Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

# Bakalářská práce

# Uživatelské výpočty nad time-series databází

Plzeň 2022 Petr Holický

**Místo této strany bude zadání práce.**

# Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 4. května 2022

Petr Holický

# Abstract

The thesis focused on time-series databases and the application of equations on data. Several databases were compared and based on the results one was selected. A comparison on two programming languages was also done on this database. The final program can process user equations, apply them on the data and the calculated results insert back into the database. It is also capable of making and writing out summaries in given time frame.



# Abstrakt

Práce se zabývala time-series databázemi a aplikací vzorečků na data. Bylo provedeno porovnání několika time-series databází a na základě výsledků byla jedna vybrána. Na této databázi také proběhlo porovnání dvou programátorských jazyků. Výsledný program dokáže zpracovat uživatelské vzorečky, aplikovat je na data a vypočtené výsledky zapsat zpět do databáze. Také je schopný vytvořit a vypsat sumace v zadaném časovém rozmezí.

# Obsah

# 1 Úvod

Cílem práce je vytvořit program pro time-series databázi, který zpracovává uživatelem zadané vzorečky. Time-series databáze slouží k efektivnímu ukládání dat v reálném čase a jejich následnému zpracování. Vzorečky provádí výpočty nad uloženými daty. Program bude schopný si poradit s různými typy dat a jejich rozdílnými hodnotami. Účelem programu je usnadnění práce s daty z databáze a aby bylo možné použít stejné vzorečky na odlišná data bez nutnosti ruční úpravy z důvodu odlišnosti ve struktuře dat. Program dovolí uživateli aplikovat vzorečky na data a vytvářet shluky či sumace z dat.

Čtenář je nejdříve seznámen s time-series databázemi, vybráním vhodné databáze, jazyka a popisu vzorečků a výpočtů. Další kapitola popisuje strukturu aplikace, použité třídy a jejich metody. Tuto část lze označit za programátorskou dokumentaci. Následující kapitola se zabývá testováním a důvodem náročnosti. Poslední kapitola popisuje budoucí vylepšení programu.

