizácie, ktoré systém podporuje. Základná aktualizácia, ktorú systém musí podporovať, je aktualizácia najnovšej verzie časového radu, ktorého sa dopyt týka. To znamená, že pre klienta je časový rad s ktorým pracuje a upravuje ho najnovšou verziou uloženého časového radu s historickými verziami. Ak systém podporuje len takéto úpravy, jednotlivé verzie, resp. verzie do konkrétnych verzií tvoria historické stavy uložených dát v systéme, ktoré je na základe nich možné rekonštruovať. Pojmy, ktoré sme zavádzali v tejto časti majú očakávaný význam práve pri takomto používaní. Systém však môže podporovať aj typ aktualizácie, ktorá logicky naväzuje na inú verziu ako najnovšiu verziu uloženého časového radu s historickými verziami. Ak systém umožňuje takúto aktualizáciu, sémantika pojmu posledná verzia môže zostať nezmenená. Takáto aktualizácia však umožňuje upravovať minulosť, teda súvislosti medzi identifikátormi verzií vytvárajú strom závislostí a nemusí platiť, že nejaká verzia naväzuje na tie, ktorých identifikátor je menší. Pri takomto fungovaní môžu najmä pojmy týkajúce sa časových radov so špeciálnymi hodnotami stratiť svoj očakávaný význam. V práci sa zaoberáme prvým prístupom, keďže reflektuje charakteristickú vlastnosť usporiadanej množiny identifikátorov verzií. Toto obmedzenie nám umožňuje vhodne

používať pojmy zavedené v tejto časti.

Systém môže umožňovať v záujme redukcie množstva používaného priestoru niektoré historické verzie zlučovať a mazať. Používateľ vtedy môže určiť významné historické verzie ako tzv. *snímky* (snapshoty) a tie verzie, ktoré v týchto zoskupeniach nie sú vymazať.

Pre systém používajúci časové rady so špeciálnymi hodnotami funguje operácia zlučovania nasledovným spôsobom. Nech k je identifikátor významnej historickej verzie časového radu f, pre ktorú systém ide vyrobiť snímku. Nech i je identifikátor verzie z poslednej snímky f, menší ako k. V prípade neexistencie predchádzajúcej snímky uvažujme, že i je špeciálna hodnota, menšia ako všetky identifikátory verzií. Nech f' je časový rad so špeciálnymi hodnotami vytvorený nasledujúcim spôsobom. Pre každú časovú pečiatku c a identifikátor verzie v taký, že $v > i$ určime, že ak je $f(v, c)$ definované tak $f'(v, c) = f(v, c)$. Snímkou f'' pre verziu k je potom časový rad do verzie k časového radu so špeciálnymi hodnotami f' so zachovaním špeciálnej hodnoty. Úprava pôvodného časového radu f potom spočíva v tom, že pre všetky časové pečiatky c a identifikátory verzií v také, že $i < v \leq k$ urobíme $f(v, c)$ nedefinované a nakoniec definujeme $f(k, t) = f''(t)$ pre všetky t z definičného oboru f''.

Ak systém pracuje s časovými radmi s jednoduchými verziami, táto operácia spôsobí, že sa upravovaný časový rad stane nedefinovaný na dvojiciach, v ktorých patrí identifikátor verzie do intervalu (i, k).

Vytváranie snímok je možné prirodzeným spôsobom rozšíriť na viacero časových radov. Snímka množiny časových radov je vtedy množinou snímok viacerých časových radov pričom platí, že takýto snímok môže obsahovať najviac jednu verziu snímky z každého časového radu.

V prípade, že systém umožňuje vytvárať snímky, hovoríme, že systém *podporuje snímky.*

Kapitola 2

Uchovávanie historických verzií

V tejto kapitole sa venujeme skúmaniu dôsledkov pridania požiadavky na uchovávanie historických verzií na systémy na správu časových radov. Pridanie podpory časových radov s historickými verziami do systému na uchovávanie časových radov vyžaduje rozhodnutie o forme identifikátorov, použitých pri identifikovaní verzie pri dopytoch.

Niektoré systémy [4, 2] označujú historické verzie celým číslom verzie pre daný časový rad. Každou aktualizáciou časového radu sa číslo aktuálnej verzie inkrementuje. Platí teda, že čím neskôr bola aktualizácia do systému zapísaná, tým vyššie je jej číslo verzie.

Číslo verzie sa používa pri dopytoch, pričom každý dopyt sa týka dát časového radu v danej verzii, časového radu do danej verzie, alebo dát aktuálnej, či poslednej verzie časového radu v čase zadania dopytu, pričom systém musí byť schopný identifikátor takejto verzie správne určiť.

Ako bolo uvedené v predchádzajúcej kapitole, účel jednotlivých dopytov môže byť rôzny v závislosti od toho, aké časové rady s historickými verziami systém uchováva a akou formou ukladá aktualizácie. Dopyty na konkrétne verzie časového radu, prípadne na poslednú verziu môžu mať v prípade systému pracujúceho s časovými radmi so špeciálnymi hodnotami uplatnenie pri zisťovaní, aké zmeny v konkrétnej aktualizácii nastali. Takýto systém tiež môže podporovať dopyt s dvoma identifikátormi verzií, ktorý vráti ako výsledok časový rad do konkrétnej verzie so špeciálnou hodnotou, pričom vo výslednom časovom rade odstráni tie záznamy, ktoré pochádzajú zo záznamov s nižším identifikátorom verzie, než je druhý uvedený v dopyte. Týmto spôsobom je možné zisťovať, aké zmeny nastali pri aktualizáciách, ktoré vytvorili verzie s identifikátormi z intervalu uvedeného v dopyte.

Takýto prístup deleguje určitú časť réžie mimo hlavného úložiska dát. Samotné číslo verzie nehovorí nič o čase, kedy v systéme prebehla aktualizácia s konkrétnym identifikátorom. Na to, aby sa dali používať dopyty s časovým údajom miesto iného identifikátora verzie, príkladom môže byť dopyt na stav časového radu v konkrétnom

čase, je potrebné aby si klient alebo systém udržiaval mapovanie medzi časovými pečiatkami a identifikátormi verzií. Uchovávanie mapovania u klienta je problematické, ak nie je klientovi zabezpečený exkluzívny prístup k aktualizovaniu konkrétneho časového radu. V takom prípade, keď jeden časový rad aktualizuje viacero nezávislých klientov vyžaduje udržiavanie mapovaní u jednotlivých klientov ďalšiu réžiu. Rozdiel medzi udržiavaním mapovania v systéme a u klienta môže byť spôsobený aj nedostatočnou synchronizáciou hodín a tiež latenciou medzi zadaním požiadavky na aktualizáciu a skutočným vykonaním tejto požiadavky systémom. Pri použití mapovania u klienta môže byť teda vhodné časové pečiatky k číslam verzií prideľovať v systéme a odosielať ich klientovi spolu s potvrdením úspešnosti aktualizácie.

Pri takomto prístupe je tiež potrebné určenie rozsahu platnosti identifikátoru verzie. Každý časový rad môže mať osobitné počítadlo verzií alebo je možné vytvárať skupiny časových radov so spoločnými počítadlami verzií, ktoré sa inkrementujú pri aktualizácii ľubovoľného z časových radov patriacich do danej skupiny. Takýto prístup uľahčuje formulovanie dopytov, ktoré počítajú s viacerými časovými radmi, pretože je možné rekonštruovať stav celej skupiny k danému času s pomocou jedného čísla verzie. Zároveň však tento prístup narúša náväznosť jednotlivých identifikátorov verzií jedného časového radu a tým narúša význam niektorých zavedených pojmov, napr. pojmu vytvorenie novej verzie.

Iným prístupom je uchovávať ako identifikátor verzie priamo časovú pečiatku vo vhodnom formáte. Táto pečiatka môže obsahovať čas spracovania aktualizácie v systéme. Vďaka tomu je rozsah takéhoto identifikátoru globálny pre všetky časové rady v systéme. Je dôležité poznamenať, že hoci tieto identifikátory sú tiež časovými pečiatkami, nemajú súvis s časovými pečiatkami v záznamoch časových radov a teda v systéme môžu vznikať na prvý pohľad neintuitívne stavy. Mohlo by sa totiž zdať, že pri vložení nových záznamov by časová pečiatka záznamu mala byť menšia alebo rovná časovej pečiatke slúžiacej ako identifikátor verzie. Toto korešponduje s predstavou, že najskôr prebehne meranie, ktoré je zdrojom údaju a až potom sa zapíše do systému. V prípade, že hodiny zdroja meraní a hodiny systému nie sú zosynchronizované, môže sa stať, že časová pečiatka verzie je nižšia ako časová pečiatka záznamu časovej verzie, teda do systému je akoby zapisovaný údaj z budúcnosti.

V prípade použitia časových pečiatok ako identifikátorov verzie je možné pokúsiť sa aj o prístup, ktorý je opačný k spomínanému a narúša význam pojmov ako aktuálna verzia záznamu či časového radu. Časová pečiatka v identifikátore verzie môže znamenať aj čas, po ktorý daný záznam platí. Toto umožňuje pri aktuálnych verziách záznamov šetriť priestor a uchovávať ich bez identifikátoru verzie, keďže ich platnosť zatiaľ nie je zhora ohraničená. Takýto prístup však vyžaduje uchovávanie akýchsi zarážok, prázdnych záznamov platných pred prvým zaznamenaným záznamom, aby bolo možné rekonštruovať historický stav v systéme aj keď sú záznamy vkladané v popre-

hadzovanom poradí, či klient zapisuje záznamy pomocou spätného dopĺňania.

2.1 Využitie existujúceho systému

Prieskum nachádzajúci sa v bakalárskej práci [14] poukázal na nedostatok dostupných a aktívne podporovaných open-source systémov na správu časových radov, ktoré podporujú historické verzie a poskytujú vhodné rozhranie pre vykonávanie agregácií.

Keďže systém Arctic vo svojom oficiálnom popise uvádza možnosť uchovávať historické verzie dát, rozhodli sme sa detailne ho preskúmať a zistiť, ktoré z časových radov s historickými verziami podporuje a akú funkcionalitu k nim poskytuje.

2.1.1 Arctic

Systém Arctic [4] je napísaný v jazyku Python a dáta ukladá do NoSQL dokumentovej databázy MongoDB [9]. Spôsob, akým Arctic ukladá dáta časových radov do MongoDB je určený použitým úložiskovým modulom. Arctic umožňuje používateľovi naprogramovať si vlastný úložiskový modul a taktiež poskytuje tri oficiálne podporované moduly, zamerané na uchovávanie dát časových radov. Jeden z nich, modul `VersionStore` implementuje uchovávanie časových radov s jednoduchými historickými verziami. Tieto úložiskové moduly využívajú možnosť vytvárania sekundárnych indexov na dátové štruktúry v MongoDB na organizovanie dát, ktoré do systému používateľ vkladá. Arctic tiež podporuje vytváranie snímkov skupín časových radov, umožňuje uchovávať časové rady s rozlíšením časových pečiatok 1μs a podporuje polia.

Ukázalo sa, že hoci Arctic podporuje ukladanie časových radov s jednoduchými historickými verziami, táto funkcionalita je implementovaná spôsobom ktorý nie je možné využiť na korekciu chýb v uchovaných dátach. Modul `VersionStore` je teda pravdepodobne určený skôr na uchovávanie časových radov tvorených agregátmi, či iným spôsobom spracovaných, než na uchovávanie surových záznamov. Rozhranie systému Arctic obsahuje dve funkcie na pridávanie záznamov do časového radu. Sú to funkcie `append` a `write`, ktoré pri svojom vykonávaní automaticky vytvárajú novú verziu daného časového radu. Ak však pomocou funkcie `append` vložíme do časového radu f dvojicu (c, u), kde c je časová pečiatka a u údaj, pričom systém už obsahuje pre identifikátor verzie poslednej verzie pred aktualizáciou i záznam $(i, c, f(i, c))$, systém sa dostane do nekonzistentného stavu a pri dopyte na novovytvorenú verziu budú výsledkom tak dáta obsahujúce dvojicu (c, u) ako aj záznam $(c, f(i, c))$. Funkciou `append` teda nie je možné korigovať a prepísať hodnotu údaju v zázname k existujúcej časovej pečiatke. Na druhej strane, funkcia `write` sa používa na kompletné prepísanie dát, to znamená, že ak je zavolaná na ľubovoľný časový rad f s dvojicou (c, u), po jej ukončení bude výsledkom dopytu na poslednú verziu časového radu f časový rad, ktorého veľ-

kosť definičného oboru je 1 a obsahuje iba jeden záznam - vložený záznam (c, u). Táto funkcia teda explicitne vymaže všetky záznamy s časovou pečiatkou inou, ako majú záznamy v jej argumente. Aby korekcia, ktorú chce používateľ vykonať prebehla správne, musel by používateľ najskôr dopytom na čítanie získať všetky záznamy poslednej verzie časového radu, prepísať tie, ktoré chce skorigovať a potom nanovo zapísať všetky tieto záznamy. Takýto spôsob práce nie je praktický a pre časové rady s väčším množstvom záznamov v poslednej verzii je nerealizovateľný. Preskúmali sme teda interné súčasti modulu `VersionStore` aby sme zistili, či je praktické upraviť ho tak, aby korekciu umožňoval.

Ukázalo sa, že systém interne rozdeľuje vkladané dáta do tzv. kusov (chunks), ktoré po jednom ukladá do dokumentov v rámci MongoDB. Zároveň, v špeciálnych dokumentoch ukladá informácie o jednotlivých verziách časového radu. Tieto dokumenty obsahujúce kusy a dokumenty s verziami sú navzájom prepojené smerníkmi. Arctic považuje každý záznam časového radu za riadok v imaginárnej tabuľke a jednou z informácií, ktoré ukladá v dokumentoch pre verzie je práve počet riadkov, ktorý obsahuje daná verzia. Počas písania práce bola vydaná verzia systému Arctic, ktorá modifikuje určité vlastnosti smerníkov, konkrétne bola pridaná podpora smerníkov, ktoré smerujú z dokumentu s verziou do dokumentov s kusmi, čo je opačný smer ako bol používaný dovtedy, naše pozorovania sú však platné pre oba spôsoby fungovania. Pre každý kus systém ukladá aj hash jeho obsahu, aby bolo možné automaticky vykonávať deduplikáciu a detegovať konfliktné zápisy. MongoDB totiž poskytuje operátor, ktorý buď aktualizuje dokument určený nejakým kľúčom, alebo dokument pre daný kľúč vytvorí. Vďaka tomuto operátoru Arctic pridáva nové a upravuje existujúce dokumenty s kusmi rovnakým spôsobom.

Funkcia `append` je implementovaná nasledujúcim spôsobom. Dáta, ktoré sú pridávané sa vložia ako jeden kus do nového dokumentu, ktorý sa špeciálne označí. Pri takomto volaní je teda zmenený jeden dokument s verziou a jeden dokument s kusom. Novovytvorený dokument s kusom ukazuje na dokument verziou, ktorá ako posledná vytvárala neoznačené dokumenty. Ako bolo uvedené, pri dopytoch na novšie verzie sa používajú na orientáciu údaje o počte zapísaných riadkov a označené dokumenty, ktoré tieto verzie vytvorili na dokumenty týchto verzií v skutočnosti neobsahujú smerník. Ak množstvo dát zapísaných v postupnosti za sebou volaných funkcií `append`, z ktorých všetky vytvárali tieto špeciálne označené dokumenty s kusmi, prerastie nastavenú hodnotu, respektíve dĺžka tejto postupnosti prerastie nastavený limit, nastane vo volaní funkcie, ktorá toto prekročenie spôsobila tzv. kompakcia. Tá spočíva v tom, že sa najskôr prečítajú určité časti z dokumentov, obsahujúcich všetky kusy, patriace do poslednej verzie. Vďaka tomu systém identifikuje označené dokumenty, z ktorých následne prečíta všetky dáta uloženého časového radu, nachádzajúce sa v nich. Tieto dáta sa nanovo skomprimujú a zapíšu ako kusy normálnej veľkosti bez označenia. Ak sú

použité smerníky z dokumentov kusov do dokumentov s verziami, je nutné upraviť toto pole smerníkov v každom z dokumentov obsahujúcim kus patriaci do poslednej verzie, ktorý predtým nebol označený, aby sa tak zaregistroval do novej verzie. Pri upravovaní smerníkov sa kvôli deduplikácii tiež prečítajú hodnoty hashov z úplne všetkých dokumentov obsahujúcich kusy daného časového radu. V úložisku po tejto operácii zostanú aj pôvodné, špeciálne označené dokumenty, aby bolo možné správne používať verzie vytvorené predchádzajúcimi volaniami `append` v uvedenej postupnosti. Pri práci so špeciálne označenými dokumentami používa Arctic počty záznamov vo verziách, čo je práve mechanizmus, ktorý v sebe implicitne zahŕňa predpoklad o tom, že jednotlivé vkladané záznamy majú rôzne časové pečiatky a riadky sa navzájom neprepisujú. Uložené dokumenty s kusmi ani dokumenty s verziami v sebe neobsahujú vôbec žiadnu informáciu o časových pečiatkach, ktoré sa v nich nachádzajú. Dôsledkom toho je aj to, že pri dopyte na čítanie nejakého intervalu časových pečiatok Arctic v skutočnosti číta všetky dáta a až potom aplikuje filter časových pečiatok. Znamená to však tiež, že ak by funkcia `append` mala podporovať prepisovanie časových pečiatok, bolo by vždy nutné čítať všetky dáta z daného časového radu, identifikovať dokumenty s kusmi, obsahujúcimi prepísané časové pečiatky a tie nanovo zapísať. Zjavne, takýto spôsob fungovania je veľmi neefektívny, keďže vykonáva presne to isté, čo by vykonal používateľ, ak by chcel korekciu vykonať pomocou funkcie `write`. Naviac, takýto spôsob prepisovania v kombinácii s podporou pôvodných smerníkov z dôvodu kompatibility vytvára potrebu pridania ďalšieho mechanizmu, upravujúceho význam smerníkov tak, aby bolo možné správne vykonať nasledujúce zápisy a kompakcie. Jedným zo spôsobov ako to docieliť je pri každom detegovanom prepísaní spustiť kompakciu, čo je však drahá operácia.

Limitujúcim faktorom je tiež absencia globálnych transakcií s ACID garanciami v MongoDB, čo spôsobuje, že Arctic musí vykonávať viacero kontrolných čítaní, či ošetrovať niektoré problémy spojené s konkurentným zápisom, ktoré ošetruje vykonaním predčasnej kompakcie, čo je, ako sme uviedli, drahá operácia.

Dospeli sme teda k záveru, že bez výrazných zmien a kompletného prerobenia spôsobu ukladania časových radov do dokumentov v MongoDB nie je modul `VersionStore` upraviteľný tak, aby podporoval korekcie historických údajov. Za takýchto podmienok snaha upraviť systém stráca význam, keďže pri kompletnom prerábaní logiky ukladania môže byť vhodnejšie zvoliť ako úložisko iný databázový systém, ktorý poskytuje lepšie garancie. Nový systém by teda nemal s pôvodným takmer nič z podstatných súčastí spoločné. Naviac, Arctic nepodporuje ani základné agregácie, čo ho spolu s uvedenými vlastnosťami diskvalifikovalo z ďalšieho skúmania či snahy o jeho rozšírenie.

Na základe výsledkov experimentu vykonaného v bakalárskej práci a taktiež skúmania systému Arctic sme sa rozhodli preskúmať možnosť rozšíriť existujúci systém OpenTSDB [12] tak, aby podporoval uchovávanie historických verzií. Hoci tento systém

nezvládol v testovacom prostredí vykonať najväčšie agregácie, v testoch ktoré úspešne splnil sa ukázal ako rýchlejší v porovnaní s ostatnými testovanými systémami.

2.1.2 OpenTSDB

OpenTSDB je systém na správu časových radov napísaný v jazyku Java a je vybudovaný na NoSQL úložisku Apache HBase [24], ktoré je založené na modeli Google BigTable [8]. Tento systém je aktívne podporovaný a je stále vo vývoji. OpenTSDB je pomerne rozšírený a stal sa de facto štandardom v uchovávaní časových radov. Podporuje tagy a tiež poskytuje základnú sadu agregácií [17, 6] vrátane agregácií cez viacero časových radov s obmedzením, že takto agregované časové rady musia mať spoločný názov a líšiť sa množinou tagov.

Úložisko HBase, ktoré OpenTSDB používa, je stĺpcovo orientovaná NoSQL databáza používajúca LSM strom [19], poskytujúca určité garancie konzistentnosti a zamykanie riadkov. OpenTSDB ukladá všetky záznamy časových radov v jednej tabuľke, využívajúc vlastnosť usporiadania a kolokácie kľúčov riadkov v HBase. Okrem toho OpenTSDB udržuje tabuľku s mapovaním názvov časových radov (v terminológii OpenTSDB meraní) a tagov na `UID` (unikátne identifikátory). Vyhýba sa tým kolíziám pri hashovaní a za cenu drahého čítania pri identifikovaní názvu časového radu spolu s tagmi šetrí priestor využitý kľúčmi riadkov v HBase a taktiež sieťovú záťaž, vzhľadom k tomu, že pri každom dopyte do HBase sa tento kľúč posiela a aj vracia vracia vo výsledku. Samotné kľúče riadkov v tabuľke s hodnotami majú nasledujúci tvar, pričom každá zo súčastí je preložená na svoj číselný `UID`.

```
[SOL']<NÁZOV_MERANIA><ČASOVÁ_PEČIATKA><TAG_K1><TAG_V1>[...<
   TAG_KN><TAG_VN>]
```

Voliteľná soľ slúži na zlepšenie distribuovania zápisov v clusteri. V prípade, že nie je použitá, všetky zápisy do časových radov so spoločným názvom merania by boli smerované do jedného nodu. Množstvo soli je konfigurovateľné, aby distribuovaním nedochádzalo k prílišnému spomaleniu čítania a agregácií cez viacero časových radov. OpenTSDB pracuje s rozlíšením 1ms, nekladie požiadavky na periodicitu ukladaných časových radov a využíva optimalizáciu zníženia počtu riadkov a zníženia počtu stĺpcov.

Zníženie počtu riadkov znamená, že miesto toho, aby jeden riadok (záznam) v databáze obsahoval iba jeden záznam časového radu, jeden riadok obsahuje viacero záznamov časového radu. Obyčajne sú to záznamy, ktorých časové pečiatky patria do časového intervalu (okna), ktoré je priradené danému riadku. Cieľom tejto optimalizácie je znížiť počet čítaní z databázy pri dopytoch na záznamy z intervalov časových pečiatok a čítať väčšie celky. V prípade periodických časových radov takáto optimalizácia pomáha ušetriť priestor na ukladanie časových pečiatok, keďže ich je možné

dopočítať z pozície v rámci okna a pečiatky jednotlivých záznamov nie je nutné explicitne uchovávať. V rámci kľúča pre riadok je potom použitá začiatočná časová pečiatka okna.