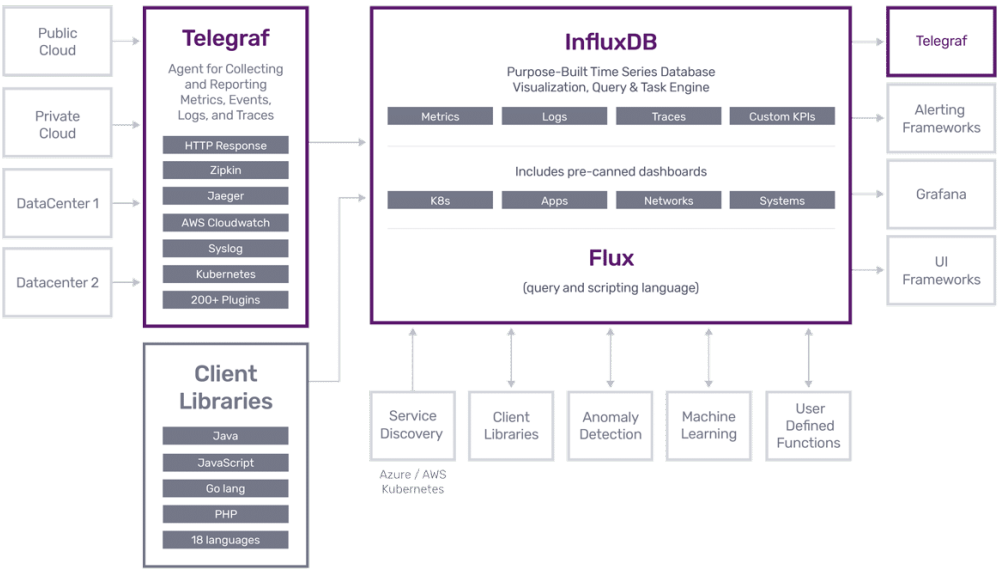
# 2 Výběr databáze

## 2.1 Time-series databáze

Time-series databáze slouží k efektivnímu ukládání dat, která se konstantě generují z běžících událostí. Mezi tyto události můžou patřit metriky ze serverů, aplikací, IoT senzorů, uživatelské interakce s aplikacemi nebo aktivity na finančním trhu. Time-series databáze jsou charakterizovány vysokou rychlostí zápisu dat, která obsahují časovou stopu, sumarizovanými pohledy na data a možností přístupu k datům ve vybraném časovém období. Přestože i normální databáze umožnují do určité míry pracovat s časově závislými daty, time-series databáze jsou navrženy tak, aby dokázaly efektivně pracovat s velkým množstvím dat. Práce s velkým množstvím dat zahrnuje sumarizaci a formátované výpisy. Svou efektivitu oproti ostatním databázím dosahují kompresí a životními cykly. Životní cyklus dat spočívá v jejich agregaci a podvzorkování. Data s velkou přesností za malý úsek času jsou sumarizována a znovu zpět uložená pro další výpočty či sumarizace.

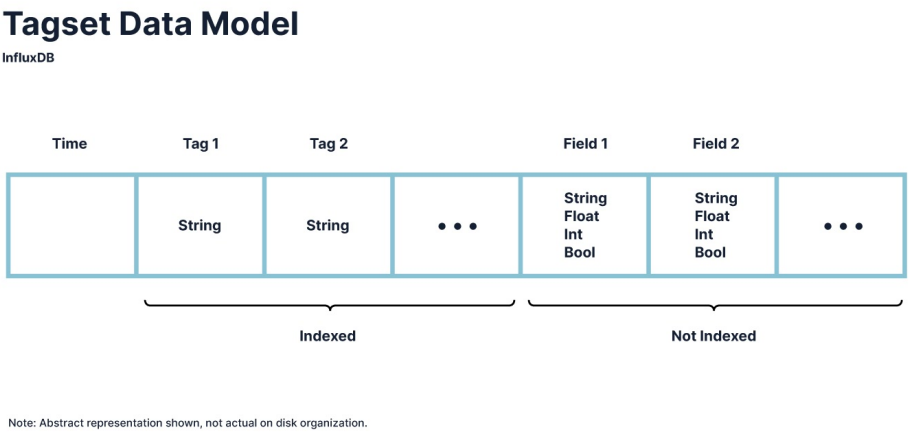
## 2.2 InfluxDB

První volbou time-series databáze se jeví InfluxDB, která je aktuálně jedna z nejvíce populárních time-series databází. Architektura InfluxDB před vydáním verze 2.x se skládala z TICK stack architektury, kterou lze vidět na obrázku 2.1 [2]. Telegraf je agent, který shromažďuje a nahlašuje různé události nebo metriky. Chronograf slouží jako kompletní interface a kapacitor je engine pro zpracování streamu dat. InfluxDB 2.x tuto architekturu zjednodušil za účelem zabalení celé TICK stack architektury do jediného binárního souboru.




Obrázek 2.1: Architektura InfluxDB verze 1.x

Forma dat v InfluxDB se skládá z: `measurement-name tag-set field-set timestamp`, grafické znázornění na obrázku 2.2 [2]. Measurement je řetězec, tag-set se skládá z klíče a řetězcové hodnoty, field-set se skládá z klíče a číselné hodnoty a timestamp je čas, který může nabývat až nanosekundové přesnosti [4].



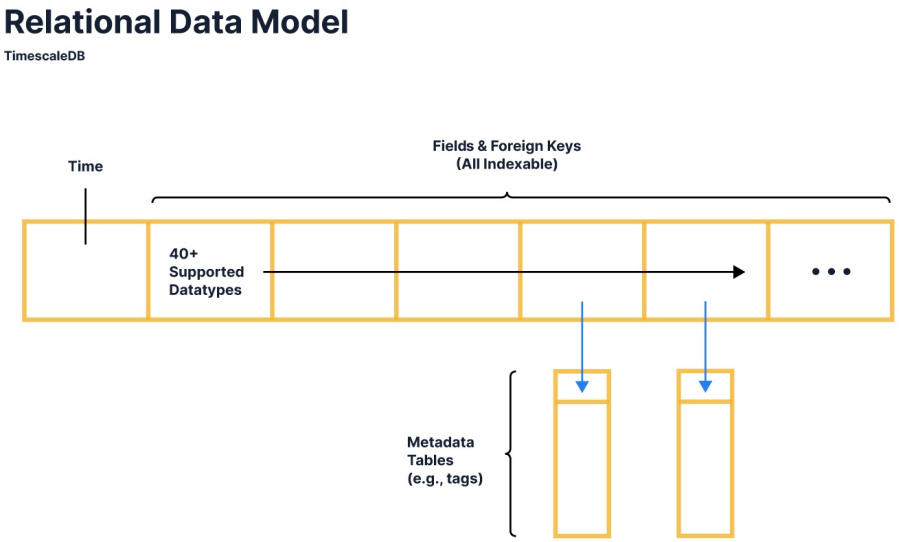

Obrázek 2.2: Struktura dat

Jedna z hlavních vlastností InfluxDB je, že nepotřebuje definovat schéma před vkládáním dat. Schéma je automaticky vytvořeno z tagů dat vložených.

To znamená, že pro datasety s nízkou kardinalitou je velice jednoduchý pro použití, ale má problémy s datasety vyžadující indexaci nebo s tagy, které se často mění. Také na rozdíl od jiných databází InfluxDB využívá svůj vlastní skriptovací jazyk Flux oproti typickému SQL.

## 2.3 TimescaleDB

Jako druhá možnost time-series databáze se nabízí TimescaleDB, která na rozdíl od vlastního datového modelu využívá tradiční relační datový přístup. Relační datový model ukládá každý časový záznam do vlastní řádky s časovou stopou a libovolným počtem dalších položek. Tyto položky mohou být číselného, řetězcového nebo boolean typu anebo tvořit pole, měny, binární data a další komplexnější typy. Indexem může být jakákoliv z těchto položek, a dokonce i více položek najednou. Položky mohou také být cizím klíčem pro propojení s dalšími tabulkami, ukázku modelu lze vidět na obrázku 2.3 [6].



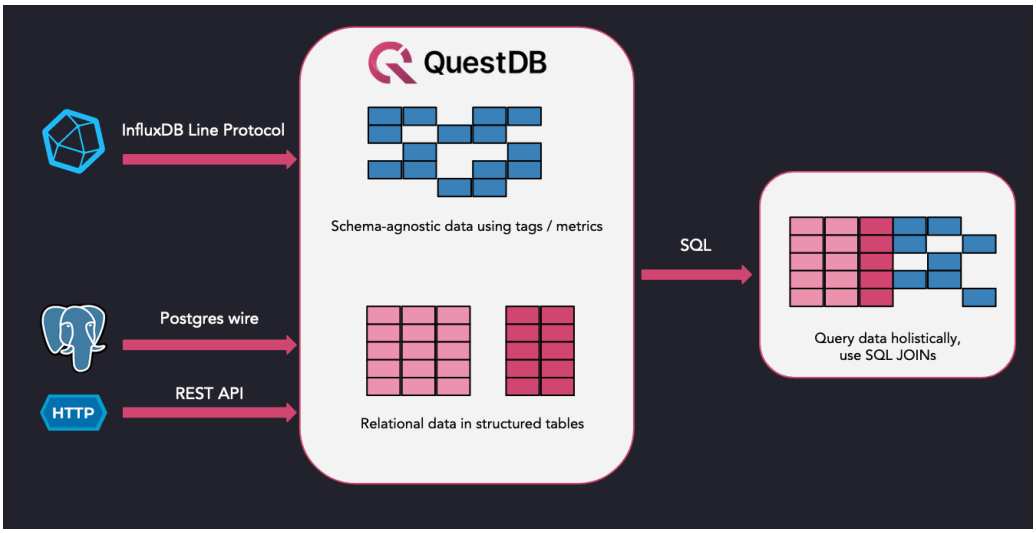

Obrázek 2.3: Relační datový model time-series databází

Tento přístup umožnuje data lépe specifikovat, ukládat a kontrolovat jejich konzistenci. Díky přesnějšímu způsobu popisu a ukládání dat se zlepší výkon při vyhledávání dat a sníží se velikost dat na disku. Nevýhodou je nutnost specifikace schématu před samotným vkládáním dat, což je přímý

opak od InfluxDB, která definici schématu nepotřebuje a schéma se generuje z dat vložených. TimescaleDB také, na rozdíl od InfluxDB, využívá tradiční SQL pro dotazy.