Zníženie počtu stĺpcov je nadstavba predchádzajúcej optimalizácie. Využíva možnosť zlúčiť stĺpce získané predchádzajúcou optimalizáciu do vlastnej dátovej štruktúry, obsahujúcej potrebné metadáta a túto štruktúru uložiť do jedného stĺpca vo forme BLOB (binary large object). Použitá databáza potom získava charakter key-value úložiska, keďže okrem identifikátoru riadku sa nepoužíva žiaden iný identifikátor štruktúry poskytovanej databázou. Uloženú štruktúru je ďalej možné komprimovať spôsobom vhodným pre dáta časových radov a tým ešte výraznejšie ušetriť priestor a množstvo prenášaných dát.

Časové okná v OpenTSDB sú konštantnej logickej veľkosti 1 hodina, čo je pre OpenTSDB maximálne 3 600 000 záznamov. OpenTSDB sa snaží dôležitý stav nebufferovať v pamäti, ale priamo ho písať do HBase. Každý prichádzajúci záznam je teda najskôr zapísaný do nového stĺpca v riadku jeho časového okna a až neskôr sa celý riadok z HBase prečíta a uplatní sa naň zníženie počtu stĺpcov a kompresia. Táto operácia je nazývaná kompakcia. Takýto prístup garantuje zachovanie dát, no zároveň výrazne spomaľuje rýchlosť vkladania záznamov. Existujúce rozšírenia, ktoré záznamy vhodným spôsobom bufferujú v pamäti a do HBase zapisujú až po kompakcii boli schopné rádovo zvýšiť rýchlosť zápisu a priepustnosť [12] pričom používali improvizovaný write-ahead log, tvorený obyčajnými súbormi, uchovávanými mimo HBase. Tieto rozšírenia však nezosilňujú poskytované garancie, ani nemenia spôsob uchovávania dát OpenTSDB a pre používateľa sú, až na rýchlosť vkladania, transparentné.

Formát kľúčov umožňuje identifikovať riadok, ktorý obsahuje záznam konkrétneho časového radu s určitou časovou pečiatkou v konštantnom čase, keďže hranice časových okien sú vopred známe. To, že najvýznamnejšie bity kľúča obsahujú informáciu o `UID` názvu merania a časovej pečiatke zabezpečuje kolokáciu záznamov, ktoré majú malý rozdiel časových pečiatok a patria do súvisiacich časových radov. Tým je zabezpečené ich efektívne čítanie pomocou skenovania, teda čítania riadkov z intervalu kľúčov v HBase, s minimálnou amplifikáciou čítania. Umiestnenie tagov v kľúči taktiež zabezpečuje efektívnu agregáciu cez viacero súvisiacich časových radov, opäť s použitím skenovania s filtrom v HBase. Takéto správanie je v HBase štandardné v prípade, že dopyt na čítanie špecifikuje množinu tagov, ktorú spĺňa viacero časových radov. Dátový model OpenTSDB je založený práve na takejto forme organizácie. Keďže OpenTSDB identifikuje časový rad pomocou názvu merania a tagov, predpokladá, že agregácie cez viacero časových radov používateľ vykonáva len pre súvisiace časové rady, teda tie, ktoré majú spoločný názov merania. Vďaka uvedenému usporiadaniu kľúča je zabezpečené, že takéto navzájom súvisiace časové rady sú kolokované a je jednoduché ich časové okná čítať.

OpenTSDB nekladie dodatočné predpoklady na poradie zápisu záznamov a umožňuje záznamy dopĺňať do minulosti. Nie je možné vymazávať záznamy z minulosti a ani ich prepisovať inými hodnotami. Ak sa pri kompakcii objaví viacero záznamov s rovnakými časovými pečiatkami a s rovnakými údajmi, systém sa s tým vysporiada. Ak sa údaje líšia, dopyty na tento záznam vyvolávajú výnimku s tým, že v novších verziách OpenTSDB je možné zabezpečiť, aby systém nevyvolával výnimku, ale postupoval metódou, pri ktorej je uchovaný posledný zápis bez nejakých explicitných garancií. Aj pri štandardnej konfigurácii, ktorá implicitne predpokladá nemennosť minulosti je možné, aby došlo k prepisu za určitých situácií - napr. v prípade, že kompakcia okna, do ktorého prepisovaný záznam patrí, nestihla ani raz prebehnúť. Dôvod na takéto správanie je práve v predpoklade nemennej minulosti. Systém ho síce prísne nevyžaduje a nekontroluje, ráta však s tým, že keď sa zmena minulosti vyskytne, jedná sa o chybný stav v zdroji záznamov a v takomto prípade jeho správanie nie je definované.

2.1.3 Diskusia

Po preskúmaní častí zdrojového kódu, ktoré interagujú s HBase sme dospeli k záveru, že snaha doplniť historické verzie do OpenTSDB by nepriniesla očakávané výsledky. Systém je vo svojom fungovaní priveľmi naviazaný na hlavný index, ktorý poskytuje HBase na prehľadávanie riadkov. Rovnako ako pri iných systémoch podobnej architektúry sú prílišné obmedzenia kladené na možnosti uchovávania dát časových radov dôsledkom priveľmi jednoduchého dátového modelu, ktorý je naviazaný na využitie štruktúry kľúčov key-value rozhrania použitého NoSQL úložiska. Problém spôsobuje potreba pridania ďalšej veličiny do kľúča a popritom zachovanie možnosti používať skenovanie cez interval kľúčov, čo je v rámci HBase efektívna operácia.

Zvažovali sme dve možnosti prístupu k pridaniu verzií do OpenTSDB a podporovaniu časových radov s historickými verziami.

Prístup upravujúci riadky

Prvý prístup používal už existujúcu štruktúru v riadkoch. V riadku by sa okrem doterajšej štruktúry uchovávali aj nové verzie záznamov, ktoré by sa po čítaní do pamäte aplikovali. Tento prístup má dve nevýhody. Vyžaduje doplniť zamykanie riadkov tak, aby pri konkurentnom upravovaní rôznych záznamov v jednom okne boli prideľované správne identifikátory verzií a tiež koordinovať identifikátory verzií medzi jednotlivými oknami, uloženými v rôznych riadkoch, aby bolo možné rekonštruovať predchádzajúce stavy celého časového radu v systéme. OpenTSDB pre svoju prácu využíva rozhranie HBase poskytujúce možnosť atomicky modifikovať jeden riadok (pridať alebo odstrániť stĺpec či zmeniť hodnotu v stĺpcoch), čo by pri takejto potrebe udržiavať konzistentné informácie o verziách nebolo dostačujúce. Taktiež, riadky v HBase majú obmedzenú

veľkosť, čiže pri uvedenom prístupe by bolo nutné pri preplnení rozdeliť dátovú štruktúru do viacerých riadkov a pridať do kľúčov riadkov ďalšiu veličinu. Táto reorganizácia tiež spôsobuje istú ťažkosť s dopytmi na aktuálnu verziu. Ak nastane rozdelenie, systém môže buď prerozdeliť existujúce dáta do štruktúr v novom aj pôvodnom riadku, alebo určiť identifikátor verzie, od ktorej sú záznamy ukladané do novej štruktúry tak, aby všetky existujúce záznamy zostali v pôvodnej. Keďže aktuálne verzie jednotlivých záznamov v rámci jedného okna môžu mať rôzne identifikátory verzií, systém pri dopyte na aktuálnu verziu nemá ako zistiť, v ktorom z použitých riadkov pre dané okno sa aktuálne verzie záznamov z dopytovaného intervalu nachádzajú a musí prečítať všetky. Toto vedie k prílišnej amplifikácii čítania. Tiež sa stráca hlavná očakávaná vlastnosť tohoto prístupu, ktorou je udržanie všetkých záznamov z jedného časového okna v rámci jedného spoločného celku - riadku v databáze.

Prístup s nemennými riadkami

Druhá možnosť fungovala na báze pridávania úprav ako nových riadkov. Tento prístup je z pohľadu použitých dátových štruktúr jednoduchší ako predchádzajúci, keďže nepotrebuje upravovať štruktúru v riadku a vystačí si iba s pridaním verzie do kľúča. Pri každej aktualizácii časového radu by bol vytvorený nový riadok, ktorého kľúč by mal na konci aktuálny identifikátor verzie ukladaného časového radu.

Takýto prístup má opäť dva problémy. Ak by sa do nových riadkov pridávali iba upravené časti časového okna, zachovala by sa možnosť čítať celý pôvodný interval riadkov a následne ich v pamäti zlúčiť, no stratili by sa výhody optimalizácie zníženia počtu riadkov. Hrozilo by, že réžia potrebná na spracovanie malých zápisov takýmto spôsobom by bola podobná, ako keby takáto optimalizácia vôbec nebola používaná, keďže napríklad postupné pridávanie záznamov pre každú časovú pečiatku okna by vytvorilo riadok pre každý jeden záznam. Takýto scenár vkladania nových záznamov by sa dal ošetriť zlučovaním operácií vkladania tak, aby sa vždy vložil dostatočne veľký počet záznamov, no pracovať s takýmto zlučovaním pri úprave existujúcich záznamov nie je praktické, keďže pre klienta môžu mať jednotlivé verzie s úpravami konkrétny význam. Taktiež, pri dopyte na aktuálnu verziu, respektíve verziu do konkrétnej verzie by bolo opäť nutné prečítať potenciálne všetky riadky patriace k danému oknu, vrátane tých, ktoré sa vo výsledku neobjavia, keďže samotný identifikátor verzie nenesie informáciu o časových pečiatkach, ktorých sa daná úprava dotkla. Na druhej strane, pri úpravách záznamov patriacich do viacerých časových okien je takýto prístup jednoduchší, keďže nepotrebuje používať zamykanie riadkov. Riadky sú totiž po vytvorení nemenné a ani pri vkladaní nových ani úprave existujúcich záznamov nie je potrebné čítať existujúce dáta a zamykať riadky. Aj pri konkurentnom zápise od viacerých klientov stačí jednotlivých klientov synchronizovať pomocou metódy na získanie identifikátoru verzie a

potom môžu pracovať bez toho, aby sa vzájomne obmedzovali.

Ak by sa pri úprave kopírovali do nových riadkov celé časové okná (ich obsah v predchádzajúcej verzii), zachovalo by to zložitosť dopytov na intervaly časových pečiatok, ktoré sa zmestia do jedného okna no zvýšilo by to pamäťové nároky systému, keďže uchovávané dáta by bolo potrebné uchovávať vo viacerých kópiách. Opäť, určité zlepšenie je možné dosiahnuť zlučovaním operácií vkladania, čo by zabránilo pri vkladaní záznamov jednotlivo vytvoriť pre jedno časové okno rovnaký počet riadkov ako obsahuje záznamov, pre operácie upravujúce existujúce záznamy to rovnako ako v prípade predošlého prístupu nemusí byť vhodné. Týmto prístupom by sa tiež stratila možnosť dvoch naraz zapisujúcich klientov vytvárať svoju verziu bez synchronizácie pri vytváraní riadkov, keďže pri vytváraní nového riadku je nutné prečítať predchádzajúcu verziu upravovaného časového okna. Tento prístup uľahčuje dopyty na verziu časového radu do konkrétnej verzie a s pridaním identifikátorov verzie pre každý záznam v dátovej štruktúre aj na dopyty na konkrétnu verziu časového radu, keďže identifikátor riadku s odpoveďou na takýto dopyt je skonštruovateľný vďaka svojmu formátu priamo a pre odpoveď na takýto dopyt stačí z časového okna prečítať iba jeden riadok. Taktiež, tento prístup si aj pre dopyty na aktuálnu verziu vystačí s čítaním najviac jedného riadku pre každé okno, nachádzajúce sa v dopytovanom intervale časových pečiatok. Nevýhodou tohoto prístupu, okrem uchovávania duplikátov je aj nutné zbytočné čítanie veľkého množstva predchádzajúcich verzií pri agregáciách medzi časovými radmi radmi patriacimi k rovnakému meraniu a tiež, podobne ako v prístupe bez kopírovania, pridaná zložitosť čítania okien, ktoré sa v konkrétnej verzii neupravovali a je potrebné identifikovať riadok obsahujúci ich predchádzajúcu verziu.

V konečnom dôsledku sme po zvážení týchto možností usúdili, že architektonický princíp OpenTSDB, ktorého variácie sú rozšírené v rôznych iných systémoch na správu časových radov nie je vhodný na pridanie funkcionality uchovávania historických verzií a je potrebné použiť iný, flexibilnejší dátový model, ktorý nie je závislý na ukladaní metadát v kľúčoch indexovaných nízkoúrovňovým úložiskom.

Kapitola 3

BTrDB

Účelom tejto kapitoly je uviesť prehľadný technický popis systému BTrDB, na ktorý sa odvolávame v neskorších kapitolách.

3.1 Technický popis

BTrDB [2] je distribuovaným úložiskom, ktoré je súčasťou nástroja SmartGridStore [22], určeného na prácu s dátami pochádzajúcimi z tzv. smart grid elektrických sietí. Smart grid sú siete, ktoré používajú technológie digitálnej komunikácie na detekciu a reakciu na lokálne zmeny spotreby energie či výpadky. Dáta z týchto sietí pochádzajú z tzv. phasor measurement units (PMU), čo sú zariadenia, vyhodnocujúce amplitúdu napätia a prúdu, frekvenciu siete a fázový uhol v danom mieste v konkrétnych časoch. Výsledkom takýchto meraní sú tzv. synchrónne fázory (synchrophasors) [1]. Merania sa dejú vo vysokej frekvencii a nie je vylúčená zmena poradia záznamov pri ich zapisovaní do úložiska. Samotný systém SmartGridStore sa skladá z viacerých modulov - menovite už spomínaného úložiska BTrDB, nástroja na administráciu clusteru, nástroja na vizualizáciu uložených časových radov Mr.Plotter [16], http servera na interakciu s databázou, analytickým nástrojom DISTIL [3] a rôznymi ingress daemonmi používanými na vkladanie a predspracúvanie dát z PMU.

SmartGridStore je poskytovaný v kontajnerizovanej podobe, využívajúcej technológie Docker a Kubernetes, čím uľahčuje nasadzovanie v clusteroch. Systém SmartGridStore je vo svojej podstate distribuovaný, keďže používa nástroje, ktoré na svoj chod potrebujú replikáciu. Z tohoto dôvodu sme používali poskytované vývojové prostredie, ktoré vytvára virtuálny cluster v rámci jedného počítača a umožňuje interakciu s plnohodnotne bežiacim systémom bez potreby konfigurovať úplne všetky komponenty, čo by bolo potrebné vykonať pri ostrej prevádzke. Jednotlivé komponenty systému spolu komunikujú prostredníctvom frameworku na vzdialené volanie procedúr gRPC [20]. Tento prístup unifikuje komunikáciu medzi jednotlivými komponentami a tiež v rámci

jedného komponentu keď je spustený v distribuovanom režime.

Pre potreby našej práce je podstatným komponentom samotné úložisko BTrDB. V publikácii, v ktorej bol BTrDB predstavený je popísaná verzia 3, ktorá medzičasom prestala byť podporovaná a bola nahradená verziou 4, ktorá so sebou priniesla viacero zmien. V práci sa venujeme práve verzii 4. BTrDB je napísaný v jazyku Go [10] a vo verzii 4 používa dve externé závislosti. Menovite sú nimi distribuované objektové úložisko Ceph [26] (spolu so službou Reliable Autonomic Distributed Object Store RADOS [27]) poskytujúce prístup k virtuálnemu blokovému médiu a key-value databáza etcd [13] (BTrDB vo verzii 3 z článku používa NoSQL databázu MongoDB a vo verzii 5 je plánovaný prechod na databázu, podporujúcu SQL rozhranie). Etcd je key-value databáza podporujúca usporiadanie kľúčov, vyhľadávanie pomocou prefixu kľúču a poskytujúca ACID garancie na minitransakcie. Minitransakcia je dopyt vo forme konštrukcie `if then else`, ktorý vyhodnotí na základe stavu databázy podmienky v bloku `if` a na základe výsledku vykoná buď blok príkazov `then` alebo `else` s obmedzením, že v rámci jednej minitransakcie môže byť každý záznam ⟨kľúč,hodnota⟩ modifikovaný najviac raz. Etcd poskytuje tiež gRPC API, čiže priamo zapadá do architektonického modelu BTrDB. BTrDB ukladá samotné dáta časových radov do Ceph úložiska (vo verzii 3 bolo možné na ukladanie dát použiť aj lokálny súborový systém, podpora tejto možnosti už skončila) a úložisko etcd využíva na ukladanie metadát o časových radoch. BTrDB tiež používa nástroje ako Opentracing [18] na monitorovanie svojej aktivity a stavu, tieto však nie sú pre logický model dôležité a priamo neovplyvňujú správanie BTrDB vzhľadom na spracúvanie časových radov.

BTrDB funguje ako cluster, pričom na svoju koordináciu používa koncept, ktorý nazýva MASH (Master Allocation by Stable Hashing). Táto štruktúra obsahuje informácie o rozdelení priestoru univerzálne unikátnych identifikátorov (UUID) časových radov medzi nodami clusteru (endpointami). BTrDB implementuje vlastnú viacvrstvovú cache urýchľujúcu bežné prístupy k dátam a používa journaling zabezpečujúci obnovu dát pri poruche endpointov.

3.2 Architektúra systému

V tejto časti je popísaná vysokoúrovňová architektúra systému BTrDB pomocou pohľadov použitia, implementácie, nasadenia, procesov a logického modelu.

3.2.1 Pohľad použitia

BTrDB pracuje s časovými radmi so špeciálnymi hodnotami a oddeľovačmi a podporuje na nich nasledovné abstraktné operácie:

- **InsertValues(UUID, [(time, value)])** Aktualizuje časový rad so špeciálnymi

hodnotami f, identifikovaný $UUID$ tak, že doň vloží zoznam záznamov **[(time, value)]**. Jednotlivé záznamy majú časové pečiatky, na ktorých nie je definovaná aktuálna verzia f.

- **GetRange(UUID, StartTime, EndTime, Version)** Vracia časový rad do verzie $Version$ časového radu so špeciálnymi hodnotami, identifikovaného svojim $UUID$, ktorého definičný obor je zúžený na interval $\langle StartTime, EndTime)$

- **GetStatisticalRange(UUID, StartTime, EndTime, Version, Resolution)** Používa sa na vykonávanie agregácií po skupinách cez jeden časový rad. Táto operácia rozdelí interval $\langle StartTime, EndTime)$ na disjunktné podintervaly (okná) dĺžky $2^{resolution}$. Pre každé z týchto okien vytvorí štatistický záznam, obsahujúci agregácie záznamov časového radu do verzie $Version$ časového radu so špeciálnymi hodnotami, identifikovaného $UUID$, ktorých časová pečiatka patrí do daného podintervalu. Výstupom je zoznam štatistických záznamov, zoradený podľa začiatočnej časovej pečiatky prislúchajúcich podintervalov. V prípade, že v niektorom z okien nie je žiadna časová pečiatka, pre ktorú by bol daný časový rad do verzie $Version$ definovaný, výstupom pre takéto okno je tzv. diera - prázdny štatistický záznam, slúžiaci na signalizáciu absencie záznamov v pôvodnom časovom rade so špeciálnymi hodnotami. Ak dĺžka vstupného intervalu $\langle StartTime, EndTime)$ nie je deliteľná dĺžkou zvoleného podintervalu, systém zabezpečí, aby posledné okno začínajúce v rámci intervalu nepresahovalo za $EndTime$ tým, že miesto $EndTime$ použije vo výpočte koncovú časovú pečiatku posledného okna, ktoré sa celé zmestilo do pôvodného intervalu $\langle StartTime, EndTime)$.

- **GetNearestValue(UUID, Time, Version, Direction)** Vráti záznam $(c, f'(c))$, pričom f je časový rad so špeciálnymi hodnotami, identifikovaný $UUID$, a f' je časovým radom do verzie $Version$ časového radu so špeciálnymi hodnotami f. Ak $Direction = True$ potom c je taká časová pečiatka, že $f'(c)$ je definované, $c < Time$ a pre všetky ostatné záznamy f' s časovou pečiatkou c' platí buď $c' < c$ alebo $c' \geq Time$. Ak $Direction = False$ tak c je taká časová pečiatka, že $f'(c)$ je definované, $c > Time$ a pre všetky ostatné záznamy f' s časovou pečiatkou c' platí $c' > c$ alebo $c' \leq Time$.

- **ComputeDiff(UUID, FromVersion, ToVersion, Resolution)** Vracia zoznam disjunktných intervalov časových pečiatok $[\langle a, b)]$. Pre každý interval $\langle a, b)$ z výsledného zoznamu platí, že existuje časová pečiatka c patriaca do tohoto intervalu a identifikátor verzie i patriaci do intervalu $\langle FromVersion, ToVersion\rangle$ taký, že $(i, c, f(i, c))$ je záznam časového radu so špeciálnymi hodnotami f identifikovaného $UUID$. Parameter $Resolution$ určuje minimálnu dĺžku intervalov

časových pečiatok vo výsledku a tým zrýchľuje výpočet. Najmenšia dĺžka každého z výsledných intervalov je teda $2^{resolution}$ a systém sa snaží do výsledku dávať najmenšie možné intervaly. Nech množina T obsahuje také časové pečiatky c', že existuje identifikátor verzie $j \in \langle FromVersion, ToVersion \rangle$ v ktorom je $f(j, c')$ definované. Intervaly vo výsledku pokrývajú celú množinu časových pečiatok T.

- **DeleteRange(UUID, StartTime, EndTime)** Aktualizuje časový rad f, identifikovaný $UUID$ tak, že pre všetky časové pečiatky c z intervalu $\langle StartTime, EndTime)$ nastaví $f(j, c) = Nil$, pričom j je identifikátor novej verzie.

Operácia **DeleteRange** je z technických dôvodov implementovaná hybridným spôsobom s určitými nekonzistenciami voči definícii. Z implementačných dôvodov táto operácia v systéme dokáže priradiť hodnotu Nil celým intervalom, no konceptuálne je vzhľadom k pojmu aktuálna verzia časového radu toto správanie totožné s priradením hodnoty Nil pre každú časovú pečiatku z intervalu. Taktiež, v niektorých prípadoch sa systém správa akoby používal časové rady so súvislými verziami a operáciu **DeleteRange** implementuje jednoducho neprekopírovaním vymazaných záznamov do novej verzie miesto explicitného priradenia špeciálnej hodnoty.