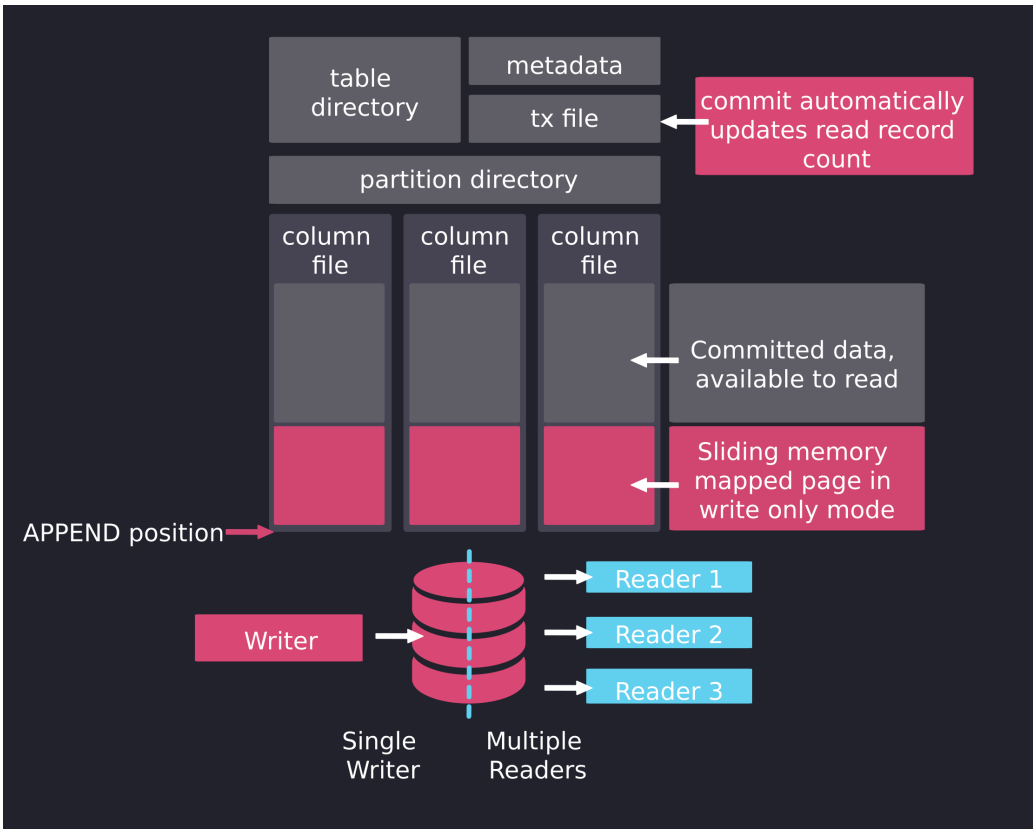
## 2.4 QuestDB

Třetí volbou je QuestDB, která je jedna z novějších time-series databází. QuestDB využívá řádkový protokol databáze InfluxDB, bez obětování výkonu. QuestDB vyřešil problém s kardinalitou pomocí obsáhlé paralelizace hashmapových operací na indexovaných sloupcích [3]. QuestDB také využívá SIMD pro paralelní provádění procedur souvisejícími s indexy a vyhledáváním v hashmapě. SIMD je specifická sada CPU instrukcí pro aritmetické výpočty využívávající syntetickou paralelizaci [1]. Na rozdíl od InfuxDB, která využívá svůj vlastní skriptovací jazyk Flux, QuestDB používá tradiční SQL pro dotazy. Strukturu QuestDB lze vidět na obrázku 2.4 [2].




Obrázek 2.4: Struktura QuestDB time-series databáze

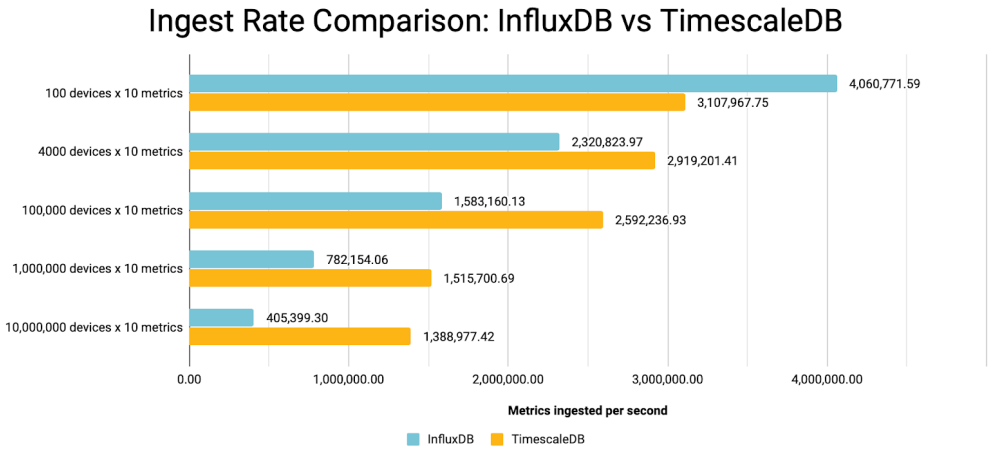
QuestDB využívá sloupcově zaměřený model pro ukládání dat. Data jsou ukládána do tabulek a každý sloupec je uložen ve svém vlastním souboru. Nová data se vkládají na konec každého sloupce, což umožňuje přístup k datům podle pořadí vložení. Čtení a zápis dat používá mapování souborů do paměťových stránek. Konzistenci dat zaručuje atomickými operacemi. Data zpracovaná jedním procesem můžou být rovnou využitá procesem jiným pomocí náhodného přístupu nebo přes datové fronty. Ilustraci modelu lze vidět na obrázku 2.5 [10].



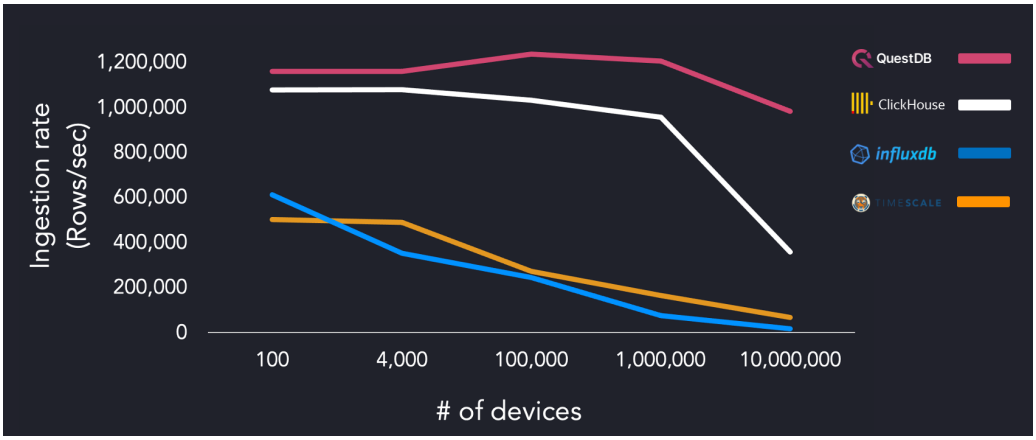

Obrázek 2.5: Model pro ukládání dat v QuestDB

## 2.5 Porovnání Time-series databází

Přestože InfluxDB je jedna z nejpopulárnějších time-series databází, ostatní databáze jako TimescaleDB a QuestDB dokážou nabídnout lepší výkon při ukládání dat oproti InfluxDB. Hlavním důvodem je, jak už bylo zmíněno, problém s kardinalitou. S rostoucím počtem různých zařízení klesá výkon InfluxDB značně oproti TimescaleDB a QuestDB, jak lze vidět na obrázku 2.6 a 2.7 [2].