Operácia **InsertValues** v BTrDB verzie 3 explicitne vytvára novú verziu v časovom rade, čiže vykonáva operáciu vloženia nových údajov. BTrDB vo verzii 4 však obsahuje vrstvu PQM (persistent queriable multiplexer), ktorá urýchľuje malé zápisy. Zabezpečuje to tým, že vkladané záznamy píše do journalu ale nezapisuje ich do hlavnej dátovej štruktúry. Až keď sa naplní buffer, alebo vyprší stanovený čas sú dáta skutočne zapísané do tejto dátovej štruktúry ako nová verzia časového radu. Pri dopytoch na čítanie sa záznamy z bufferu zlučujú s tými, čo už sú uložené v hlavnej dátovej štruktúre, čiže výsledky sú vždy aktuálne. Tento prístup minimalizuje amplifikáciu zápisov a výrazne zlepšuje priepustnosť systému. Podobný prístup bol zvolený aj v spomínanej úprave systému OpenTSDB, keď sa ukázalo, že zapisovanie prichádzajúcich záznamov jednotlivo je nevhodné. PQM teda spôsobuje určité oddelenie logickej verzie a skutočnej zapísanej verzie. Vzhľadom k tejto zmene má operácia InsertValues z verzie 4 novú sémantiku. Keďže nastalo oddelenie volania **InsertValues** a skutočného zápisu do hlavnej dátovej štruktúry spolu s vytvorením novej verzie, bola pridaná operácia **Flush(UUID)**, ktorá spoľahlivo vyprázdni buffer PQM pre časový rad so špeciálnymi hodnotami identifikovaný $UUID$ a vytvorí z jeho záznamov novú verziu stromu.

Uvedené základné operácie sú zapuzdrené vo viacerých funkciách nachádzajúcich sa v knižniciach pre jednotlivé programovacie jazyky. Okrem týchto operácií na časových radoch podporuje BTrDB aj administrátorské príkazy, ktoré napr. odstránia celý časový rad so špeciálnymi hodnotami zo systému. BTrDB ukladá aj rôzne metadáta k časovým radom, aby sa uľahčila ich organizácia a vyhľadávanie v nich. Podporuje

tzv. collections, čo sú množiny časových radov a podporuje tiež tagy a anotácie. Pri použití tagov síce platí, že ich množina priradená k určitému časovému radu je nemenná, nemusia však slúžiť na identifikáciu časového radu. Každý časový rad má totiž v systéme svoj UUID nezávislý od množiny tagov, ktorý je sám postačujúci na vyhľadanie konkrétneho časového radu. Všetky tieto metadáta sa používajú na organizáciu uložených časových radov tak, aby bolo možné vyhľadať ich UUID. Konceptuálne je ale práca s metadátami úplne oddelená od spracúvania samotných dát časových radov. Je to zjavné z toho, že spôsob ukladania a organizácie metadát sa už menil (použité úložiská MongoDB a etcd). Pre potreby skúmania dátového úložiska sa teda nebudeme zaoberať fungovaním metaúložiska, vyhľadávaním či prácou s metadátami, ale priamo použijeme unikátny identifikátor časového radu na prácu s ním.

BTrDB pracuje s množinou časových pečiatok, ktorých dátový typ je 64 bitové celé číslo so znamienkom a množinou údajov, ktorých dátový typ je 64 bitové číslo s pohyblivou desatinnou čiarkou. Rozlíšenie systému je 1ns a dátový typ identifikátora verzie je 64 bitové celé číslo bez znamienka.

Z hľadiska agregácií podporuje BTrDB prostredníctvom operácie **GetStatisticalRange** radové aj hodnotové agregácie cez jeden časový rad, poskytované pomocou agregácií po skupinách. Na zložitejšie analýzy sa používa externý nástroj DISTIL.

3.2.2 Implementačný pohľad

BTrDB je naprogramovaný v jazyku Go, ktorý organizuje zdrojové súbory do balíčkov (packages), obyčajne označených názvom priečinku v ktorom sa nachádzajú. V nasledujúcich častiach rozoberieme úlohu jednotlivých balíčkov v systéme BTrDB a ich usporiadanie. Prostredie pre vývoj v jazyku Go pracuje v rámci pracovnej plochy (workspace), čo je priečinok obsahujúci podpriečinky `src/` a `bin/`. Priečinok `bin/` obsahuje skompilované spustiteľné súbory, zatiaľčo `src/` je koreňovým priečinkom pre projekty. Projekt býva obyčajne uložený do hierarchie priečinkov podľa názvu repozitára, v ktorom je vyhľadateľný na internete. V rámci jednotlivých zdrojových súborov sa potom balíčky adresujú celou touto cestou, aby bolo možné udržiavať mapovanie medzi lokálne uloženými súbormi a ich online umiestnením pre potreby balíčkovacích nástrojov. Koreňový priečinok projektového repozitáru BTrDB je

```
github.com/BTrDB/btrdb-server
```

V ďalších častiach budú jednotlivé súbory a priečinky označované relatívne voči tomuto koreňovému priečinku. Taktiež, ak nie je uvedené inak, názov balíčku vynechávame a miesto neho uvádzame názov priečinku s jeho zdrojovými súbormi, ktorý býva totožný s názvom balíčku. Z popisu vynechávame aj priečinky a súbory, ktoré neobsahujú zdrojový kód, prípadne obsahujú pozostatky predchádzajúcich verzií systému či iné, s implementáciou BTrDB priamo nesúvisiace programy.

Hlavný balíček je v priečinku `btrdbd/` a má názov `main`. Zabezpečuje správne spustenie inštancie BTrDB, pripojenie k loggeru, úložisku Ceph a etcd, vytvorenie databázy, zavedenie rozhraní pre používateľa a načítanie konfigurácie.

V priečinku `bte/` sa nachádza balíček s kódmi všetkých systémových chýb a štruktúrou, ktorá chyby zapuzdruje.

Balíček v priečinku `cliplugin/` obsahuje plugin do administrátorského rozhrania pre SmartGridStore.

Priečinok `qtree/` obsahuje zdrojové súbory s implementáciou a testami pre hlavnú dátovú štruktúru systému - k-árny strom, detailne popísaný v ďalších častiach.

Niekoľko balíkov sa nachádza v podpriečinkoch priečinku `internal/`. Toto umiestnenie znamená, že dané balíčky sú interné a nie sú určené na používanie v iných projektoch. Balíček `bprovider` obsahuje implementáciu rozhrania pre ukladanie BLOB na špecifikovanú adresu v úložisku. Obsahuje tiež rozhranie abstrahujúce od funkcií blokového úložiska. Balíček `bstore` obsahuje implementáciu funkcií používajúcich blokové úložisko, implementáciu vlastnej blokovej cache a linkeru, používajúceho na serializovanie ukladaných dátových štruktúr a mapovanie logických adries v smerníkoch na fyzické adresy. Balíček `configprovider` obsahuje funkcie pre prácu s konfiguráciou clusteru a jednotlivých nodov. Balíček `jprovider` obsahuje rozhranie na prácu s journalom. Balíček `mprovider` obsahuje implementáciu funkcií na prácu s metadátami a komunikáciu s etcd. Balíček `cephprovider` obsahuje implementáciu nízkoúrovňových funkcií spolupracujúcich s blokovým úložiskom Ceph. Implementuje rozhranie z balíčkov `bprovider` a `jprovider` a obsahuje ďalšiu cache. Balíček `rez` obsahuje nízkoúrovňový menežér prostriedkov, ktorý obmedzuje záťaž na úložisko pri konkurentných dopytoch.

V priečinku `grpcinterface/` sa nachádza balíček obsahujúci gRPC interface a súbor `btrdb.proto` obsahujúci definície dátových štruktúr a vzdialených volaní funkcií pre použitie v Protocol Buffers - serializačnom formáte používanom frameworkom gRPC.

V samotnom koreni adresára je umiestnený balíček `btrdb` obsahujúci zdrojové súbory a testy k centrálnym súčastiam systému. Sú to súbory `pqm.go` a `merger.go` obsahujúce implementáciu PQM a súbor `quasar.go` obsahujúci implementáciu funkcií tvoriacich centrálne rozhranie medzi úložiskom a požiadavkami od klienta.

3.2.3 Pohľad nasadenia

Pre základné použitie BTrDB ako databázy bez dodatočných nástrojov SmartGridStore je potrebné spustiť systémy Ceph a etcd. Cluster so spusteným Ceph obsahuje Ceph monitory spolu s Ceph menežérmi a niekoľko Ceph object-storage démonov. Cluster s etcd obsahuje niekoľko (SmartGridStore odporúča tri) inštancií etcd. Samotný BTrDB

môže byť tiež spustený na niekoľkých nodoch, potom je však vhodné spustiť aj nástroj apifrontend, ktorý je súčasťou SmartGridStore, ktorý zabezpečuje správne smerovanie klientských dopytov v rámci clusteru. Návody na používanie vývojového prostredia ako aj produkčného nasadenia systému odporúčajú používať linuxové systémy.

3.2.4 Logický dátový model

V tejto časti rozoberieme fungovanie základnej dátovej štruktúry, použitej v BTrDB na implementáciu uvedených abstraktných operácií nad časovými radmi. Uvedieme tiež zdôvodnenia určitých dizajnových rozhodnutí a vlastnosti tejto štruktúry v praktickom nasadení.

Dátová štruktúra

Dátový model pre časový rad so špeciálnymi hodnotami je k-árny COW (copy on write [11]) strom s označeniami verzií, staticky deliaci časovú os. Každý strom reprezentuje fixný časový úsek od -2^{59} po $3 * 2^{59}$ ns od Unix epoch (cca roky 1933 až 2079), pričom samotné dáta sú ukladané vždy len do listov, ktoré podporujú rozlíšenie 1ns. Vo vnútorných vrcholoch sa nachádzajú verziou označené smerníky na synov a taktiež predpočítané hodnotové agregácie záznamov z podstromov jednotlivých synov. V BTrDB implementácii má každý vnútorný vrchol najviac 64 synov a každý list obsahuje najviac 1024 záznamov. Statickosť delenia časovej osi znamená, že každý vnútorný vrchol delí svoj interval časových pečiatok na 64 rovnako veľkých disjunktných podintervalov, ktoré sú spravované jeho synmi. Strom je COW, čo znamená, že pri ľubovoľnej úprave sa vytvorí jeho nová kópia. Vďaka označeniam verzií smerníkov na synov nie je potrebné vytvárať kópiu celého stromu, stačí skopírovať iba zmenené listy (prípadne, keď sa pridá nová úroveň aj pridané vnútorné vrcholy) a tie vnútorné vrcholy, čo sú na ceste ku koreňu. Tento princíp COW sa nazýva path-copying. Novovytvorený koreň sa pridá do tzv. mapy koreňov a tým sa pre systém zverejní nová verzia dátovej štruktúry.

Praktické vlastnosti dátovej štruktúry

Zo statickosti rozdelenia časovej osi vyplývajú pre strom rôzne vlastnosti.

Strom má fixnú maximálnu hĺbku a čím je vrchol vo väčšej hĺbke, tým reprezentuje kratší interval časových pečiatok. Vďaka tomu, že strom delí časovú os staticky a nie na základe toho v akom poradí sú doň vkladané dáta, je možné priamočiaro lokalizovať polohu záznamu v ňom a zároveň z pozície vrcholu v strome je možné vypočítať, aký interval časových pečiatok pokrýva. Taktiež, v závislosti od periódy vstupného časového radu, ovplyvňujúcej počet záznamov v jednotlivých intervaloch si strom sám reguluje svoju hĺbku, ktorá sa tiež dá určiť iba z periódy časového radu, bez ohľadu na to, v

akom poradí sú doň vkladané dáta. Hoci systém periodickosť uchovávaných časových radov nevyžaduje, takáto vlastnosť uloženého časového radu je preň výhodná práve z týchto dôvodov, keďže umožňuje, aby sa všetky naplnené listy nachádzali v rovnakej hĺbke. Tieto vlastnosti sú dôsledkom toho, že vrchol sa mení z listu na vnútorný vrchol, až keď interval ktorý spravuje obsahuje viac záznamov, ako sa zmestí do jedného listu. Pri takejto konverzii sa záznamy posunú do listov o úroveň hlbšie a nové vnútorné vrcholy sa vytvárajú opäť len v prípade, že je niektorý z nových listov preplnený. Hoci je strom konceptuálne fixnej maximálnej veľkosti, z praktických dôvodov sa buduje priebežne a môže obsahovať listy vo vyšších vrstvách, prípadne diery - označenia neexistujúcich synov vnútorných vrcholov, ktorí signalizujú, že interval, ktorý by títo synovia spravovali, neobsahuje žiadne záznamy.

Statickosť delenia časovej osi sa využíva aj pri dopytoch na štatistické agregácie. Operácia **GetStatisticalRange** je implementovaná dvoma funkciami `Windows` a `AlignedWindows`. Funkcia `Windows` implementuje uvedenú operáciu priamo, funkcia `AlignedWindows` očakáva, že okná majú dĺžku 2^w pre nejaké w a že hodnoty **StartTime** a **EndTime** sú násobkami 2^w, vďaka čomu výpočet urýchľuje tým, že jednotlivé okná pri výpočte zarovnáva voči vrcholom stromu.

Parameter **resolution** z funkcií implementujúcich operáciu **GetStatisticalRange** určuje, do akej maximálnej hĺbky sa bude strom prehľadávať pri počítaní odpovede pre dopyt a tým umožňuje explicitne obmedziť použité rozlíšenie dát. Táto vlastnosť je dôležitá, pretože systém sa používa na ukladanie dát z PMU, ktoré sa vyznačujú veľmi vysokou frekvenciou vytvárania meraní a čítanie surových dát by preto bolo mimoriadne zaťažujúce. Takto je možné čítať vďaka predpočítaným agregáciám predspracované dáta a tým ušetriť prenos aj čítanie z disku. Výhoda tohoto prístupu sa dá demonštrovať napríklad pri vizualizácii dát časového radu z veľkého intervalu na monitore. Systémy, ktoré nepodporujú predpočítané agregácie musia kvôli vykresleniu grafu čítať všetky surové dáta a agregovať ich po skupinách, aby mohli vykresliť jednotlivé pixely, reprezentujúce podintervaly vstupného intervalu. Pri zmene vizualizácie, napríklad v prípade, že operátor priblíži pohľad na určitý podinterval pôvodného intervalu je opäť nutné prečítať surové dáta so záznamami časových radov a agregovať ich. BTrDB surové dáta čítať nemusí a pri vhodných intervaloch na vizualizáciu, keď jednotlivé pixely zobrazujú agregovanú hodnotu intervalu, ktorý práve pokrýva jeden vrchol, nie je potrebné dokonca nič počítať. Stačí iba prečítať už vypočítané výsledky agregácií.

Predpočítané agregácie vo vnútorných vrcholoch sú vždy konzistentné voči údajom, nachádzajúcim sa v podstromoch, pretože sa počítajú zároveň s úpravou stromu počas jeho prechádzania pri konštruovaní novej verzie. Pri vytváraní kópií vnútorných vrcholov na ceste ku koreňu sa prepočítajú aj agregácie z ich podstromov. Keďže prepočítavanie sa vykonáva počas kopírovania vrcholov, udržiavanie aktuálnych agregácií

nespôsobuje ďalšie náklady vo forme diskových operácií pre systém. Efektívnosť tohoto prístupu je zabezpečená aj tým, že pri ukladaní novej verzie linker najskôr uloží listy a ich fyzickými adresami nahradí dočasné logické adresy uložené vo vnútorných vrcholoch. Týmto spôsobom je logická adresácia v rámci stromu priamo fyzickou a nie je potrebné pri čítaní stromovej štruktúry prekladať medzi logickými a fyzickými adresami.

Vďaka tomu, že strom sa zverejní až po úplnom uložení, keď je jeho koreň publikovaný v mape koreňov, je možné urobiť aj rollback poslednej operácie modifikujúcej strom, hoci zapísané dáta zostanú v úložisku neodstránené. Stačí dekrementovať číslo aktuálnej verzie, čím sa zabezpečí, že systém zabudne na posledný uložený koreň stromu, ktorý pri ďalšej modifikácii v rámci mapy koreňov prepíše.

Preloženie unikátneho identifikátora časového radu spolu s jeho verziou na fyzickú adresu konkrétneho koreňa stromu je zabezpečené pomocou tzv. superblokov, čo sú špeciálne objekty na vopred určených adresách v úložisku, ktorých adresu je možné priamo vypočítať z $UUID$ a verzie dopytovaného časového radu. Nie je preto potrebné udržiavať globálnu tabuľku s mapovaním $UUID$ a verzií na fyzické adresy superblokov. Pomocou offsetov v rámci superblokov je možné získať informáciu o adrese koreňa pre vybranú verziu časového radu.

Je zjavné, že COW prístup vedie k amplifikácii písania. Táto amplifikácia je očakávaná a je minimalizovaná vlastnosťami PMU, ktoré síce vytvárajú časový rad s malou periódou, ale odosielajú namerané záznamy po skupinách. Amplifikáciu písania ďalej znižuje popísaná PQM vrstva, ktorá zlučuje zápisy v malých skupinách.

V záujme zväčšenia priepustnosti organizuje BTrDB v úložisku Ceph dáta do rôznych oblastí (pools). Samostatná oblasť je určená pre journaling a dve oblasti sú určené pre dáta vrcholov jednotlivých stromov. Vnútorné vrcholy stromov a superbloky sú ukladané v tzv. hot oblasti, kvôli potrebe častého čítania počas prechádzania stromových štruktúr. Listy sú ukladané v tzv. cold oblasti. Do oboch týchto oblastí sa serializované vrcholy stromu ukladajú v skomprimovanej podobe. Podľa pozorovaní z produkčného nasadenia systém používa 5.514 bajtov na 16 bajtový záznam, vrátane všetkých režijných nákladov súvisiacich s uchovávaní stromovej štruktúry a predpočítaných agregácií. Ceph tiež umožňuje určiť fyzické zariadenia, na ktoré sa jednotlivé oblasti ukladajú, je teda možné umiestniť hot oblasť na rýchlejšie SSD médiá a cold oblasť na pomalšie disky, čo ďalej zrýchľuje prácu systému.

3.2.5 Procesný pohľad

BTrDB je založený na princípoch SEDA (staged event-driven architecture [28]) a jeho fungovanie je rozdelené do troch fáz - prvej, zaoberajúcej sa interakciou s klientami, druhej pracujúcej s dátovými štruktúrami v pamäti a tretej, poskytujúcej perzistentné

úložisko pre dátové štruktúry.

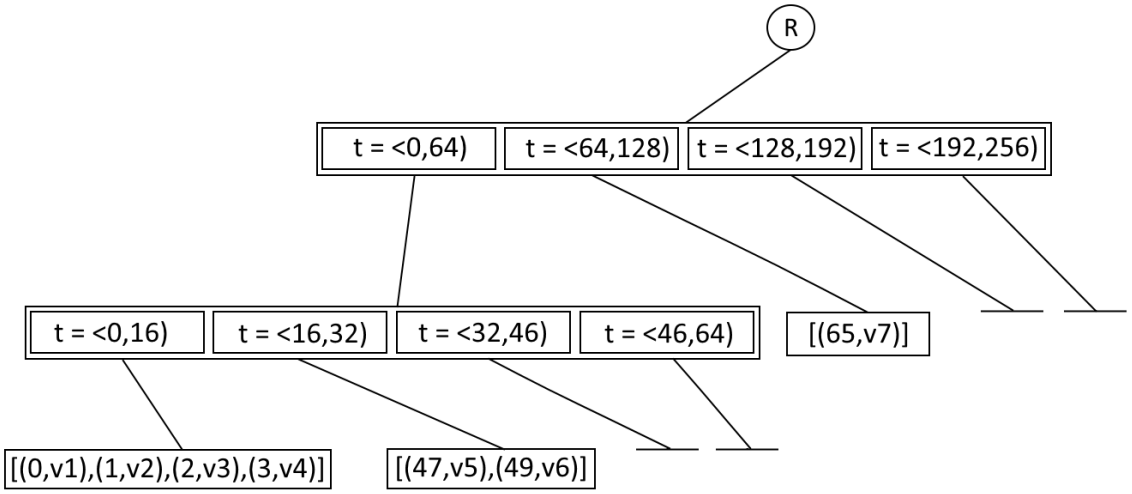
Fungovanie systému ilustrujeme na dvoch dopytoch, poukazujúcich na správanie systému a funkcie upravujúce strom pri zápise a pri čítaní. Rozoberieme operácie **InsertValues** a **ComputeDiff**, ktoré dobre popisujú dianie v systéme. Vzhľadom k tomu, že ich argumentom je $UUID$ časového radu, tieto dopyty sú vytvárané až potom, ako klient tento identifikátor získa z metaúložiska, ktoré však môže byť poskytované rôznymi spôsobmi, preto sa dopytom naň v tejto časti nevenujeme a popisujeme iba dopyty interagujúce s hlavnými dátovými štruktúrami. Najskôr popisujeme zjednodušený pohľad na vybrané operácie, rozoberajúci iba ich vplyv na štruktúru stromu a následne podrobne rozoberáme prechod systémom pri vykonávaní dopytov na tieto operácie.

Zjednodušená ilustrácia InsertValues

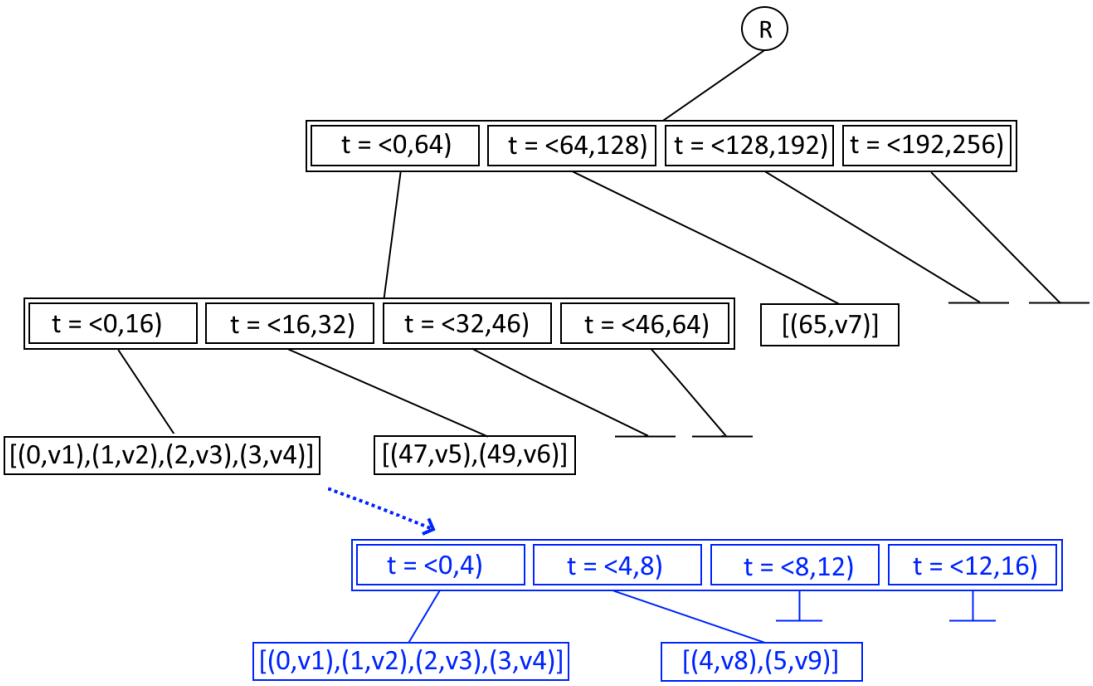
Pre jednoduchosť uvažujme 4-árny strom, ktorého maximálny počet záznamov v liste je 4 a interval časových pečiatok, ktorý pokrýva je $\langle 0, 256)$. Do stromu na obrázku 3.1 vložíme zoznam záznamov `[(4,v8),(5,v9)]`. Pre zjednodušenie na obrázkoch neuvádzame označenia verzií smerníkov, iba farebne odlišujeme vrcholy, patriace do novovytvorenej verzie stromu. Na obrázku 3.2 vidíme stav po skopírovaní a úprave listu, do ktorého na základe časových pečiatok nových záznamov patria vložené dáta. Keďže kapacita listu je 4 a už pred vložením zasiahnutý list obsahoval 4 záznamy, nastalo preplnenie a tento list bol skonvertovaný na vnútorný vrchol, pričom všetky záznamy, ktoré mu patrili sa podľa časových pečiatok rozdelili medzi jeho nových potomkov. Vidíme, že týmto rozdelením majú nové listy maximálnu hĺbku, teda nie je možné ich preplniť bez zduplikovania časových pečiatok. Obrázok 3.3 zobrazuje stav po skončení vykonávania operácie **InsertValues** s farebne odlíšeným nový stromom, na ktorého koreň je presmerovaný smerník na aktuálnu verziu stromu.

InsertValues

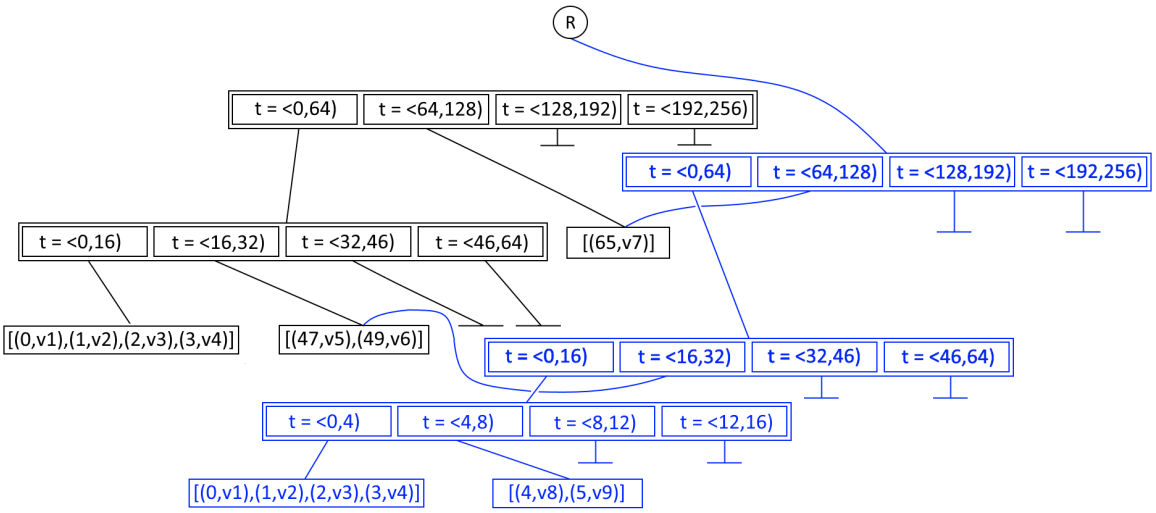
Požiadavka na vkladanie záznamov sa najskôr spracúva vo funkciách poskytujúcich rozhranie klientovi z balíku `grpcinterface/`. Urobí sa úvodná kontrola veľkosti zoznamu vkladaných záznamov a taktiež sa pomocou funkcie balíčka `rez` určí, či je dostupný dostatok prostriedkov, cluster nie je preťažený a dá sa doň písať. Ak je možné pokračovať vo vykonávaní vkladania, zavolá sa funkcia z hlavného rozhrania v balíčku `btrdb`. Tam sa skontroluje, či vstup neobsahuje nelegálne údaje a či sa vkladá do endpointu, ktorý má na starosti daný časový rad podľa aktuálnej štruktúry MASH. Ak je všetko v poriadku, záznamy z dopytu sa vložia do PQM bufferu. Ak sa PQM buffer pridaním nových dát nepreplnil, je jeho stav zaznamenaný do journalu a tým sa vykonávanie dopytu končí. Ak sa PQM buffer preplnil, je nutné úrobiť operáciu **Flush**, vytvoriť novú verziu stromu a uvoľniť body obnovenia v journale. Najskôr sa pomocou



Obr. 3.1: Ukážka čiastočne naplneného 4 árneho stromu. R obsahuje smerník na aktuálnu verziu koreňa.



Obr. 3.2: Ukážka preplnenia listu. List s intervalom $\langle 0, 16)$ bol pri vkladaní preplnený a v novej verzii skonvertovaný na vnútorný vrchol zodpovedný za ten istý interval časových pečiatok. Záznamy z tohoto intervalu boli rozdelené medzi jeho potomkov, pričom boli vytvorené dva nové listy, ktoré sa už nachádzajú pre tento konkrétny strom v maximálnej hĺbke.



Obr. 3.3: Ukážka konštrukcie novej cesty od upravených listov ku koreňu. Všetky vnútorné vrcholy na ceste od upraveného listu až po koreň sú zduplikované a predpočítané štatistiky v nich sú pre zmenené podstromy prepočítané (nie sú vyznačené na obrázku). Referencie na nezmenené vrcholy sú použité z predchádzajúcej verzie stromu. Na záver je presmerovaný smerník na koreň aktuálnej verzie. Smerník na koreň predchádzajúcej verzie je tiež uložený, aby bolo možné prechádzať aj predošlú verziu stromu.

funkcií balíkov `bstore`, `bprovider` a `cephprovider` prečíta superblok stromu, obsahujúci rozhodujúce informácie o fyzickom umiestnení záznamov stromu v úložisku. Samotný superblok obsahuje okrem iných údajov aj fyzickú adresu koreňa stromu, prípadne indikátor oznamujúci, že žiadny koreň neexistuje, čo nastáva, ak dopyt vkladá záznamy do novovytvoreného časového radu so špeciálnymi hodnotami. Samotná konštrukcia nového stromu v pamäti prebieha pomocou funkcií balíčka `qtree`. Používa sa jednoduchý rekurzívny algoritmus na prechod stromom, ktorý na základe časových pečiatok vkladaných záznamov vie identifikovať, do ktorých podstromov má ísť. Pri tomto prechádzaní sa čítajú potrebné bloky z úložiska, obsahujúce vrcholy z predchádzajúcej verzie a na základe nich sa vytvára nový strom. Tento strom je následne potrebné pretransformovať do uložiteľnej podoby. Túto úpravu opäť zabezpečujú funkcie z balíka `bstore`. Najskôr sa z úložiska získa voľná adresa, za ktorú je možné zapisovať a zapíšu sa listy stromu. Následne sa zapisuje celý strom po vrstvách v smere od najhlbších vrcholov, pričom fyzické adresy z predchádzajúcej vrstvy (počínajúc listami) sú zapísané ako hodnoty smerníkov vo vyššej vrstve. Týmto sa zabezpečuje efektivita pri čítaní vrcholov stromu z úložiska. Čítanie sa začína sa od koreňa a smerníky na potomkov sú priamo fyzickými adresami, na ktorých sa nachádzajú serializované dáta vrcholov. Vďaka tomu nie je potrebné udržiavať globálnu tabuľku s mapovaním identifikátora vrcholu na jeho fyzickú adresu. Po uložení celého stromu sa následne uloží

nový superblok pre aktuálnu verziu stromu a pridá sa do cache.

Zjednodušená ilustrácia ComputeDiff

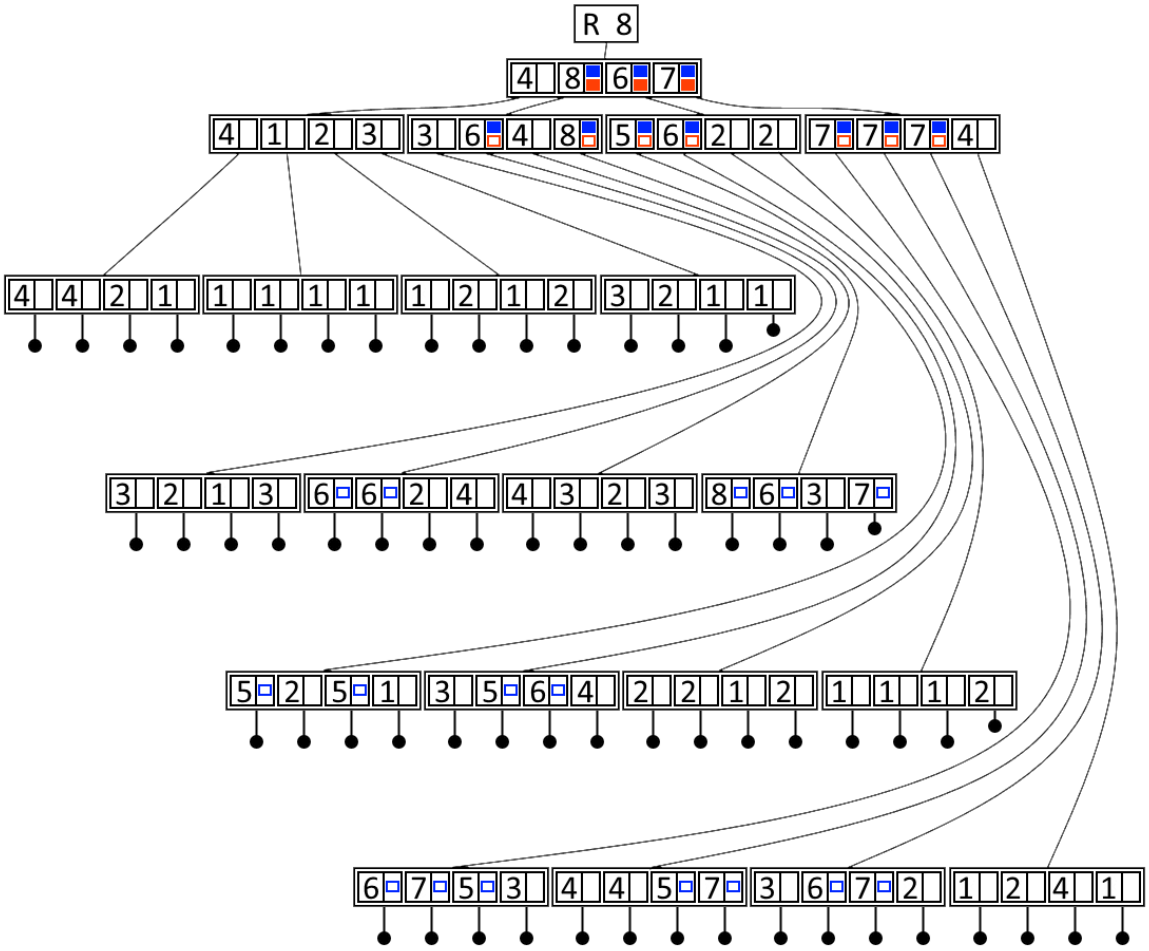
Uvažujeme 4-árny strom, ktorého maximálny počet záznamov v liste je 4 a interval, ktorý pokrýva je $\langle 0, 256)$. Na obrázku 3.4 zobrazujúcom takýto strom sú označené všetky vrcholy navštívené pri vykonávaní dopytu na zmenené intervaly medzi verziou 5 a 8 bez obmedzenia rozlíšenia a s obmedzením rozlíšenia. Výsledkom prvého dopytu je zoznam intervalov

```
[<80,88),<112,120),<124,132),<136,140),<148,156),<192,204)
   ,<216,224),<228,236)]
```

a druhého dopytu zoznam `[<80,96),<112,160),<192,240)]`.

ComputeDiff

Vykonávanie operácie **ComputeDiff** taktiež najskôr spracuje požiadavku vo funkciách z balíku
`grpcinterface` a `rez`. Následne sa zavolajú funkcie balíčku `btrdb`. Keď má systém dostatok prostriedkov a platí, že ak je **ToVersion** špeciálna hodnota, označujúca ešte nezapísanú verziu dát nachádzajúcu sa v bufferi PQM, t.j. aktuálnu verziu stromu spolu s obsahom PQM, je dopyt vykonávaný na správnom endpointe, ktorý má na starosti daný časový rad, výpočet môže pokračovať. Pomocou funkcií balíčkov `bstore`, `bprovider` a `cephprovider` sa prečíta superblok stromu a vyhľadá v ňom koreň pre verziu **ToVersion**. Potom sa pomocou funkcií v balíku `qtree` vykoná samotný prechod stromu. Postupuje sa po smerníkoch, ktorých označenie má verziu väčšiu ako ako **FromVersion**. To znamená, že v podstrome vrcholu, na ktorý ukazuje daný smerník došlo k nejakej zmene od verzie **FromVersion**. Hĺbka, kam sa až strom prechádza je určená vstupným parametrom **Resolution**. Ak nastane situácia, že potomok aktuálne prehľadávaného vrcholu pokrýva časový úsek, ktorý je menší ako požadované rozlíšenie a smerník, ktorý naň ukazuje má označenie verzie s verziou väčšou ako **FromVersion**, ďalšie rekurzívne volanie nepokračuje a celý interval, pokrytý potomkom sa vráti na vstup, ako zmenený. Vzhľadom k tomu, že z dôvodu šetrenia miesta sa v listoch neuchovávajú pečiatky verzie pre jednotlivé záznamy, nie je možné odlíšiť, aké zmeny sa v listoch udiali bez toho, aby nebol prečítaný celý list z predchádzajúcej verzie. Toto sa nedeje, keďže sa okrem stromu s verziou **ToVersion** neprechádzajú žiadne iné stromy a tak nie je možné listy z iných verzií lokalizovať. Rozlíšenie intervalov, ktoré môžu byť dané na výstup je teda obmedzené počtom záznamov v liste, ktorých je najviac 1024. Toto implementačné rozhodnutie má negatívny vplyv na presnosť dopytov, najmä ak je strom veľmi plytký. Ak sa napríklad nejaký list, do ktorého bol v skúmanom intervale identifikátorov verzií vložený jeden záznam, nachádza v hĺbke 1 a odpoveď na



Obr. 3.4: Ukážka prechodu stromu. R obsahuje smerník na aktuálnu verziu koreňa s označením verzie. Pre potreby tejto ilustrácie nie sú dôležité konkrétne hodnoty záznamov v listoch ale štruktúra stromu, ktorý má na starosti rovnaký interval časových pečiatok ako strom z obrázku 3.1. Čísla vo vnútorných vrcholoch sú označením verzie k smerníku na daného potomka. Na ilustrácii je možné vidieť ako sú využité označenia verzií na vyhnutie sa prechodu nezmenenými podstromami, ktoré určite neobsahujú žiadne intervaly z výsledného zoznamu a taktiež pri získavaní výsledkov pri obmedzení hĺbky prechodu určeným rozlíšením. Modrou farbou sú označené smerníky, prejdené pri dopyte s neobmedzeným rozlíšením a oranžovou farbou tie, ktoré sú prejdené pri dopyte s obmedzeným rozlíšením. Vyplnený obdĺžnik znamená, že algoritmus prečítal vrchol, na ktorý ukazuje daný smerník, nevyplnený znamená, že algoritmus zaregistroval interval patriaci vrcholu, na ktorý smerník ukazuje.

dopyt bude obsahovať celý interval, ktorý má na starosti, čo môže byť zahŕňať časové pečiatky z intervalu niekoľkých rokov, hoci časový úsek, ktorému zodpovedá najmenší interval časových pečiatok pokrývajúci časové pečiatky záznamov, ktoré sú v tomto liste reálne uložené, by mohol mať niekoľko nanosekúnd.

Výsledkom rekurzívneho prechádzania stromu môžu byť intervaly, ktoré je možné zlúčiť. Intervaly $\langle a, b)$ a $\langle c, d)$ sú zlúčiteľné ak $b = c$. Tieto sú zlúčené funkciami v balíčku `btrdb` pri návrate z rekurzie a posielaní údajov do gRPC rozhrania.

Kapitola 4

Úprava BTrDB

V tejto kapitole sa nachádzajú informácie o úpravách systému BTrDB, ktoré sme vykonali, aby sme dosiahli požadovanú funkcionalitu.

4.1 Implementácia variancie

BTrDB vo verzii 4 podporuje predpočítané agregácie minimum, maximum, počet záznamov v podstrome a aritmetický priemer. Agregácie, ktoré je možné implementovať sú obmedzené na také, ku výpočtu ktorých sú postačujúce medzivýsledky z podstromov a nepotrebujú prístup priamo k záznamom časového radu. Napríklad aritmetický priemer je implementovaný tak, že sa pomocou agregácií počtu prvkov a priemeru pre jednotlivé podstromy vypočíta celková suma prvkov v podstromoch a potom sa delí celkovým počtom prvkov z podstromov. Ukazuje sa, že aritmetický priemer a počet prvkov z podstromov je dostačujúci aj na výpočet výberovej variancie, ktorá je významným štatistickým ukazovateľom. Rozhodli sme sa ju teda do systému doplniť, čo nám tiež pomohlo preskúmať, nakoľko je systém rozšíriteľný. Pri výpočte vo vrchole používame párový algoritmus na výpočet výberovej variancie [7, 21] v nasledujúcej ukážke implementovaný v jazyku Go.

```
1  func Variance(mean_a float64, mean_b float64, count_a int64,
2    count_b int64, variance_a float64, variance_b float64) float64
3  {
4      delta := mean_b - mean_a
5      sum_a := variance_a * float64(count_a - 1)
6      sum_b := variance_b * float64(count_b - 1)
7      total := (count_a + count_b)
8      temp := (delta * delta) * float64(count_a * count_b) / float64(total)
9      sum := sum_a + sum_b + temp
10     return (sum / float64(total - 1))
11 }
```

Implementácia bola rozdelená do troch častí - úprava dátového modelu, úprava úložiskových a komunikačných vrstiev systému a úprava externých závislostí.

4.1.1 Úprava dátového modelu

V prvej časti sme implementovali samotný algoritmus do stromovej štruktúry. V balíčku `qtree` sme doplnili nasledujúcu funkciu:

```
1 func (n *QTreeNode) OpCountVariance(mean float64) float64
```

Táto funkcia sa používa na výpočet agregácií z podstromu keď sa podstrom novej verzie (upravený vložením nových záznamov alebo inou operáciou) podvesuje pod vrchol, ktorého novú verziu práve vytvárame. Keďže na výpočet variancie priamo zo záznamov (čo nastáva v prípade, že nový podstrom je list) používame celkový priemer, ktorý potrebujeme z vrcholu vypočítať tiež, v záujme čo najmenších zásahov do zdrojového kódu sme ho nechali vypočítať v separátnej funkcii

```
1 func (n *QTreeNode) OpCountMean() float64
```

tak ako doteraz a jeho hodnotu sme do `OpCountVariance` poslali ako argument. Bolo by tiež možné za cenu opakovaného výpočtu ho vypočítať v OpCountVariance ešte raz, alebo zlúčiť `OpCountMean` a `OpCountVariance` do jednej funkcie, ktorá by vrátila obe hodnoty. Náš prístup však zachoval oddelenosť zodpovedností jednotlivých agregačných funkcií a tým neporušil modulárny štýl počítania agregácií. V prípade, že je `OpCountVariance` volaná na koreň podstromu, ktorý je sám vnútorným vrcholom, používame párový algoritmus na vypočítanie variancie z agregácií podstromov tohoto koreňa, ktoré sú už v ňom uložené. Ďalej sme tiež upravili funkciu

```
1 func (n *QTreeNode) OpReduce(pointwidth uint8, index uint64)
2   (uint64,float64,float64,float64,float64)
```

tak, aby vracala všetky agregácie, t.j. aj varianciu. Táto funkcia sa používa pri získavaní štatistických záznamov z podstromu. Je volaná z funkcie z rovnakého balíčka

```
1 func (n *QTreeNode) QueryStatisticalValues(ctx context.Context,
2   rv chan StatRecord, err chan bte.BTE, start int64, end int64, pw uint8)
```

Táto funkcia implementuje operáciu **GetStatisticalRange** tým, že rekurzívne prechádza strom až do hĺbky určenej požadovaným rozlíšením okna, kde následne pomocou `OpReduce` počíta z predspracovaných agregácií výsledné hodnoty agregácií pre dopyt. Táto funkcia je používaná na počítanie štatistík pre zarovnané okná dĺžky mocniny 2, ktoré priamo využívajú tvar stromu.

Ďalej sme upravili už spomenutú funkciu, ktorá podvesí vrchol `c` pod vrchol `n` tak, aby brala do úvahy existenciu variancie ako novej predpočítavanej agregácie.

```
1 func (n *QTreeNode) SetChild(idx uint16, c *QTreeNode)
```

Taktiež sme upravili dátové štruktúry používané v `QueryStatisticalValues`. Upravili sme aj funkcie, ktoré sa používajú na odoslanie výsledku pre už vypočítané štatistické okno a na výpočet štatistík v rámci funkcie `QueryWindow` implementujúcej operáciu **GetStatisticalRange**. Funkcia

```
1 func (n *QTreeNode) QueryWindow(ctx context.Context,
2   end int64, nxtstart *int64, width uint64, depth uint8,
3   rv chan StatRecord, rve chan bte.BTE, wctx *WindowContext)
```

na strome rekurzívne počíta štatistické okná pokrývajúce zvolený časový interval v zvolenom ľubovoľnom rozlíšení a v záujme zrýchlenia umožňuje do výpočtu zaviesť voliteľnú nepresnosť obmedzením hĺbky prechádzania stromu. Oproti `QueryStatisticalValues` nekladie žiadne požiadavky na zarovnanie intervalov. Ďalšie nami upravené funkcie sa používajú na spájanie už vypočítaných výsledkov s hodnotami ďalších dát patriacich do okna.

```
1 func (n *QTreeNode) emitWindowContext(ctx context.Context,
2   rv chan StatRecord, width uint64, wctx *WindowContext,
3   rve chan bte.BTE)
```

```
1 func (n *QTreeNode) updateWindowWholeChild(child uint16, c *QTreeNode)
```

Opäť sme využili párový algoritmus na výpočet variancie.