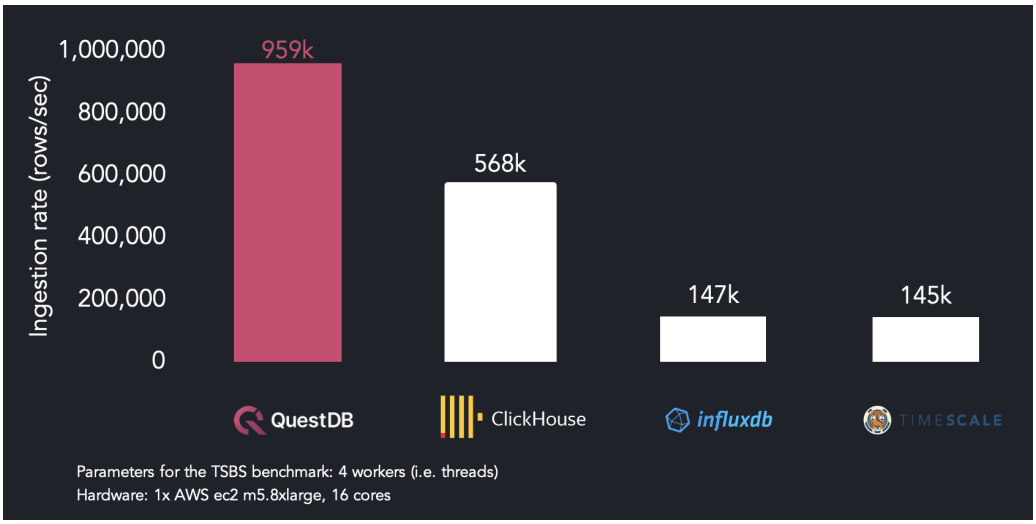



Obrázek 2.6: Počet vložených dat za sekundu v InfluxDB vs TimescaleDB



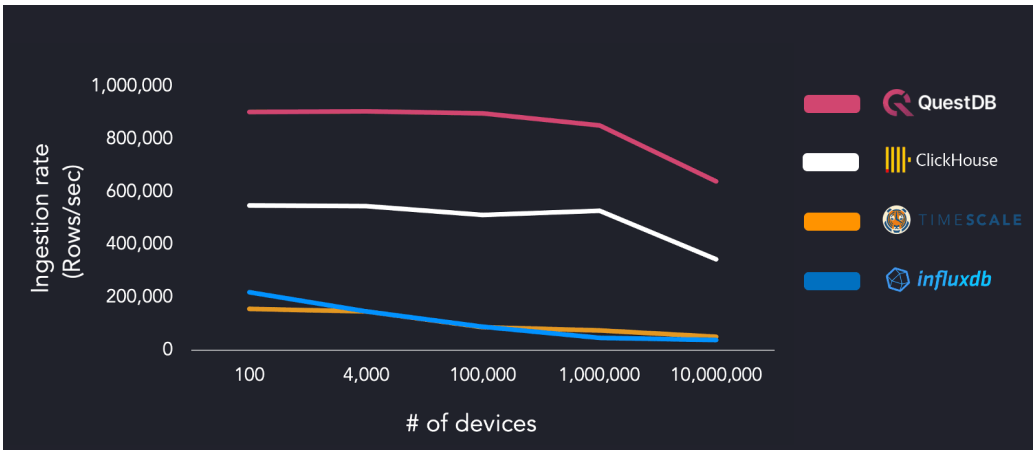

Obrázek 2.7: Porovnání počtu vložených řádek za sekundu

Z dalšího porovnání QuestDB oproti ostatním time-series databázím můžeme na obrázku 2.8 [11] vidět, že QuestDB je 1,7krát rychlejší než ClickHouse, 6,4krát rychlejší než InfluxDB a 6,5krát rychlejší než TimescaleDB.

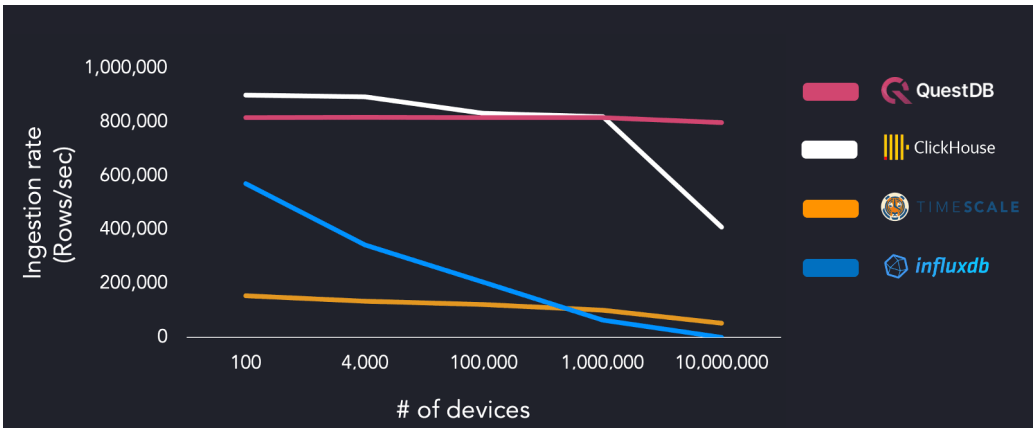



Obrázek 2.8: Srovnání rychlosti QuestDB s ostatními databázemi

Při využití více vláken můžeme na obrázcích 2.9 a 2.10 [11] vidět, že výkon QuestDB s postupným zvyšováním kardinality vypadá téměř konstantě.