Pri podrobnom skúmaní funkcie `QueryValues` sme si všimli, že obsahuje chybu, spôsobujúcu, že v určitých prípadoch sú jej výsledkom nesprávne štatistiky. Problém sa nachádzal v časti zodpovednej za spracovanie listu obsahujúceho prvé časové okno. Použitý algoritmus neošetroval okrajový prípad, v ktorom sa pred prvým existujúcim záznamom patriacim do intervalu $\langle StartTime, EndTime)$ môžu nachádzať prázdne okná. V takom prípade systém vo výsledku premiestnil prvý záznam nachádzajúci sa v intervale do prvého okna, bez ohľadu na to, že časová pečiatka tohoto záznamu do zvoleného okna vôbec nepatrila. Tento problém sme opravili tak, aby táto funkcia mohla byť volaná aj na intervaly, ktoré pri zvolenej veľkosti okna začínajú prázdnymi oknami. Naša oprava bola prijatá aj do oficiálneho repozitára BTrDB projektu.

4.1.2 Úprava úložiskových a komunikačných vrstiev

V druhej časti sme zabezpečili, aby sa počítaná agregácia ukladala do perzistentného úložiska a aby bola prístupná klientom. V prvom rade to znamenalo prerobiť nasledujúce funkcie zo súboru `merger.go`

```
1 func CreateStatWindows(rz []qtree.Record, tCutoffStart int64,
2   tstart int64, tEnd int64, w uint64)
```

```
1 func mergeStatisticalWindowChannels(parent chan qtree.StatRecord,
2   parentCE chan bte.BTE, pqbuffer []qtree.StatRecord)
3   (chan qtree.StatRecord, chan bte.BTE)
```

Tieto funkcie sa používajú na výpočet odpovedí dopytov používajúcich operáciu **GetStatisticalRange** bez ohľadu na zarovnanie počítaných okien. Prvá z týchto dvoch funkcií vypočíta agregácie po skupinách na základe rozdelenia okien z vykonávaného dopytu zo záznamov, ktoré sa nachádzajú v pamäti PQM a ešte neboli zapísané do stromu v hlavnom úložisku. Druhá funkcia zlučuje agregáty získané z prvej funkcie s hodnotami vypočítanými z dát uložených v strome. Opäť sme na implementovanie výpočtu variancie použili párový algoritmus. Výpočty agregácií predpokladajú, že množina časových pečiatok záznamov, nachádzajúcich sa v PQM bufferi je disjunktná s definičným oborom aktuálnej verzie časového radu, zapísaného v strome, čo je vďaka obmedzeniu na prácu s časovými radmi so špeciálnymi hodnotami a oddeľovačmi zabezpečené.

Pri skúmaní tejto časti systému sme zistili, že zlučovanie štatistík zo stromu s obsahom PQM neberie do úvahy verziu stromu, z ktorého dopyt číta. V dopytoch je totiž možné určiť buď konkrétnu verziu, z ktorej sa majú štatistiky vypočítať, alebo špeciálnou hodnotou identifikátoru verzie signalizovať, že výsledok má obsahovať štatistiky z aktuálnej verzie uloženej v strome, zlúčené so štatistikami, vypočítanými z obsahu PQM bufferu. Hodnoty z PQM sa však do výsledku zlučovali bez ohľadu na to, či sa čítala aktuálna verzia stromu alebo historická. V konečnom dôsledku to znamenalo, že ak dopyt vykonával endpoint, ktorý mal vtedy na starosti daný časový rad, garancia správnych výsledkov bola zabezpečená iba pre aktuálnu verziu. Pre všetky ostatné verzie nemuseli byť vypočítané agregácie presné, v závislosti od aktuálneho obsahu PQM pre dopytovaný časový rad s špeciálnymi hodnotami. Na druhej strane, ak dopyt vykonával niektorý z iných endpointov, ktoré, keďže nemali daný časový rad na starosti, nepodporovali čítanie aktuálnej verzie, ich výsledky boli garantovane správne, pretože ich PQM buffer pre daný časový rad musel byť prázdny. Upravili sme teda funkcie používajúce PQM tak, aby jeho obsah zlučovali iba s výsledkami zo stromu pre dopyt s uvedenou špeciálnou hodnotou. Dosiahli sme tým, že pri dopyte na čítanie historickej verzie na ľubovoľný endpoint dostane klient garantovane správne výsledky. Táto naša úprava bola tiež prijatá do oficiálneho repozitára projektu.

Ďalej, na ceste smerom ku klientovi bolo potrebné upraviť štruktúry v gRPC interface, aby brali do úvahy prítomnosť nového agregátu a taktiež aj metódy v balíčku `grpcinterface`, ktorý je medzičlánkom medzi gRPC interface a operáciami na uložených dátach.

Smerom k perzistentnému úložisku sme museli upraviť všetky dátové štruktúry, používané na reprezentáciu stromov a doplniť pomocné metódy na klonovanie, serializovanie či deserializovanie z balíčku `bstore` tak, aby ukladali aj predpočítanú varianciu,

nachádzajúcu sa v štruktúrach v pamäti.

4.1.3 Úprava externých závislostí

V tretej časti sme upravovali http server tak, aby odosielal počítanú a uloženú hodnotu variancie spolu s ostatnými agregáciami. Spočívalo to v pridaní variancie do dopytovacieho jazyka, schémy gRPC a schém nástroja Swagger, ktorý SmartGridStore používa ako frontend a taktiež do BTrDB klienta. BTrDB klient je napísaný v jazyku Go a obsahuje aj testy interagujúce so systémom. Niektoré z týchto testov sme upravili a pridali sme kontrolu správne počítanej variancie. Doplnili sme aj metódy na export CSV súborov v balíku `grpcinterface` tak, aby generovaný CSV súbor obsahoval stĺpce s vypočítanou varianciou. Pre potreby interaktívneho testovania sme upravili aj knižnicu pre jazyk Python. Táto úprava opäť spočívala v prerobení gRPC interface a umožnila nám vykonávať dopyty na BTrDB z interaktívneho prostredia. V tejto knižnici sme tiež opravili chybu, ktorá spôsobovala, že nebolo možné vykonať dopyt s funkciou `ComputeDiff`. Oprava spočívala v úprave funkcie, použitej na výpis výstupu z dopytu. Taktiež sme v tejto knižnici opravili metódy používané na vytváranie CSV súborov, ktoré tiež obsahovali niekoľko chýb.

4.2 Skúmanie upravovania historických verzií

Ako bolo ukázané v predchádzajúcej časti, systém pracuje s časovými radmi so špeciálnymi hodnotami a oddeľovačmi, nepodporuje teda upravovanie údajov, ale iba pridávanie záznamov pre nové časové pečiatky a ich odstraňovanie. Tento dizajn bol pravdepodobne zvolený kvôli cieľovému typu dát - údajom z PMU ktoré sa pravdepodobne v bežnej prevádzke neupravujú. Systém preto pri svojej činnosti nepočíta s alternatívou, že by sa v PQM bufferi mohli nachádzať záznamy s identickou časovou pečiatkou, prípadne že by dopyt na vkladanie mohol obsahovať aj záznam s časovou pečiatkou, ku ktorej už systém záznam uložený má a aktuálny záznam pre túto časovú pečiatku nemá hodnotu Nil. Aby sme rozšírili funkcionalitu systému tak, aby vedel spracúvať časové rady so špeciálnymi hodnotami bez obmedzujúcich oddeľovačov, rozhodli sme sa implementovať možnosť upravovať existujúce záznamy.

Vzhľadom na charakteristiku časových radov so špeciálnymi hodnotami a oddeľovačmi je možné úpravu záznamu vykonať tým, že záznam najskôr odstránime a potom ho nahradíme novým. V BTrDB by to zabezpečovala nasledujúca sekvencia operácií, kde $UUID$ je identifikátor časového radu, u nový údaj a c časová pečiatka záznamu, ktorý chceme upraviť.

```
1 DeleteRange(UUID, c, c+1)
2 InsertValues(UUID, [(c, u)])
3 Flush(UUID)
```

Je nutné uviesť, že posledné volanie operácie **Flush** nemusí byť nevyhnutné, ak nechceme úpravu okamžite uložiť ako novú verziu. Keďže však samotná funkcia, vykonávajúca operáciu **DeleteRange** spúšťa **Flush** aj na svojom začiatku aj na konci, a teda môže vytvoriť až 2 verzie v prípade, že PQM pri jej volaní obsahovalo nejaké záznamy z predchádzajúcich vložení, môže jedna sekvencia operácií, reprezentujúca operáciu *Update* vytvoriť až 3 verzie. Jedna z nich, vytvorená po operácii **DeleteRange** je redundantná, keďže zmazanie nastalo len kvôli potrebe úpravy záznamu a samé o sebe nemalo z pohľadu používateľa žiaden význam. Je zjavné, že takýto spôsob upravovania záznamov je nepraktický a neefektívny. Taktiež, nie je vylúčené, že medzi operáciou **DeleteRange** a poslednou operáciou **Flush** nenastane konkurentná operácia **InsertValues**, ktorá upravené záznamy zmieša s novými aj v prípade, že klient ich chcel uložiť ako osobitnú verziu. Aby sme umožnili priamočiare upravovanie záznamov, rozhodli sme sa upraviť procedúry používané na bežné vkladanie tak, aby túto novú funkcionalitu umožnili, ale aby sme čo najmenej menili rozhranie pre používateľov.

V pôvodnej implementácii BTrDB kontroluje duplikované časové pečiatky v rekurzívnej funkcii z balíčku `qtree`

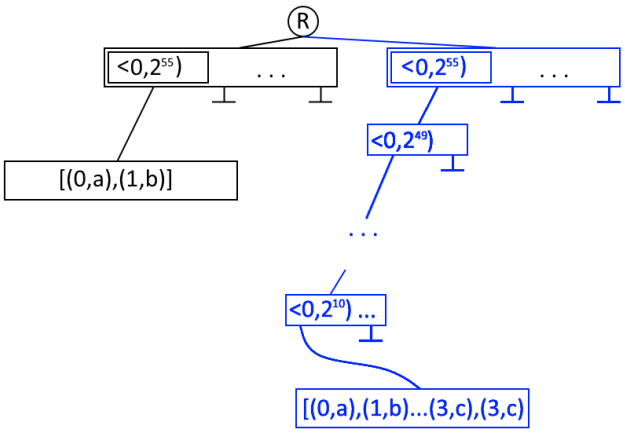
```
1 func (n *QTreeNode) InsertValues(records []Record) (*QTreeNode, error)
```

ktorá vkladá záznamy do jednotlivých vrcholov stromu. Jej argument `records` obsahuje vkladané záznamy, ktorých časová pečiatka patrí do intervalu, ktorý má na starosti vrchol, na ktorom je funkcia práve volaná. Ak by systém dôsledne vyžadoval používanie časových radov s špeciálnymi hodnotami a oddeľovačmi, táto kontrola by mala ošetriť chyby dvoch typov. V prvom rade by mala detegovať prichádzajúce duplikáty, t.j. prípad, keď pole záznamov `records` obsahuje dva záznamy s rovnakou časovou pečiatkou. V druhom rade by mala táto kontrola detegovať situáciu, keď pole záznamov `records` obsahuje záznam s takou časovou pečiatkou, ktorú obsahuje aj nejaký záznam už nachádzajúci sa v strome. Kontrola však prebieha až v prípade, keď sú nové záznamy vkladané do listu, ktorý sa nachádza v maximálnej hĺbke a funkcia kontroluje, či dĺžka poľa `records` sčítaná s počtom záznamov v liste je menšia alebo rovná maximálnej kapacite listu. Ak nie je, systém vyhlási chybu a preruší vkladanie tým, že vloží iba toľko záznamov, čo sa zmestí do daného listu, pričom neposkytuje žiadne garancie týkajúce sa časových pečiatok záznamov z poľa `records`, ktoré skutočne zapíše. Takáto kontrola však zachytáva duplikáty časových pečiatok nedostatočným spôsobom. Chyba totiž môže byť vyhlásená aj vtedy keď vkladané záznamy neobsahovali duplikáty a ani neprepisovali už existujúce záznamy a teda boli z hľadiska požiadaviek systému korektné. Na druhej strane, pri vkladaní záznamov s duplikátmi, či záznamov,

ktoré prepisujú existujúce záznamy sa chyba nemusí vyvolať a hoci nastáva z hľadiska časových radov so špeciálnymi hodnotami poškodenie obsahu dátových štruktúr a strata presnosti predpočítaných agregácií, kontrola tento stav neodhalí a používateľ o tom nevie. Tieto nedostatky sa nám podarilo experimentálne overiť, uvádzame preto zjednodušené príklady, ktoré popisujú scenáre, ktoré vyvolajú uvedené správanie.

V prvom rade platí, že pokiaľ ukladaný časový rad nemá primalú periódu, respektíve zatiaľ obsahuje málo záznamov, je málo pravdepodobné, že strom dosahuje maximálnu hĺbku. Pri praktickom použití je tiež možné, že perióda ukladaného časového radu spôsobí, že aktuálne existujúce listy (bez ohľadu na ich hĺbku) nie sú úplne zaplnené. Pri vložení záznamu s duplikovanou časovou pečiatkou sa preto môže zapísať chybný záznam bez toho, aby sa list preplnil alebo toho, aby sa dostal až na najnižšiu úroveň stromu a teda žiadna výnimka nie je vyvolaná, hoci predpočítané agregácia už neobsahujú očakávané hodnoty. Následne sa môže stať, že po sérii vložení korektných záznamov sa list preplní, prepadne sa na najnižšiu úroveň a nastane chyba, pri ktorej sa nezapíšu korektné dáta bez duplikovaných časových pečiatok, hoci tie chybné v systéme ostanú. Takéto tiché chyby sa nám podarilo v systéme vyvolať. Naviac, ak boli existujúce záznamy dostatočne nahustené v malom intervale časových pečiatok, pri ich vkladaní sa mohlo stať, že kvôli niekoľkým duplikátom sa jedna vetva stromu neúmerne predĺžila a vzniklo veľa zbytočných vnútorných vrcholov, ktoré sú už kvôli prítomnosti správnych dát ťažko odstrániteľné. Príkladom je strom hĺbky 1, ktorý má koreň a jeden list, obsahujúci záznamy s časovými pečiatkami 0 a 1. Následne je chybne pridaných 1023 duplikátov záznamu s časovou pečiatkou 3. Pri vkladaní sa list preplní, skonvertuje sa na vnútorný vrchol a jeho obsah sa presunie do listu o úroveň nižšie, no keďže všetky existujúce aj vkladané záznamy patria do jediného z najmenších intervalov stromu $\langle 0, 1024)$, prepĺňanie pokračuje až kým sa nedosiahne najhlbší možný list, kde sa vyhlási chyba a zapíše sa 1022 duplikátov. Ilustráciu tohoto správania môžeme vidieť na 4.1. Na túto chybu môže operátor reagovať vymazaním intervalu $\langle 3, 4)$, čím sa odstráni všetkých 1022 duplikátov, no neskráti sa cesta vedúca do listu, keďže obsahuje aj platné záznamy a žiaden z jej vrcholov nespravuje podinterval intervalu $<3, 4)$. Pre jej skrátenie je potrebné vymazať aj správne záznamy a vložiť ich nanovo.

Systém sme najskôr upravili tak, aby detegoval vkladané záznamy s duplikovanými časovými pečiatkami a v prípade, že nejaké operácie obsahovali takéto duplikáty, zahadzoval všetky záznamy z týchto operácií. Zabezpečili sme tým, aby systém pracoval s časovými radmi so špeciálnymi hodnotami a oddeľovačmi a túto vlastnosť vkladaných dát dôsledne kontroloval. Alternatívou k tomuto prístupu bolo z operácií obsahujúcich duplikáty odstrániť iba chybné záznamy a zvyšné vložiť, čo by tiež zabezpečilo prácu s časovými radmi so špeciálnymi hodnotami a oddeľovačmi. Motivácia k nášmu prístupu, ktorý zahadzoval viac vkladaných záznamov bola, že v prípade, že nejaká operácia obsahuje chybný záznam, je vhodné predpokladať, že aj ostatné záznamy,



Obr. 4.1: Ukážka chybne zväčšeného stromu pri vložení duplikátov. Pre jednoduchosť uvažujeme strom s posunutým intervalom časových pečiatok $\langle 0, 2^{61})$. Farebne je odlíšená nová verzia stromu, vytvorená vložením 1023 duplikátov záznamu $(3, c)$. Výsledný strom má, podobne ako pôvodný, iba jeden list, tentokrát však nachádzajúci sa v maximálnej hĺbke.

ktoré sa síce môžu zdať korektné, v zmysle, že nekolidujú s časovými pečiatkami existujúcich záznamov, v skutočnosti môžu byť chybné. Zmeny, ktoré sme vykonali boli do veľkej miery podobné zmenám, ktoré popisujeme v nasledujúcej časti, venujúcej sa pridaniu podpory historických verzií, keďže sme na detekciu duplikátov použili totožný mechanizmus, používajúci množiny časových pečiatok. Okrem týchto zmien sme museli upraviť aj funkciu

```
1 func (pqm *PQM) flushLockHeld(ctx context.Context, id uuid.UUID,
2   st *streamEntry)
3   (maj uint64, min uint64, err bte.BTE)
```

ktorá je používaná všetkými zapisovacími operáciami na vyprázdnenie PQM bufferu a zápis do stromu. Ak zápis do stromu zlyhá kvôli prítomnosti duplikátov, je nutné špeciálne ošetriť tento prípad v porovnaní s inými chybami zápisu, keďže v takomto prípade je isté, že zápis sa ani pri opakovaní nepodarí. Je preto nutné zahodiť zapisované záznamy, uvoľniť záznamy v journale a odomknúť zamknutý strom, keďže odomykanie sa v bežnej prevádzke vykonáva až pri dokončení zápisu do úložiska Ceph.

Ani jeden z týchto prístupov však nedokáže ošetriť situáciu, keď niektorý z vkladaných záznamov, ktoré sa práve nachádzajú v PQM bufferi, má rovnakú časovú pečiatku ako už existujúci záznam v strome. Keďže PQM buffer slúži práve na to, aby nebolo nutné pri každej jednej operácii vkladania čítať surové záznamy počas konštruovania stromu, detegovať takto duplikované časové pečiatky je možné až keď je strom so surovými dátami skutočne prečítaný do pamäte, čo nastáva až pri vyprázdňovaní PQM bufferu. Ak teda na upravovaný časový rad prichádzajú od klienta agregujúce dopyty

na aktuálnu verziu medzi operáciou vloženia duplikátu a medzi operáciou, ktorá vyprázdni buffer, výsledky týchto agregácií nie sú presné, keďže pre duplikované časové pečiatky boli niektoré hodnoty zarátané viac krát. Takúto chybu nie je možné detegovať, keďže agregácie čítané zo stromu nenesú so sebou informáciu o časových pečiatkach záznamov, z ktorých boli vypočítané.

Kontaktovali sme vývojárov systému a oboznámili sme ich s našou úpravou. Ukázalo sa, že k problému záznamov z duplikovanými časovými pečiatkami pristupujú iným spôsobom, ktorý jemne mení pohľad na význam pojmu časového raduv rámci ich systému. Uvedená kontrola je údajne zámerne voľná, pretože v podmienkach štandardnej prevádzky BTrDB môže nastať situácia, že PMU odošle do systému záznamy s nesprávnymi časovými pečiatkami, ktoré sú napríklad posunuté do minulosti a kolidujú s už zapísanými časovými pečiatkami. Toto posunutie hodín PMU môže spôsobiť napríklad zmena GPS satelitov, viditeľných týmto zariadením. V takejto situácii sa systém zámerne snaží zapísať čo najviac dát, vrátane všetkých duplikátov až neskôr ich oddeliť a dáta tak vyčistiť pomocou výpočtov v nástroji DISTIL. Hoci teda v systéme v takomto prípade existuje duplikát, z pohľadu klienta sa stále jedná o dve separátne merania, z ktorých jedno má nesprávne priradenú časovú pečiatku. Na výsledky agregácií sa pri takomto pohľade dá pozerať ako na správne, keďže klient očakáva, že vo výsledkoch sa prejavia skutočne údaje všetkých meraní, aj tých, čo majú nesprávne priradenú časovú pečiatku.

Uvedená kontrola vo funkcii `InsertValues` teda má na starosti iba zabrániť preplneniu listov, nie zabrániť zápisu duplikátov. Pracuje týmto spôsobom aj za cenu, že môže spôsobiť, že záznamy, ktoré sú zahodené v skutočnosti majú správne časové pečiatky a hrozbu preplnenia listu nespôsobili. Prítomnosť tejto kontroly tiež limituje počet možných záznamov s duplikovanými časovými pečiatkami, ktorý závisí aj od periódy uchovávaného časového radu, či rozdielu jednotlivých duplikovaných časových pečiatok, čiže nie je možné povedať, že by systém počítal s ľubovoľným množstvom duplikátov.

Aj pri takto zvoľnenom pohľade je teda prítomnosť duplikovaných časových pečiatok považovaná za anomálny jav, chybu prístroja, ktorý záznamy vytvára a správanie systému v takomto okrajovom prípade nie je explicitne uvedené v žiadnej dokumentácii ani v existujúcej publikácii o systéme BTrDB. Javí sa nám teda vhodné uvažovať o systéme BTrDB ako o systéme na uchovávanie časových radov so špeciálnymi hodnotami a oddeľovačmi a počítať pri výpočtoch agregácií s garanciami, ktoré nám toto obmedzenie spracúvaných časových radov prináša. Reakciu systému na záznamy duplikovanými môžeme považovať za nedefinované správanie, miesto explicitnej detekcie a ošetrovania prítomných duplikátov, pričom je zodpovednosťou klienta, aby vzniku takejto situácie zabránil.

4.2.1 Implementácia upravovania historických verzií

Pre zabezpečenie správneho upravovania historických verzií sme sa rozhodli upraviť existujúci spôsob vkladania záznamov a tiež pridať novú funkciu do aplikačného rozhrania, pričom sme sa snažili zachovať efektivitu vkladania pomocou PQM a voľné podmienky pre používateľa.

Existujúci spôsob vkladania záznamov sme upravili tak, že sme miesto kontroly v listoch popísanej v predchádzajúcej časti použili kontrolu v PQM a v rozhraní zo súboru `quasar.go`. Do dátovej štruktúry `streamEntry`, ktorá sa používa na evidenciu časových radov, ktorých záznamy sa aktuálne nachádzajú v PQM, sme pridali množinu časových pečiatok implementovanú pomocou hashovacej tabuľky. V tejto množine uchovávame časové pečiatky záznamov konkrétneho časového radu, aktuálne nachádzajúcich sa v PQM bufferi. Pri vkladaní nových záznamov v rámci funkcie

```
1 func (pqm *PQM) Insert(ctx context.Context, id uuid.UUID, r []Record)
2   (major, minor uint64, err bte.BTE)
```

z balíčku `btrdb` sa pomocou uvedenej množiny skontroluje, či neexistuje vkladaný záznam s rovnakou časovou pečiatkou ako má nejaký iný záznam, nachádzajúci sa už v PQM bufferi. Ak sa tam taký nachádza, PQM buffer pre tento časový rad sa najskôr vyprázdni, vytvorí sa nová verzia stromu a potom sa prichádzajúce záznamy zapíšu do vyprázdneného bufferu. Tento mechanizmus zabezpečuje, že pre každý časový rad v systéme platí, že jeho PQM buffer v každom čase obsahuje pre ľubovoľnú časovú pečiatku najviac jeden záznam, a taktiež ľubovoľný zápis do stromu obsahuje pre každú časovú pečiatku najviac jeden záznam.

Podobnú kontrolu sme zaviedli aj do funkcie

```
1 func (q *Quasar) InsertValues(ctx context.Context, id uuid.UUID,
2    r []qtree.Record)
3   (maj, min uint64, err bte.BTE)
```

ktorá predošlú funkciu volá. Kontrola v tejto funkcii zabezpečuje, že zoznam pridávaných záznamov `r` neobsahuje duplikované časové pečiatky.

Táto garancia pre zápisy nám umožnila v balíčku `qtree` upraviť funkcie upravujúce strom. Pridali sme funkciu

```
1 func updateRecords(orig []Record, updat []Record) []Record
```

ktorá zlučuje dva zoznamy záznamov, usporiadané podľa časových pečiatok tak, že ak pre nejakú časovú pečiatku c existuje záznam v oboch zoznamoch, zachová sa iba záznam zo zoznamu `updat`. Túto funkciu sme použili v rekurzívnej funkcii

```
1 func (n *QTreeNode) InsertValues(records []Record) (*QTreeNode, error)
```

v ktorej sme odstránili v predchádzajúcej časti popisovanú kontolu duplikovaných časových pečiatok. Taktiež sme upravili funkciu

```
1 func (n *QTreeNode) MergeIntoVector(r []Record)
```

ktorá je volaná z `InsertValues` keď sa pri rekurzívnom prehľadávaní lokalizuje list, do ktorého treba vložiť záznamy a funkciu

```
1 func (n *QTreeNode) ConvertToCore(newvals []Record) *QTreeNode
```

ošetrujúcu prípad preplneného listu.

Funkcia `MergeIntoVector` je zodpovedná za správne vkladanie záznamov a ošetrovanie prípadu, keď sa list preplní. Vďaka možnosti prepísania pôvodného záznamu novým a garancii, že zoznam zapisovaných záznamov neobsahuje duplikované časové pečiatky je zabezpečené, že táto funkcia funguje správne, t.j. list delí iba v prípade, že je to skutočne potrebné. Nemôže teda nastať problematická situácia so zbytočným rozdelením, spôsobená predchádzajúcim nedostatočným ošetrením duplikovaných časových pečiatok.

Funkcia `updateRecords` má lineárnu časovú zložitosť, čiže jej pridanie nezhoršilo zložitosť operácie **InsertValues** oproti predchádzajúcej implementácii. Práve naopak, vo funkcii `ConvertToCore` sa pred našim zásahom zlučovali staré a nové záznamy v zložitosti $O((m + n') \log(m + n'))$, kde m je aktuálny počet záznamov v liste a n' počet záznamov, ktoré sa doň vkladajú. Časovú zložitosť tejto funkcie sme teda zlepšili na $O(m + n')$. Ukazuje sa však, že ešte pred volaním rekurzívnej funkcie `InsertValues` sa triedi celé pole **records** v čase $O(n \log n)$ a počet volaní funkcie `ConvertToCore` je zhora ohraničený počtom listov stromu existujúcich pred vkladaním, čiže výsledné ušetrenie nie je až také signifikantné. Z implementačných dôvodov je n obmedzené konštantou, z teoretického hľadiska sme však zlepšili časovú zložitosť funkcie `ConvertToCore`.

Popísané úpravy umožnili používať operáciu **InsertValues** aj na úpravu existujúcich údajov. Z teoretického hľadiska sme teda dosiahli funkcionalitu časového radu s historickými verziami. Keďže však používateľ stále nemá kontrolu nad zapisovaním do stromu, aj napriek používateľskému vykonaniu dopytu s operáciou **Flush** môžu vznikať verzie, obsahujúce nové aj upravené záznamy. Rozhodli sme sa preto doimplementovať funkciu **Patch**, ktorá obnovuje funkcionalitu operácie **InsertValues** z BTrDB verzie 3, t.j garantuje vytvorenie novej verzie, obsahujúcej práve záznamy, ktoré boli pridané jej volaním. V súčasnom stave systému to znamená, že dopyt vykonávajúci operáciu **Patch** pri zápise obchádza PQM buffer a zabezpečuje, že záznamy s ktorými je volaný vytvoria novú verziu stromu, pričom pre záznamy, ktoré sa počas volania nachádzali v PQM zaručuje, že sa budú nachádzať v predchádzajúcej vytvorenej verzii. Dopyt vykonávajúci operáciu **Patch** môže teda vytvoriť až 2 verzie stromu pri jednom volaní.

Operácia **Patch** používa na úpravu stromov a písanie do úložiska Ceph rovnaký mechanizmus ako operácia **InsertValues**, zmeny s ňou súvisiace sme teda implementovali v balíčkoch `grpcinterface` a `btrdb`, kde sme pridali funkciu obsluhujúcu

nové gRPC volanie spolu s jeho následným spracovaním a ošetrením chýb v nových funkciách v súboroch `quasar.go` a `pqm.go` balíčku `btrdb`.

Operácia **Patch** môže rovnako ako upravená funkcia **InsertValues** vkladať nové aj upravovať existujúce záznamy aj v rámci jedného volania. Bolo by možné striktne ich funkcionalitu oddeliť, prípadne kontrolovať, že v prípade použitia operácie **Patch** sa bude robiť iba jedna forma vkladania, považovali sme to však vzhľadom na voľné rozhranie verzie 4 zabezpečené pomocou PQM ako zbytočne reštriktívne pre používateľa. Prípadné problémy spojené so zlučovaním viacerých úprav existujúcich záznamov sme vyriešili pridaním kontroly s automatickým spúšťaním operácie **Flush** v PQM.

4.2.2 Úprava externých závislostí

Aby bolo možné používať novú operáciu **Patch**, bolo rovnako ako pri pridaní variancie nutné upraviť gRPC schému, schému pre nástroj Swagger a taktiež klientské knižnice v jazyku Go a Python. Keďže v bežnom nasadení BTrDB v rámci systému SmartGridStore je spustených viacero BTrDB endpointov, je nutné zabezpečiť, aby dopyty od klientov smerovali do správnych endpointov, zodpovedných za časové rady z daných dopytov. Toto smerovanie a ďalšiu funkcionalitu poskytuje nástroj `apifrontend`. Aby bolo možné používať novú operáciu **Patch**, bolo potrebné do tohoto nástroja pridať funkciu, ktorá gRPC volanie funkcie implementujúcej túto operáciu prijíma a presmeruje na správny endpoint.

4.2.3 Vplyv úprav v praxi

Rozhodli sme sa porovnať rýchlosť vkladania záznamov do pôvodnej a nami upravenej verzie systému. Na tento účel sme naprogramovali nástroj v jazyku Go, ktorý nepoužíval dostupnú knižnicu pre BTrDB, ale interagoval priamo s gRPC rozhraním. Naviac, nástroj sme použili bez medzičlánku tvoreného nástrojom `apifrontend` a tak komunikoval priamo serverom, na ktorom bol spustený systém BTrDB. Náš nástroj tiež umožňoval zvoliť počet zapisovaných záznamov, ich vzájomné poradie a periodicitu a taktiež aj počet časových radov do ktorých sa zapisuje spolu s počtom spojení, ktoré sa vytvárajú so systémom. Existuje aj oficiálny nástroj s podobnou funkcionalitou `QuasarLoadGenerator`, no tento je už dlhšiu dobu nepodporovaný a teda nekompatibilný s verziou 4 BTrDB. Pozorovanie však ukázalo, že aj pri priamej komunikácii s BTrDB je réžia spojená s prevádzkou vývojového prostredia prevyšujúca prácu samotného systému a nebolo možné pozorovať žiadnu zmenu rýchlosti pri spustení pôvodnej ani upravenej verzie. Naše pozorovanie nasvedčuje informácii z manuálu pre systém SmartGridStore, že v porovnaní s nasadením v serverovom clusteri v ktorom majú jednotlivé servery špecializovanú hardvérovú konfiguráciu, je používanie vývojového

prostredia na reálnu prevádzku úplne nevhodné a teda nie je použiteľné ani na získavanie relevantných výkonnostných porovnaní.

Kapitola 5

Vplyv zmien na predpočítané agregácie

Umožnením používania operácie **InsertValues** na vykonávanie úprav existujúcich záznamov sme síce zachovali efektívnosť zabezpečenú zlučovaním jednotlivých prichádzajúcich zápisov pomocou PQM, na druhej strane sme stratili garancie na presnosť agregovaných štatistík. Totiž, vzhľadom k tomu, že pri agregáciách sa BTrDB snaží nečítať všetky surové záznamy s časovými pečiatkami z intervalu daného dopytom, ale použiť už predpočítané agregácie pre jednotlivé okná, v súčasnom stave nie je možné pri zlučovaní s obsahom PQM identifikovať, či záznamy z PQM prepisujú hodnoty vypočítaných agregácií zo stromu. Napríklad, nech f je aktuálna verzia časového radu f', ktorý je uložený v strome a pre údaj zo záznamu $(c, f(c))$ platí, že pre všetky c' z definičného oboru f také, že $c' \neq c$ platí $f(c) > f(c')$. Zjavne, pri dopyte na maximum(maximálny údaj) z intervalu, pokrývajúceho celý definičný obor f, dostaneme údaj $f(c)$. Uvažujme ale, že v PQM bufferi sa nachádza záznam (c, u), čakajúci na vloženie do stromu, pričom $u < f(c)$ a pre nejaké c' z definičného oboru f platí $f(c') > u$. Striktne vzaté, z pohľadu klienta, ktorý do úvahy berie aktuálnu verziu časového radu, ktorý do systému uložil a nie aktuálnu verziu časového radu, ktorú systém stihol zapísať do stromu je už údaj zo záznamu s časovou pečiatkou c v strome prepísaný a tak by sa nemal objavovať vo výsledkoch agregácie. Systém však pri zlučovaní s PQM bufferom nemá ako zistiť, ku ktorej časovej pečiatke patril maximálny údaj vypočítaný zo stromu a teda nevie, či ho má zahodiť a nebrať do úvahy alebo iba porovnať s maximom z PQM. Ak by aj systém poznal časovú pečiatku záznamu z ktorého maximálny údaj zo stromu pochádzal a vedel by, že po zapísaní záznamov z PQM bufferu by bol tento záznam prepísaný, opäť by nemal možnosť zistiť, aká je skutočná hodnota výsledku dopytu, pretože pre strom už nemá žiadne ďalšie dostupné štatistiky. V nasledujúcej časti skúmame, aký postup je možné zvoliť, aby sa zvýšili garancie na správne výsledky niektorých zo štatistík, konkrétne minimálneho a maximálneho údaju, a zároveň aby

nebolo potrebné čítať surové záznamy, t.j. vykonávať toľko práce, čo by vyžadovala operácia **Flush** celého stromu.

5.1 Zložitosť operácie Flush

Pre jednoduchosť uvažujeme v nasledujúcich častiach zjednodušený model, t.j. strom v ktorom sú uchované záznamy aktuálnej verzie časového radu je úplný binárny, jeho hĺbku označujeme h a maximálnu kapacitu PQM bufferu označujeme m. Počet prvkov v strome je n, čiže $h = \log n$. Platí, že n je rádovo väčšie ako m, čo je zas rádovo väčšie ako $VSIZE$, čím označujeme kapacitu listu. Zároveň, kvôli zjednodušeniu uvádzaných algoritmov predpokladáme, že všetky údaje, nachádzajúce sa v strome a bufferi sú navzájom rôzne. Tým sa vyhneme situácii, v ktorej by boli údaje nejakých dvoch rôznych porovnávaných záznamov rovnaké. Taktiež, podobne ako pri reálnom systéme platí, že časové pečiatky všetkých záznamov v strome sú navzájom rôzne a časové pečiatky všetkých záznamov v bufferi sú navzájom rôzne. Pre potreby analýzy časových zložitostí algoritmov predpokladáme, že buffer je naplnený, t.j. počet záznamov, ktoré sa v ňom nachádzajú je práve m. Ďalej predpokladáme, že prichádzajúci dopyt je na interval časových pečiatok, ktorý pokrýva interval celého stromu a že tento interval spadá do jedného okna, pre ktoré sa počítajú agregácie. To znamená, že v rámci dopytu vykonávame hodnotovú agregáciu cez celý definičný obor časového radu uloženého v dátovej štruktúre stromu.

Pri operácii **Flush** sa obsah celého PQM bufferu vloží do stromovej štruktúry, pričom systém nemá žiadne predpoklady na časové pečiatky ani údaje záznamov, ktoré z neho pochádzajú. Tieto záznamy môžu prepisovať už existujúce záznamy k určitým časovým pečiatkam, alebo môžu pridávať záznamy s novými časovými pečiatkami. Podľa postupu práce algoritmu vykonávajúceho operáciu **Flush** popísaného v kapitole 3, analyzujeme časovú zložitosť tohoto algoritmu. Utriedenie obsahu bufferu podľa časových pečiatok má zložitosť $m \log m$. Každý list v ktorého intervale časových pečiatok nastáva vložením nejaká zmena je nutné prekopírovať, čo má v najhoršom prípade, ak každý vkladaný záznam patrí do iného listu, časovú zložitosť $O(m * VSIZE)$. Na každej úrovni stromu musí každý vnútorný vrchol, do ktorého podstromu sa vkladá aspoň jeden nový záznam, rozdeliť záznamy, čo sú doň vkladané, do intervalov časových pečiatok, za ktoré sú zodpovední jeho jednotliví potomkovia. To znamená, že na každej úrovni sa spracuje obsah celého bufferu, čo dáva časovú zložitosť $O(h * m)$. Zároveň sa pri vkladaní musí prečítať dokopy r vrcholov. Zostáva teda vyčísliť r v najhoršom možnom prípade. Zjavne, najhoršie rozdelenie záznamov je také, že v každej úrovni stromu spôsobí prečítanie čo najväčšieho počtu vrcholov. Keďže prečítaných listov je práve m, pri najhoršom rozdelení sa v každej úrovni, v ktorej sa nachádza viac ako m

vrcholov prečíta práve m vrcholov a v zvyšných úrovniach, ktoré sú blízko koreňu sa prečítajú všetky vrcholy. Platí teda, že $r = m * (h - \log m) + 2m - 1$

Časová zložitosť operácie **Flush** je teda $O(m\log m + m * VSIZE + h * m + m * (h - \log m) + 2m - 1)$ a zložitosť v zmysle prečítaných a zapísaných vrcholov stromu je $2 * r$. Vďaka vzťahom n,m a $VSIZE$ môžeme vzťah pre časovú zložitosť zjednodušiť a určiť ho ako $O(m\log n)$. Rovnako, počet prečítaných a zapísaných vrcholov je tiež $O(m\log n)$.

5.2 Maximum a minimum

Keďže hľadanie maximálneho a minimálneho údaju je za takýchto podmienok symetrickým problémom, v tejto časti budeme hovoriť o maxime, pričom uvedené postupy sa dajú priamočiaro prerobiť tak, aby ich výsledkom bolo minimum.

V situácii popísanej na začiatku kapitoly skutočne nie je možné o výsledku povedať nič, keďže máme k dispozícii iba výsledný údaj zo stromu a z bufferu. Chýba informácia o tom, či bol záznam s maximálnym údajom zo stromu prepísaný nejakým iným záznamom s rovnakou časovou pečiatkou z bufferu. Možnosťou, ako túto situáciu vyriešiť je o konštantu zväčšiť priestorové nároky vnútorných vrcholov stromu a pridať k výsledným údajom z agregácie aj časové pečiatky záznamu, v ktorom sa daný údaj nachádza. Takto je možné rozdeliť situáciu pri zlučovaní buffera s výsledkom zo stromu na dva prípady. Zlý prípad nastáva, ak platí, že maximálny údaj z bufferu je menší ako maximálny údaj zo stromu a k záznamu (c, u), obsahujúcemu maximálny údaj zo stromu existuje nejaký záznam (c, u') nachádzajúci sa v bufferi. V opačnom prípade je výsledok vypočítateľný - buď je globálne maximum z bufferu, alebo globálne maximum je zo stromu a nemohlo byť prepísané žiadnym záznamom v bufferi. Nevypočítateľná situácia je istým spôsobom rekurzívna - ak zahodíme prepísané maximum zo stromu a nejakým spôsobom získame zo stromu záznam, v ktorom je druhý maximálny údaj, situácia sa opakuje. Zjavne, všeobecne najhorší prípad nastáva v prípade, že sa zlá situácia vyskytne v každom opakovaní. To znamená, že postupne všetkých m maxím zo stromu je prepísaných záznamami z bufferu a pre maximálny údaj z bufferu u_b a m-tý maximálny údaj u_m zo stromu platí $u_b < u_m$.

Uvádzame náš základný algoritmus na výpočet maxima pri použití PQM bufferu v zjednodušenom modeli. Tento algoritmus používa na výpočet jednotlivých maxím zo stromu obojstranný prioritný front fixnej veľkosti $m + 1$. Toto obmedzenie znamená, že ak vloženie prvku spôsobí, že sa v ňom nachádza viac ako $m + 1$ prvkov, automaticky sa odstráni najmenší z nich (časová zložitosť takejto zloženej operácie je pri vhodnej implementácii, napr. intervalovými haldami [25] $O(\log m)$). Algoritmus kontroluje podmienky uvedené v predchádzajúcom odseku a ak nastane zlý prípad, prejde cestu z koreňa do listu, kde sa nachádza aktuálne maximum zo stromu. Počas tohoto

prechodu vkladá do frontu všetky záznamy, ktoré doposiaľ nevidel ak sú maximami zo súrodencov vrcholov, nachádzajúcich sa na ceste k listu s aktuálne spracúvaným maximom, alebo patria do tohoto listu. Všetky záznamy, ktoré pôvodné maximum zatienilo, t.j. všetky, ktoré boli od neho menšie na ceste od koreňa k nemu, sa takto dostanú do frontu. Tým je zabezpečené, že sa vo fronte nachádza korektný záznam s ďalším maximálnym údajom v poradí.

V nasledujúcej ukážke uvádzame vzorovú, kvôli zjednodušeniu nerekurzívnu implementáciu tohoto základného algoritmu v pseudojazyku podobnom Go. V premennej `ds` je obojstranný prioritný front fixnej veľkosti $m + 1$. Prvky, ktoré doň vkladáme sú trojice
⟨časová pečiatka, údaj,smerník na zdrojový vrchol stromu⟩, reprezentované štruktúrou `treeRec`. Metóda `Insert` implementuje sled operácii vloženia prvku, kontroly preplnenia a následného odstránenia najmenšieho prvku z frontu. Prvky sú v rámci tejto dátovej štruktúry porovnávané na základe hodnoty údajov. V premennej `dsBuffer` je hash tabuľka, ktorou overujeme, či buffer obsahuje k určitej časovej pečiatke záznam. Záznam časového radu je v nasledujúcom zdrojovom kóde reprezentovaný štruktúrou `Record` s atribútmi `Time` a `Val`, obsahujúcimi časovú pečiatku a údaj. Vrchol stromu je reprezentovaný štruktúrou `QTreeNode`, ktorá má atribút `core_block`, obsahujúci štruktúru s údajmi z vnútorného vrcholu a atribút `vector_block` obsahujúci údaje pre list. V prípade vnútorného vrcholu obsahuje `core_block` polia `Max` s maximálnymi údajmi z podstromov pre jednotlivých synov a `TimeMax` s časovými pečiatkami pre záznamy s maximálnymi údajmi z podstromov jednotlivých synov. Listová štruktúra `vector_block` obsahuje polia `Time` a `Value` obsahujúce časové pečiatky a údaje záznamov nachádzajúcich sa v liste a atribút `Len` obsahujúci počet záznamov, ktoré list obsahuje. Konštanty `KFACTOR` a `VSIZE` označujú aritu stromu a počet záznamov uložených v rámci jedného listu. Použité označenia existujúcich metód a štruktúr sú kompatibilné s označeniami v zdrojovým kódom systému BTrDB. Popis a vysvetlenie jednotlivých častí algoritmu uvádzame v ďalších odsekoch.

```
1  type treeRec struct{
2      time  int64
3      val   float64
4      src   *QTreeNode
5  }
6
7  func (root *QTreeNode) ComputeMax(PQMBuffer []Record) Record {
8      ds := LimitedPriorityDeque{}
9      dsBuffer := make(map[int64]struct{})
10
11     bMax := Record{Time: (MinimumTime-1) , Val: math.Inf(-1)}
12     for _, element := range PQMBuffer {
13         dsBuffer[element.Time] = struct{}{}
```

```
14          if bMax.Val < element.Val {
15              bMax = element
16          }
17      }
18      for i := 0; i < KFACTOR; i++ {
19          ds.Insert(treeRec{
20              time: root.core_block.TimeMax[i],
21              val:  root.core_block.Max[i],
22              src:  root})
23      }
24
25      for true {
26          gMax := ds.popMax()
27          _, contains := dsBuffer[gMax.time]
28
29          if bMax.val > gMax.val {
30              return bMax
31          } else if !contains {
32              return Record{Time:gMax.time, Val:gMax.val}
33          } else {
34              if gMax.src.isLeaf(){
35                  continue
36              } else {
37                  start := gMax.src
38                  for true {
39                      start, err = start.Child(start.ClampBucket(gMax.time))
40                      if err != nil {
41                          lg.Panicf("%v",err)
42                      }
43                      if start.isLeaf() {
44                          for i := 0; i < int(start.vector_block.Len); i++ {
45                              if gMax.time == start.vector_block.Time[i] {
46                                  continue
47                              }
48                              ds.Insert(treeRec{
49                                  time: start.vector_block.Time[i],
50                                  val:  start.vector_block.Value[i],
51                                  src:  start})
52                          }
53                          break
54                      } else {
55                          for i := 0; i < KFACTOR; i++ {
56                              if gMax.time == start.core_block.TimeMax[i] {
57                                  continue
58                              }
59                              ds.Insert(treeRec{
60                                  time: start.core_block.TimeMax[i],
```

```
61                                      val:  start.core_block.Max[i],
62                                      src:  start})
63 }   }   }   }   }   }   }
```

Výpočet je zabezpečovaný funkciou `ComputeMax`, spúšťanou na štruktúre `QtreeNode` označenej `root`, tvoriacej koreň stromu. Ako argument táto funkcia dostáva zoznam záznamov z PQM bufferu, označený názvom `PQMBuffer`.

Premenná `bMax` obsahuje záznam s maximálnym údajom z PQM bufferu. Na riadkoch 12 až 17 je napĺňaná hash tabuľka časových pečiatok, nachádzajúcich sa v PQM bufferi.

Na riadkoch 18 až 23 je napĺňaný front informáciami o maximách z jednotlivých podstromov koreňa. Toto napĺňanie je vykonávané ešte pred hlavným cyklom, aby sme zabezpečili invariant, že každý vrchol, na ktorý ukazuje smerník z atribútu `src` dátovej štruktúry získanej z frontu počas hlavného cyklu je už spracovaný, t.j. informácie z neho už sú do frontu vložené. Tento invariant zabezpečuje, že do frontu sa každý záznam dostane najviac raz.

V hlavnom cykle na riadkoch 25 až 63 obsahuje premenná `gMax` aktuálne spracúvané maximum zo stromu, t.j. záznam obsahujúci najväčší doposiaľ nespracovaný údaj.

V cykle nachádzajúcom sa vo vetve na riadkoch 36 až 63 sa vykonáva prechádzanie stromom ak nastane zlý prípad. Na riadku 39 sa vloží do premennej `start` smerník na potomka vrcholu, na ktorý doposiaľ ukazoval smerník v premennej `start` tak, že tento potomok má na starosti interval časových pečiatok, do ktorého patrí časová pečiatka záznamu v `gMax`.

Rozoberme časovú zložitosť tohoto algoritmu v najhoršom prípade, keď všetkých m záznamov z bufferu prepisuje záznamy s prvými m maximami zo stromu, pričom maximum z bufferu je menšie ako m-té maximum zo stromu a zároveň každý z týchto záznamov sa nachádza v inom liste. Zjavne, algoritmus musí lokalizovať a prejsť všetkých m listov, obsahujúcich záznamy s m maximálnymi údajmi zo stromu, pričom sú to záznamy práve z listov, ktoré by sa pri operácii **Flush** čítali a kopírovali. Znamená to, že algoritmus pri čítaní prejde v takomto prípade všetky vrcholy, ktoré by prešla operácia **Flush**. Vďaka udržiavaniu si smerníkov na vrcholy, v ktorých boli jednotlivé záznamy s maximálnym údajom objavené, nie je potrebné v každom kole hlavného cyklu prechádzať cestu z koreňa do listu s aktuálne spracúvaným záznamom celú, ale iba jej doposiaľ nenavštívenú časť. Platí teda, že každý prejdený vnútorný vrchol aj list je pri takomto prechode prečítaný práve raz. Označme r počet vrcholov, ktoré algoritmus (rovnako ako algoritmus z operácie **Flush**) prechádza. Riadky 10 až 20 majú odhadovanú časovú zložitosť $O(m + KFACTOR * \log m)$. Z frontu sa vyberá funkciou `popMax` počas celého behu algoritmu najviac $m + 1$ krát, pretože celý hlavný cyklus

sa otočí najviac toľko krát (každý záznam z bufferu prepíše najviac jedno z maxím stromu). Zložitosť všetkých výberov je teda $O(m\log m)$. Zostáva určiť, koľkokrát sa do frontu vkladá. Operácie vkladania do frontu je možné rozdeliť do dvoch skupín. Do prvej skupiny započítame záznamy vkladané z listov a maximálne záznamy, ktoré boli počas výpočtu prepísané obsahom bufferu, keďže počas výpočtu boli čítané aj listy, ktoré ich obsahovali. V druhej skupine sú ostatné záznamy, ktoré boli vložené do frontu z vnútorných vrcholov. Vložení prvej skupiny je $m * VSIZE$, počet vkladaní druhej skupiny je $((r-m) * KFACTOR - 1)$. Spolu je teda zložitosť tohoto algoritmu $O(m+KFACTOR*\log m+(m+1)*\log m+m*VSiZE+((r-m)*KFACTOR-1)+r)$ čo sa rovná $O(m + KFACTOR * \log m + (m+1) * \log m + m * VSIZE + (((m * (h - \log m) + 2m - 1) - m) * KFACTOR - 1) + m * (h - \log m) + 2m - 1)$. Vďaka vzťahom n,m a $VSIZE$ môžeme vzťah pre časovú zložitosť opäť zjednodušiť a určiť ho ako $O(m\log n)$

Ak považujeme každé čítanie a zapisovanie vrcholu stromu za diskovú operáciu, tak v tomto najhoršom prípade vykoná náš algoritmus rovnaké množstvo čítaní ako algoritmus z operácie **Flush** (operácia **Flush** samozrejme rovnaký počet vrcholov aj zapíše). Počet prenesených vrcholov je teda opäť $O(m\log n)$.

Je zjavné, že z praktického hľadiska má tento algoritmus niekoľko nevýhod. V prvom rade, hoci v porovnaní s udržiavaním PQM bufferu prázdneho ustavičným používaním operácie **Patch** šetrí priestor, jeho zložitosť je v uvedenom najhoršom prípade rovnaká ako zložitosť operácie **Flush**. Praktickosť jeho opakovaného spúšťania medzi operáciami **Flush** je teda diskutabilná. Naviac, algoritmus musí vykonávať réžiu spojenú s udržiavaním haldy. Je však zjavné, že uvedený najhorší prípad, keď záznamy z celého bufferu prepisujú práve záznamy s maximálnymi údajmi, z ktorých sa každý nachádza v inom liste je skutočne extrémny a v praktickom využití veľmi málo pravdepodobný. V praxi môžu pomôcť dva prístupy na ošetrenie tohoto zlého prípadu.

Prvé riešenie berie ohľad na ďalšie dopyty. Pracuje ako predchádzajúci algoritmus, ale ak po spracovaní prvých k maxím nastane zlý prípad, preruší vykonávanie algoritmu a spustí operáciu **Flush**. Po ukončení tejto operácie vráti globálne maximum z novovytvoreného stromu. Konštanta k musí byť vhodne zvolená tak, aby sa systém nevzdával priskoro, ale ani prineskoro. Ako ďalší atribút heuristiky môže byť vhodné použiť aktuálne zaplnenie PQM bufferu. Pri analýze algoritmu sme počítali s bufferom, ktorý bol celkom zaplnený a čakal na operáciu **Flush**. V reálnom použití môže byť buffer skoro prázdny a volanie operácie **Flush** by nebolo výhodné, keďže novovytvorené vnútorné vrcholy by zaberali viac priestoru ako samotné zapísané záznamy. Preto môže byť výhodné určiť konštantu $b > k$ určujúcu, že ak počet prvkov nachádzajúcich sa počas vykonávania funkcie `computeMax` v PQM bufferi je menší ako b, systém určite nespustí operáciu **Flush**. Hodnota tejto konštanty záleží od použitej konfigurácie BTrDB, konštanty m a tiež charakteristiky vkladaných dát. Totiž, ak mávajú záznamy

v bufferi pre konkrétny časový rad časové pečiatky také, že najmenší interval časových pečiatok, ktorý ich všetky pokrýva má malý rozdiel začiatočnej a koncovej časovej pečiatky, počet vrcholov, ktoré je treba vytvoriť pri operácii **Flush** je menší, keďže je pravdepodobné, že viacero vkladaných záznamov sa bude nachádzať v rovnakých listoch.

Samozrejme, je možné použiť aj opačný prístup, keď hodnota konštanty b hovorí o tom, či má systém vôbec začať vykonávať výpočet funkcie `computeMax` spojený s prechádzaním stromu. Môže postupovať spôsobom, že ak je buffer plnší ako b, a zároveň globálne maximum je zo stromu a je prepísané záznamom z bufferu, systém automaticky spustí **Flush**.

Taktiež, platí, že do frontu počas výpočtu `computeMax` netreba vkladať záznamy, ktorých údaj je menší ak minimálny údaj nachádzajúci sa vo fronte. V najhoršom prípade je postupnosť čítaných údajov taká, že takáto situácia nenastane ani raz, ale opäť, z praktického hľadiska môže byť vhodné túto podmienku ošetrovať a ušetriť tým vo volaniach určité vkladania, keďže je pravdepodobné že špecifický najhorší prípad nenastane.

5.3 Prúdové algoritmy

Napriek určitým odlišnostiam, niektoré aspekty nami uvažovaného problému zodpovedajú správaniu problémov, riešených prúdovými (streaming) algoritmami. Tieto algoritmy majú nasledovné vlastnosti [5]:

- Vstup pre algoritmus prichádza v prúde, čo znamená, že ho tvorí postupnosť sekvenčne prichádzajúcich prvkov, pričom algoritmus vykonáva výpočty v reakcii na jednotlivé prichádzajúce prvky.
- Systém nemá kontrolu nad poradím prichádzajúcich prvkov.
- Prúd má potenciálne neobmedzenú dĺžku.
- Akonáhle je nejaký prvok z prúdu spracovaný, je obyčajne zahodený alebo archivovaný. To znamená, že nie je možné opäť ho čítať okrem prípadu, keď si algoritmus tento prvok explicitne uloží do pamäte.

Vzhľadom na tretiu vlastnosť sú prúdové algoritmy obyčajne navrhované s obmedzenou priestorovou zložitosťou. Typicky je používaná polylogaritmická zložitosť. V nasledujúcom tvrdení dokazujeme, že algoritmus riešiaci prúdový problém, ktorý je v určitom zmysle podobný hlavnému problému má lineárnu priestorovú zložitosť od veľkosti prúdu.

Tvrdenie 1. *Uvažujme prúd prichádzajúcich záznamov časových radov $a_1 \dots a_n$, pričom n nie je vopred známe. Platnými údajmi po spracovaní prúdu označujeme také údaje, ktoré sa nachádzajú v nejakom zázname a_i a platí, že pre $j > i$ neexistuje záznam v prúde, ktorý by mal rovnakú časovú pečiatku, ako má a_i. Prvky množiny $\mathcal{U}$ sú dvojice $\langle c, b \rangle$, kde c je prirodzené číslo a b reťazec náhodných bitov, ktorého dĺžka je väčšia ako dĺžka zápisu čísla c. Jednotlivé údaje sú porovnávané na základe hodnoty c. Potom každý deterministický algoritmus, ktorý určí záznam s maximálnym z platných údajov má lineárnu priestorovú zložitosť od počtu vložených záznamov.*

Dôkaz. Označme b_i náhodný binárny reťazec vygenerovaný pre záznam a_i. Konštruujme vstup postupne a sledujme správanie algoritmu. Najskôr pridajme do algoritmom čítaného prúdu prvky a_1 až a_m v ktorých platí $a_i = (i, (m + i + 1, b_i))$. Zjavne, ak by prúd v tomto momente skončil, musí algoritmus vrátiť na výstup ako maximum záznam $(m, (m + m + 1, b_m))$. Vkladajme teraz do prúdu postupne prvky a_{m+1} až a_{2m}, pričom platí, že prvok a_j z týchto pridávaných je záznamom $(m-(j-m)+1, (m-(j-m), b_j))$. Zjavne, ak by prúd skončil po prvom vložení, musí algoritmus dať na výstup záznam $(m-1, (m+i, b_{m-1}))$. Každým vloženým prvkom sa ukazovateľ na záznam s platným maximom posunie v pôvodnom prúde akoby o jeden záznam smerom k skôr prijatým a na výstupe sa teda postupne objaví každý z pôvodných prvkov a_1 až a_m. Algoritmus si teda musel pamätať počas svojho vykonávania minimálne tieto záznamy, pričom je ich m a dĺžka celého prúdu bola $2m$. Počet prvkov, ktoré si algoritmus musel zapamätať je teda $\Omega(m)$ □

Domnievame sa, že tento výsledok môže pre dáta, ktoré môžu byť počas výpočtu prepísaním zneplatnené, naznačovať určitú vlastnosť, ktorá spôsobuje nárast priestorovej zložitosti a môže mať vplyv aj na obmedzenia riešení hlavného uvažovaného problému.

5.4 Zmena uložených údajov

Premýšľali sme aj o možnosti ukladať do vnútorných vrcholov stromu aj iné agregáty tak, aby bolo možné zrekonštruovať maximum rýchlejšie, hoci možno za cenu nie presného výsledku. Môže byť vhodné, aby po vyskúšaní k kôl základného algoritmu (po malú konštantu k) systém nevyvolal operáciu **Flush**, ale vrátil výsledok, ktorý určite nebude horší ako m-té maximum v konštantnom čase. Systém sa tiež môže úplne vyhnúť akémukoľvek prechádzaniu stromu a vrátiť iba výsledok na základe dát, ktoré má predpočítané v rámci koreňa. Dôležité však je, aby systém v odpovedi na dopyt nevracal údaje, ktoré sú väčšie ako skutočné maximum, keďže je možné, že v praxi bude PQM buffer obsahovať korekcie chybne nameraných extrémnych hodnôt.

Stále musí platiť, že výpočet agregátov uložených v nejakom vrchole môže používať iba agregáty, pochádzajúce zo synov tohoto vrcholu. Je to napríklad preto, aby pri vložení do jedného listu bolo možné prepočítať agregáty vo všetkých vrcholoch na ceste ku koreňu bez toho, aby bolo nutné čítať nejaké iné vrcholy, alebo dokonca celý strom.

Tvrdenie 2. *Nech vrcholy v_1 a v_2 majú spoločného otca v_3 a v každom z nich je uložených k maxím z ich podstromu. Potom najväčší počet maxím, ktoré môžeme vypočítať pre podstrom v_3 je k.*

Dôkaz. Sporom. Všetky záznamy, ktoré má algoritmus pri počítaní maxím do v_3 k dispozícii, sú práve dve sady po k záznamov z jeho synov. Majme teda algoritmus, ktorý vypočíta $k+1$ maxím z podstromu v_3 len za pomoci týchto dvoch sád. Uvažujme nasledovnú situáciu, nech najmenší z údajov záznamov z podstromu v_2 je väčší ako maximálny údaj zo záznamov podstromu v_1. Nazvime $a_1 \dots a_r$ všetky záznamy z podstromu s koreňom v_2, zoradené podľa údaju od najväčšieho po najmenší. Rovnako nazveme $a'_1 \dots a'_r$ záznamy z podstromu s koreňom v v_1. Zjavne, v takejto situácii je $k+1$ maxím zo záznamov z oboch podstromov v skutočnosti tvorených záznamami $a1 \dots a_{k+1}$, čo sú všetko záznamy z podstromu v_2. Algoritmus pri počítaní v_3 teda dá na výstup tieto záznamy. Hovorili sme však, že algoritmus nemá k dispozícii záznam a_{k+1}. To je spor. Algoritmus by sa mohol snažiť usporiadať $2k$ dostupných záznamov a vybrať z nich $k+1$ maxím, je však zjavné, že v tomto okrajovom prípade o priestore medzi najmenším z maxím jedného podstromu a maximom z druhého nemá dostatok informácií. □

Tvrdenie 3. *Nech vrcholy v_1 a v_2 majú spoločného otca v_3 a v každom z nich je uložených k maxím z ich podstromu. Ukážeme, že v tomto prípade je možné vypočítať aspoň k maxím.*

Dôkaz. Opäť použijeme značenie záznamov podstromov z predchádzajúceho dôkazu. Pomenujme pre každý záznam a_i údaj z neho ako u_i a pre každý záznam a'_i údaj z neho ako u'_i. Bez ujmy na všeobecnosti, nech pre údaje u_k a u'_k platí $u_k > u'_k$. Pri konštrukcii nového zoznamu postupuje algoritmus tak, že vykoná zlúčenie zoradených zoznamov z vrcholov v_1 a v_2 tak, aby vznikol nový zoznam, zoradený podľa údajov. Zjavne, ak by sme zoradili zoznam všetkých záznamov z podstromu v_3, majú so zoznamom, ktorý algoritmus získal zlučovaním, spoločných prvých j prvkov, kde j označuje pozíciu a_k. Je to preto, lebo o vzájomných vzťahoch všetkých záznamov s väčšími údajmi ako u_k má algoritmus informáciu zo zoznamov vrcholov v_1 a v_2. Ak $u_k > u'_1$ (analogicky $u'_k > u_1$), tak má algoritmus úplné informácie práve o prvých k maximách. Ak $u_k < u'_1$ (analogicky $u'_k < u_1$), tak má algoritmus úplné informácie dokonca viac ako k záznamoch s maximálnymi údajmi. □

V nasledujúcej časti skúmame postupne dve alternatívy zväčšenia počtu uložených záznamov.

Z praktického hľadiska je pre tento problém číslo m priveľké na to, aby sme v každom z vrcholov stromu ukladali m záznamov. Uvažovali sme teda o uchovávaní menšieho množstva záznamov, napríklad $logm$, čo je aj z praktického hľadiska realizovateľné.

Ako prvý prístup sme zvažovali ukladanie tzv. zarážok. Pre jednoduchosť pridávame predpoklad, že pre nejaké l platí, že $m = 2^{l-1} - 1$. Zarážky sú záznamy, obsahujúce postupne $1, 2, 4, 8 \ldots 2^l$-té maximálne údaje zo stromu. Každý vrchol obsahuje zarážky pre záznamy z každého podstromu svojich potomkov. Pri dopyte na maximum zo stromu môže systém v lineárnom čase od veľkosti PQM bufferu zistiť veľkosť maximálneho údaju v bufferi a skontrolovať, či sú zarážky po tú, čo obsahuje 2^{l-1}-tý maximálny údaj z koreňa, ktorých údaj je väčší ako maximálny údaj z bufferu prepísané záznamami z bufferu. Ak niektorá týchto zarážok nie je prepísaná, systém ju môže vrátiť. V opačnom prípade môže systém vrátiť maximum z bufferu okrem situácie, keď je tento maximálny údaj menší ako údaj zo zarážky 2^{l-1}. V takejto situácii systém vráti údaj zo zarážky, pretože, hoci je záznam tvoriaci túto zarážku prepísaný iným záznamom z bufferu, keďže $m < 2^{l-1}$, určite sa medzi ňou a prvým maximom zo stromu nachádza údaj aspoň z jedného neprepísaného záznamu, a teda hodnota údaju, ktorý systém takto vráti nie je menšia, ako z m-tého skutočného maxima.

Hľadali sme spôsob, akým je možné zarážky pri zmenách v strome udržiavať aktuálne. Najskôr sme uvažovali o možnosti, kedy by každá zarážka mala daný interval v rámci zoradených záznamov, v ktorom by sa mohla voľne pohybovať. Až v momente, keď by postupnými úpravami záznamov v strome opustila tento interval, nastalo by jej prepočítanie na základe zarážok z potomkov vrchola, v ktorom sa nachádzala. V istom zmysle je takýto prístup lenivý, keďže sa zarážku snaží prepočítať, až keď je to nevyhnutné. Ako vhodné intervaly sa nám javili tie, ktoré zasahovali práve do hraníc susedných zarážok. To znamená, že zarážka, reprezentujúca 2^j-té maximum by sa mohla pri vkladaní pohybovať medzi 2^{j+1}-tým a 2^{j-1}-tým maximálnym údajom z podstromu. Skutočnú aktuálnu polohu konkrétnej zarážky je možne udržiavať stále vypočítanú tým, že pri každej operácii vloženia vkladané záznamy zoradíme podľa hodnôt ich údajov a zistíme, koľko z nich bolo vložených medzi ktoré zarážky. Pri každej zarážke udržiavame počítadlo, ktoré počíta, koľko prvkov bolo vložených s hodnotou údaju väčšou ako je hodnota údaju v zarážke. Každým vložením záznamu teda inkrementujeme všetky počítadlá zarážok, ktorých hodnota údaju je menšia, ako má aktuálne vkladaný záznam. Pri návrate z listov smerom do koreňa zas vhodne dekrementujeme počítadlá zarážok, pri ktorých platí, že záznam s údajom, ktorý bol väčší ako ich údaj, bol prepísaný záznamom s menším údajom. Ako základ je samozrejme možné použiť iné číslo, 2 používame kvôli zvolenému m. Ak nejaká zarážka pri vložení

vystúpi zo svojho intervalu, na základe zarážok v svojich potomkoch sa prepočíta na novú hodnotu, ktorá už patrí do vymedzeného intervalu.

Tento prístup však nefunguje, keďže zarážky zo synov neposkytujú dostatočnú informáciu na vypočítanie opravených zarážok. Uvažujme nasledujúci príklad. Nech vrchol v_3 má synov v_1 a v_2. Predpokladajme, že zarážky vo všetkých týchto vrcholoch sú na správnej pozícii, t.j. akoby boli práve prepočítané a počítadlá k nim majú hodnotu 0. Vo vrchole v_3 označme z_i zarážku s 2^i-tým maximálnym údajom. Nech údaje všetkých záznamov, nachádzajúcich sa v podstrome s koreňom v_1 sú menšie, ako minimálny údaj zo záznamu v podstrome v_2. Predpokladajme, že strom s koreňom vo v_3 má dostatočnú hĺbku na to, aby počet záznamov v každom z podstromov s koreňmi v_1 a v_2 bol väčší ako 2^{l+1}. Znamená to teda, že všetky maximálne údaje až po poslednú zarážku v v_3 pochádzajú z podstromu v_2. Uvažujme postupnosť operácií, ktoré vkladajú vždy po jednom novom zázname, podľa časovej pečiatky patriacom do podstromu v_2. Nech $p = l - 3$. Každý záznam vkladaný operáciami tejto postupnosti má údaj, pre ktorý platí, že tento údaj je väčší ako hodnota údaju z z_p a zároveň menší ako hodnota údaju z z_{p-1}. Uvedená postupnosť obsahuje presne $p - 1$ operácií. Po vykonaní všetkých operácií z nej bude teda počítadlo z_p inkrementované toľko krát, že jeho nasledujúce zvýšenie spôsobí prepočítanie zarážky z_p. V ďalšej operácii z postupnosti je vložený záznam, ktorý podľa časovej pečiatky patrí do podstromu v_1 a má údaj, ktorý je väčší ako ktorýkoľvek z údajov záznamov, ktoré sú v celom strome s koreňom v_3. Zjavne, takáto operácia spustí prepočítanie zarážky z_p, keďže táto zarážka opustila svoj povolený interval. Prepočítanie však nemôže prebehnúť, pretože všetky záznamy z údajmi, ktoré patria do jej intervalu boli zapísané v predchádzajúcich operáciách a patria do podstromu s koreňom v_2. V aktuálnej operácii bol prechádzaný iba podstrom s koreňom v_1, o ktorom však vieme, že všetky jeho záznamy okrem aktuálne vloženého, ktorý sa použije na prepočítanie globálneho maximu, majú údaj ešte menší ako údaj v zarážke z_p, ktorú potrebujeme prepočítať. Zo stromu s koreňom v_2 sú k dispozícii iba zarážky z koreňa, ktoré však sú úplne totožné s zarážkami z v_3 pred operáciou posledného vkladania a teda neobsahujú žiadny záznam, ktorým by bolo možné nahradiť prepočítavaný z_p. Prepočítanie v takejto situácii teda nie je možné vykonať.

Zjavne, problém predchádzajúceho prístupu bol v tom, že záznamy s údajmi potrebnými na prepočítanie mohli byť do stromu uložené v skorších operáciách a neskoršie operácie, ktoré vyvolali potrebu prepočítavanie skutočne spustiť, už nemohli (bez čítania surových dát v podstromoch) poskytnúť správne údaje zo zarážok potomkov. Skúmali sme preto aj nie lenivý prístup a to, aké by mohol poskytnúť garancie. Keďže použitie logaritmického rozdelenia zarážok bolo motivované práve očakávaním využitia tohoto rozdelenia na lenivé prepočítavanie, pre jednoduchosť použime iba zarážky v prvom a aspoň w-tom maxime. Vzťah medzi w a m popíšeme na záver analýzy tohoto prístupu. Keďže sa jedná o nelenivý prístup, pri každej operácii vloženia sa hodnoty

zarážok vo všetkých zasiahnutých vrcholoch aktualizujú smerom odspodu, t.j. od listov ku koreňu. Pri každom vložení záznamov systém prečíta do pamäte minimálne jeden list, a tie listy, ktoré prečíta sú práve tie, v ktorých vykoná v rámci operácie vkladania nejakú zmenu. Keďže sú tieto zmenené listy už v pamäti, je možné pre nich priamočiaro vypočítať, ktoré záznamy obsahujú prvé a presne w-té maximum a tým získať presné zarážky. Uvažujme ďalej nad vrcholom v nasledujúcom poschodí. Nech jeho potomkami sú listy, ktorých zarážky nazývame a, b a a', b'. Z predchádzajúcej konštrukcie platí, že a aj a' sú záznamy obsahujúce prvé maximá z listov, ktorým patria a b a b' sú záznamy obsahujúce w-té maximá. Tieto záznamy porovnávame na základe údajov, ktoré obsahujú. Bez ujmy na všeobecnosti predpokladajme, že $a > a'$. Rozlišujeme tri prípady zlučovania týchto hodnôt, charakterizované nasledujúcimi podmienkami: $b > a'$, $b < b'$ a $b > b' \wedge b < a'$. Ak chceme zachovať invariant, že druhá zarážka obsahuje údaj, ktorý je aspoň w-tým maximom v celom podstrome, musíme pre zarážky aktuálne spracúvaného vnútorného vrcholu a'', b'' voliť nasledovne: $a'' = max(a', a)$ $b'' = max(b', b)$, pričom porovnávame údaje z uvedených záznamov. V prvom prípade bude zjavne pozícia druhej zarážky práve v w-tom maxime (respektíve vo všeobecnosti v rovnakej pozícii, ako mal potomok so zarážkami a, b). V druhom a treťom prípade však vieme iba to, že údaj zarážky b'' sa bude nachádzať medzi w-tým a $2w$-tým maximom (respektíve medzi minimom vzdialeností maxím z potomkov a súčtom týchto vzdialeností). Keďže nevieme povedať nič o údajoch záznamov, ktoré sa nachádzajú medzi zarážkami, nemôžme vypočítať, koľko z nich sa bude naozaj nachádzať medzi novými zarážkami. Tento istý vzťah platí pre všetky úrovne stromu, čiže výsledná pozícia zarážky v koreni je medzi w-tým a $2^h * w$-tým maximom z celého stromu. Aby bola hodnota zarážky v koreni použiteľná ako odpoveď na dopyt na maximum aj v prípade, že všetky záznamy bufferu prepisujú prvých m maxím, musí platiť, že $w > m$. Ako sme už uviedli, v praxi platí, že kapacita jedného listu je výrazne menšia ako kapacita bufferu m, a preto by v tomto zjednodušenom režime nebolo pre žiaden list možné vypočítať správnu začiatočnú hodnotu pre druhú zarážku. Túto situáciu môžeme vyriešiť pomocou čítania okolí. Identifikujeme najnižšiu úroveň v strome, pre ktorú platí, že podstromy vrcholov v nej obsahujú aspoň w záznamov. Tieto podstromy sa pre nás stanú okoliami, ktoré pri vkladaní záznamov spracúvame vždy v celku, bez ohľadu na to, ktoré listy by sme skutočne potrebovali prečítať. Týmto spôsobom simulujeme listy s väčšou kapacitou za cenu zvýšeného počtu čítaní. Ak však aj použijeme túto úpravu a za w zvolíme $m + 1$, exponenciálna zložka v zarážke v koreni spôsobí, že v najhoršom prípade, keď zlučovaním vždy zarážku posunieme čo najďalej od maxima, spraví táto zložka hodnotu $2^h * (m + 1)$-vého maxima nepoužiteľnú. Zjavne, exponenciálnosť vzniká kvôli nedostatku informácie o hodnotách medzi zarážkami a teda nutnosti počítať s najhorším možným prípadom, kedy sa zarážka posúva tak ďaleko, ako je to možné. Na vyriešenie tohoto problému sme zvažovali použitie troch zarážok, z ktorých

jedna by stále označovala skutočné maximum, jedna by bola dostatočne malá na to, aby exponenciálnym rastom dosiahla akurát želanú hodnotu $m + 1$ a tretia by bola väčšia, jej rast by sme však úmyselne spomaľovali tým, že by sme sa pri jej výpočte odrážali aj od vypočítaných hodnôt druhej zarážky. Absencia informácie o údajoch v záznamoch medzi zarážkami však spôsobila, že rozobratie prípadov aj v tomto režime ukázalo, že takto fungujúci systém počas výpočtu smerom nahor nedokáže dobre určiť, či pri zlučovaní naozaj nastáva prípad exponenciálneho rastu a teda nedokáže pozíciu zarážok vhodne regulovať.

Snaha získať výsledok na dopyt v takomto režime má podobný problém ako skôr popísaný prúdový algoritmus s jeho obmedzeniami. Dátová štruktúra stromu totiž principiálne nezachytáva vlastnosti údajov uložených záznamov a zameriava sa na usporadúvanie podľa času, čo je to hlavné vzhľadom na jej zameranie na uchovávanie časových radov. Vzhľadom k tomu, že zarážky neposkytujú dostatok informácie o hodnotách údajov obsiahnutých v záznamoch jednotlivých podstromov, skúmali sme ešte ďalšiu možnosť, rozširujúcu základný algoritmus prechádzajúci strom, ktorá z praktického hľadiska môže vylepšiť jeho efektívnosť.

5.5 Úprava základného algoritmu

Základný algoritmus musel pri prechode stromom vstupovať v najhoršom prípade do m listov, keďže potreboval zistiť, aké údaje boli zatienené väčšími údajmi zo záznamov, ktoré však boli prepísané záznamami v bufferi. Uvedieme spôsob, ktorý zabezpečí, že počet vrcholov, ktoré je pri výpočte potrebné čítať bude nižší. Je možné zvoliť konštantu g takú, že $VSIZE > g > 1$ a miesto zarážok v každom z vrcholov stromu uchovávať záznamy s presne g maximálnymi údajmi podstromu daného vrcholu. Tieto záznamy je podľa tvrdení, uvedených v predchádzajúcich častiach, možné udržiavať pri aktualizáciách pre každý vrchol vypočítané iba s pomocou záznamov zo synov. Zjavne, pre $g = 1$ dostávame pôvodný základný algoritmus.

V nasledujúcej ukážke uvádzame vzorovú implementáciu tohoto rozšíreného algoritmu v pseudojazyku podobnom Go. Použité označenia nezmenených objektov sú rovnaké ako v predchádzajúcej ukážke. Konštanta `CONST_G` označuje zvolenú konštantu g. Štruktúra `treeRec`, používaná vo fronte je rozšírená o prvok `rank`, označujúci poradie záznamu v rámci zoznamu g maxím z konkrétneho vnútorného vrcholu. Hodnota -1 v tomto prvku je vyhradená na označenie záznamu pochádzajúceho z listu. Zmena nastala aj v štruktúre `QTreeNode.core_block` obsahujúcej dáta vnútorného vrcholu. Táto štruktúra po novom obsahuje pole `Max`, ktorého prvkami sú polia dĺžky g, obsahujúce prvých g maximálnych údajov z podstromu pre daného potomka a tiež rovnako štrukturované pole `TimeMax` s časovými pečiatkami pre záznamy, obsahujúce

uvedené údaje.

```
type treeRec struct{
    time  int64
    val   float64
    src   *QTreeNode
    rank  int64
}

func (root *QTreeNode) ComputeMax(PQMBuffer []Record) Record {
    ds := LimitedPriorityDeque{}
    dsBuffer := make(map[int64]struct{})

    bMax := Record{Time: (MinimumTime-1) , Val: math.Inf(-1)}
    for _, element := range PQMBuffer {
        dsBuffer[element.Time] = struct{}{}
        if bMax.Val < element.Val {
            bMax = element
        }
    }
    for i := 0; i < KFACTOR; i++ {
        for j := 0; j < CONST_G ; j++ {
            ds.Insert(treeRec{
                time: root.core_block.TimeMax[i][j],
                val:  root.core_block.Max[i][j],
                src:  root,
                rank: j})
        }
    }

    for true {
        gMax := ds.popMax()
        _, contains := dsBuffer[gMax.time]

        if bMax.val > gMax.val {
            return bMax
        } else if !contains {
            return Record{Time:gMax.time, Val:gMax.val}
        } else {
            if (gMax.src.isLeaf() || gMax.rank < CONST_G -1){
                continue
            } else {
                start := gMax.src
                for true {
                    start, err = start.Child(start.ClampBucket(gMax.time))
                    if err != nil {
                        lg.Panicf("%v",err)
```

```
46                      }
47                      if start.isLeaf() {
48                          for i := 0; i < int(start.vector_block.Len); i++ {
49                              if ( gMax.val < start.vector_block.Value[i] ||
50                                  gMax.time == start.vector_block.Time[i] ){
51                                  continue
52                              }
53                              ds.Insert(treeRec{
54                                  time: start.vector_block.Time[i],
55                                  val:  start.vector_block.Value[i],
56                                  src:  start,
57                                  rank: -1})
58                          }
59                          break
60                      } else {
61                          child_overwritten := false
62                          for i := 0; i < KFACTOR; i++ {
63                              flag_child := true
64                              for j := 0; j < CONST_G ; j++ {
65                                  if (gMax.val <
66                                    start.core_block.Max[i][j] ||
67                                    gMax.time ==
68                                    start.core_block.TimeMax[i][j]){
69                                      continue
70                                  }
71                                  ds.Insert(treeRec{
72                                      time: start.core_block.TimeMax[i][j],
73                                      val:  start.core_block.Max[i][j],
74                                      src:  start,
75                                      rank: j})
76                                  flag_child = false
77                              }
78                              if flag_child {
79                                  child_overwritten = true
80                              }
81                          }
82                          if !child_overwritten {
83                              break
84  }   }   }   }   }   }   }
```

Viacero zapamätaných maxím umožňuje algoritmu na riadku 38 zostupovať do potomka až keď je to nevyhnutné, t.j. všetkých g záznamov s maximálnymi údajmi, ktoré k nemu boli uložené v jeho otcovi, je prepísaných záznamami s menšími údajmi z bufferu. Táto vlastnosť tiež spôsobuje, že nie je potrebné každý zostup skončiť až v liste, ale stačí zostúpiť po úroveň, v ktorej už pre všetky zoznamy maxím potomkov platí, že v rámci zoznamu je aspoň jeden záznam doposiaľ nespracovaný. Vďaka tomu, že vo

všetkých vrcholoch je uchovaných práve g maxím, situáciav ktorej je potrebné zostupovať do ďalších úrovní nastáva iba ak všetkých g prepísaných záznamov s maximami z nejakého vrcholu patrilo do intervalu časových pečiatok iba jedného z jeho synov. Takúto situáciu detegujeme pomocou premennej `child_overwritten`.

Dôsledok takejto zmeny algoritmu je, že strom, tvorený vrcholmi, ktoré čítal základný algoritmus s $g = 1$ bude pre algoritmus s iným g mať skrátené všetky vetvy tak, že ak z podstromu určitého vrcholu bolo zatienených menej ako g maximálnych záznamov, vrcholy pod ním nebudú čítané. Najhorší prípad vtedy nastáva, keď je čítaných $r = (m/g) * (h - \log(m/g)) + (2m/g) - 1$ vrcholov.

Hoci takáto úprava zníži počet čítaných vrcholov, čiže čítacích operácií, bola dátová štruktúra každého vrcholu zväčšená, a teda je potrebné čítať viac dát pre každý čítaný vnútorný vrchol. Na druhej strane, tieto pridané čítania vždy patria do pôvodných dátových štruktúr, čo môže byť výhodnejšie ako čítanie menších dátových štruktúr na viacerých adresách. Zároveň, keďže g je konštanta, hoci sa z teoretického hľadiska vo výslednej časovej zložitosti a odhade počtu prenesených vrcholov neprejaví, z praktického hľadiska tento prístup môže byť vhodný. Naviac, ako už bolo uvedené, z praktického hľadiska je výskyt skutočne najhoršieho možného vstupu mimoriadne nepravdepodobný. Najhorší vstup pre uvedený algoritmus nastáva, ak maximálny údaj z bufferu je menší, ako m-tý maximálny údaj zo stromu a zároveň platí, že časové pečiatky záznamov v bufferi sú totožné s časovými pečiatkami záznamov, obsahujúcich prvých m maxím údajov zo stromu. Pre tieto časové pečiatky tiež musí platiť, že sa nachádzajú správne rozmiestnené v dostatočne vzdialených skupinách tak, aby algoritmus musel prečítať čo najviac vrcholov. Ak by bola v heuristike základného algoritmu vhodne zvolená konštanta k tak, že v prostredí, v ktorom by bol systém používajúci takýto algoritmus nasadený, by kvôli charakteristike dát nachádzajúcich sa v bufferi algoritmus nespúšťal operáciu **Flush**, ale k výsledku by dospel vždy počas k kôl základného algoritmu, pri zvolení $g = k$ pre upravený algoritmus by tento systém vedel vypočítavať odpovede na dopyty priamo zo zoznamov záznamov, nachádzajúcich sa v koreni a algoritmus by strom vôbec nemusel prechádzať. Z tohto dôvodu pokladáme tento algoritmus fungujúci v rámci zjednodušeného modelu za vhodné riešenie skúmaného problému.

Záver

Spracúvanie dát časových radov je v dnešnej dobe kľúčovou súčasťou mnohých procesov. Je preto potrebné mať k dispozícií systémy, ktoré s takýmito dátami dokážu pracovať efektívne. Naviac, je dôležité, aby tieto systémy poskytovali rozšírenú funkcionalitu vo forme podpory uchovávania historických verzií a agregácií.

Na základe poznatkov nadobudnutých počas experimentu z predchádzajúcej práce sme vybrali a preskúmali niektoré dostupné systémy na správu časových radov a identifikovali a popísali sme špecifiká ich dátového modelu, ktoré ich robia nevhodnými kandidátmi na pridanie podpory historických verzií. Taktiež sme dôsledne preštudovali systém BTrDB, vrátane jeho vnútorných mechanizmov a identifikovali sme niektoré nedokumentované vlastnosti. Počas tohoto štúdia sa nám podarilo odhaliť funkčné nedostatky tohoto systému, ktoré sme na základe nadobudnutých poznatkov opravili. Taktiež sme doplnili experimentálnu podporu nových funkcionalít vhodných do praxe, vrátane nového spôsobu ošetrovania chybne vložených záznamov. Systém sme tiež rozšírili o podporu uchovávania historických verzií a rozobrali sme dôsledky tejto úpravy na garancie, ktoré poskytuje na presnosť výsledkov agregácií.

Literatúra

[1] Mark Adamiak, Bogdan Kasztenny, and William Premerlani. Synchrophasors: definition, measurement, and application. *Proceedings of the 59th Annual Georgia Tech Protective Relaying, Atlanta, GA*, pages 27–29, 2005.

[2] Michael P Andersen and David E Culler. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *FAST*, pages 39–52, 2016.

[3] Michael P Andersen, Sam Kumar, Connor Brooks, Alexandra von Meier, and David E Culler. DISTIL: Design and implementation of a scalable synchrophasor data processing system. In *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 271–277. IEEE, 2015.

[4] Arctic TimeSeries and Tick store. [Citované 2017-12-21] Dostupné z `https://github.com/manahl/arctic`.

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[6] Andreas Bader, Oliver Kopp, and Michael Falkenthal. Survey and comparison of open source time series databases. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband*, 2017.

[7] Tony F Chan, Gene H Golub, and Randall J LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37(3):242–247, 1983.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media, Inc., 2013.

[10] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[11] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[12] Ted Dunning and Ellen Friedman. *Time Series Databases: New Ways to Store and Access Data*. O'Reilly Media, 2015.

[13] etcd: Distributed reliable key-value store for the most critical data of a distributed system. [Citované 2019-2-20] Dostupné z `https://github.com/etcd-io/etcd`.

[14] Filip Janitor. *Systém na správu periodických časových radov*. Bakalárska práca, UK, Bratislava, 2017.

[15] Alexander Kiel. Datomic-a functional database. 2013.

[16] Sam Kumar, Michael P Andersen, and David E Culler. Mr. Plotter: Unifying Data Reduction Techniques in Storage and Visualization Systems. 2018.

[17] Florian Lautenschlager, Michael Philippsen, Andreas Kumlehn, and Josef Adersberger. Chronix: Long term storage and retrieval technology for anomaly detection in operational data. In *FAST*, pages 229–242, 2017.

[18] OpenTracing API Consistent, expressive, vendor-neutral APIs for distributed tracing and context propagation. [Citované 2019-2-20] Dostupné z `https://github.com/opentracing`.

[19] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[20] Vijay Pai. gRPC Design and Implementation, 2016. Stanford Platform Lab Seminar.

[21] Philippe Pierre Pebay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. Technical report, Sandia National Laboratories, 2008.

[22] Smart grid store. [Citované 2019-2-20] Dostupné z `https://docs.smartgrid.store/`.

[23] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of SciDB. In *Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

[24] Apache HBase Team. Apache HBase reference guide. *Apache, version*, 2(0), 2016.

[25] Jan van Leeuwen and Derick Wood. Interval heaps. *The Computer Journal*, 36(3):209–216, 1993.

[26] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[27] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 35–44. ACM, 2007.

[28] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

Príloha A

Súčasťou práce sú zdrojové kódy z repozitárov, v ktorých sme počas práce robili úpravy. Nachádzajú sa v priečinkoch `pyt/`, `go/`, `go_fix/`.

Priečinok `pyt/` obsahuje repozitár s upravenou knižnicou pre jazyk Python. Príklady jej použitia sa nachádzajú v jej balíku v súbore
`pyt/btrdb-python/README.md`.

Priečinok `go/` obsahuje repozitár knižnice v jazyku Go, systému BTrDB s pridanou podporou variancie a uchovávania historických verzií a tiež repozitár systému SmartGridStore. Obsah tohoto priečinku je pre kompilovanie zdrojových kódov a vytvorenie obrazov pre nástroj Docker potrebné premiestniť do priečinku `$GOPATH/src`.

Na kompilovanie a spúšťanie systému je potrebné mať nainštalovaný a správne nakonfigurovaný jazyk Go, nástroj Docker, python, rôzne balíčky pre Go, nástroj Dep a knižnicu pre RADOS. Potrebné knižnice pre RADOS sú dostupné v nasledujúcich balíčkoch inštalovateľných nástrojom `apt`:

```
libcephfs-dev librbd-dev librados-dev
```

Nástroj Dep je možné nainštalovať príkazom

```
sudo apt-get install go-dep
```

Potrebné závislosti pre Go je možné nainštalovať nasledujúcimi príkazmi, nachádzajúcimi sa tiež v súbore `deps/prerequisities.sh`

```
go get -u github.com/jteeuwen/go-bindata/...
go get github.com/golang/protobuf
go get google.golang.org/grpc
go get github.com/huichen/murmur
go get github.com/pborman/uuid
go get github.com/grpc-ecosystem/go-grpc-middleware
go get github.com/grpc-ecosystem/grpc-gateway
go get github.com/maruel/panicparse
cd $GOPATH/src/github.com/maruel/panicparse
go build
mv panicparse $GOPATH/bin/panicparse
```

Kompiláciu BTrDB následne zabezpečuje skript

```
$GOPATH/src/github.com/BTrDB/smartgridstore/devtools/make-
    release.sh
```

ktorý stiahne zostávajúce balíčky pre Go a súčasti obrazov pre Docker z internetu a vytvorí potrebné obrazy pre nástroj Docker. Následne je možné spustiť vývojové prostredie. Je pri tom potrebné postupovať podľa návodu v súbore

```
$GOPATH/src/github.com/BTrDB/smartgridstore/devmachine/README
    .md
```

Spúšťanie vývojového prostredia môže tiež sťahovať obrazy niektorých komponentov pre nástroj Docker z internetu.

Knižnica pre jazyk Go sa nachádza (po premiestnení súborov z priečinka `go/`) v priečinku

```
$GOPATH/src/github.com/BTrDB/btrdb/
```

Repozitár so zdrojovými kódmi systému BTrDB sa nachádza (po premiestnení súborov z priečinka `go/`) v priečinku

```
$GOPATH/src/github.com/BTrDB/btrdb-server/
```

Priečinok `go_fix/` obsahuje repozitár systému BTrDB bez pridanej variancie a podpory historických verzii. Obsahuje iba zmeny súvisiace s opravami systému a pridaním detekcie duplikovaných časových pečiatok. Obsahom tohoto priečinku je možné nahradiť priečinok

```
$GOPATH/src/github.com/BTrDB/btrdb-server/
```

a takto upravený systém skompilovať podobným spôsobom ako je popísaný v predchádzajúcich odsekoch. Na interakciu s ním je však potrebné použiť neupravené knižnice a neupravený obraz pre nástroj `apifrontend`. Vhodné je teda použiť oficiálny repozitár so zdrojovými kódmi pre SmartGridStore a používať oficiálne skripty na kompiláciu.

K dispozícii je aj DVD, na ktorom sú dostupné skompilované obrazy pre nástroj Docker, umožňujúce priamo spustiť vývojárske prostredie s bežiacim BTrDB, obsahujúcim naše úpravy pridávajúce podporu historických verzií a varianciu. Pre ich používanie je potrebné najskôr v súbore `runnable/environment.sh` nakonfigurovať v riadku 6 v premennej `DEVMACHINE_BASE` cestu do priečinka, kam má bežiaci systém ukladať svoje pracovné súbory. Pre spustenie systému následne stačí vykonať z priečinku `runnable` nasledujúce príkazy:

```
source environment.sh
sudo -E ./start.sh
```

Druhý z príkazov pri prvom spustení z internetu stiahne obrazy niektorých potrebných komponentov pre nástroj Docker. Na zastavenie bežiaceho systému je potrebné vykonať príkaz

```
sudo -E ./teardown_devmachine.sh
```