Obrázek 2.9: Rychlost QuestDB s využitím 4 vláken




Obrázek 2.10: Rychlost QuestDB s využitím 16 vláken

V rychlosti výpisu dat je TimescaleDB při vysoké kardinalitě až 7000krát rychlejší oproti InfluxDB [14] a QuestDB je 2,7krát rychlejší než TimescaleDB [13]. Při porovnání datových schémat již víme, že InfluxDB nevyžaduje definici schématu, TimescaleDB schéma definovat potřebuje před jakýmkoliv vkládáním dat a QuestDB umožňuje oba dva přístupy. QuestDB vyžaduje schéma, pokud data vkládáme běžným způsobem, ale pokud využijeme InfluxDB řádkový protokol, tak QuestDB, stejně jako InfluxDB, vygeneruje schéma automaticky z dat vložených. Všechny tři databáze podporují základní datové typy jako jsou například: celá čísla, desetinná čísla, řetězce, boolean, časová stopa a datum [8]. InfluxDB navíc podporuje pole, funkce a slovníky [5]. TimescaleDB navíc podporuje nejen pole, ale také geometrické objekty, síťové adresy, XML, JSON a rozsahy [7]. Jelikož je QuestDB relativně nová time-series databáze, tak oproti InfluxDB a TimescaleDB podporuje méně programátorských jazyků. Seznam podporovaných jazyků lze vidět v tabulce 2.1.

| QuestDB | InfluxDB | TimescaleDB |
|---|---|---|
| NodeJS | C# | C |
| Python | Go | C++ |
| Java | Java | Java |
| Go | JavaScript | Python |
| Rust | Kotlin | PHP |
| C | PHP | R |
| C# | Python | Ruby |
| | R | Perl |
| | NodeJS | Delphi |
| | Ruby | JavaScript |
| | Scala | Scheme |
| | Swift | Tcl |

Tabulka 2.1: Přehled podporovaných jazyků

## 2.6 Zvolení time-series databáze

Z porovnání je vidět, že QuestDB se jeví jako nejlepší kandidát na výběr z předem zmíněných time-series databází. Nevýhodou QuestDB je, že je relativně nová, tím pádem některé funkce a nástroje nejsou podporovány. Přesto ale již nabízí dostatečné možnosti pro práci s daty a chybějící funkce jsou průběžně přidávány v rámci nových verzí. QuestDB je kombinace InfluxDB a TimescaleDB z hlediska využívání InfluxDB řádkového protokolu a tradičního SQL stejně jako TimescaleDB, bez značné ztráty výkonu s rostoucí kardinalitou a s vyšší rychlostí počtu vložených řádek za sekundu.

## 2.7 Práce s daty v QuestDB

QuestDB umožňuje zapisovat a číst data pomocí různých nástrojů. Data lze zapisovat přes InfluxDB řádkový protokol, Postgres SQL dotazy, import ze CSV souborů nebo ručně skrze webovou konzoli. Zápis dat lze také provést skrze klienta, který může být napsán v několika různých jazycích, jako například: NodeJS, Java, Python či Go [9]. Data lze vypsat přes webovou konzoli, kde je můžeme i vizuálně zobrazit ve grafech, nebo přes již předem zmíněného klienta [12]. Tím pádem se klient jeví jako ideálním nástrojem pro tvorbu uživatelských vzorečků.

# 3 Výběr jazyka

## 3.1 Python

QuestDB umožnuje přístup k databázi a pracovat s daty přes klienta. Pro realizaci tohoto klienta v jazyce Python je nutný balíček `psycopg2`, který využívá objektově-relační databázový systém Postgres. Přístup k databázi přes Postgres je defaultně na portu 8812.

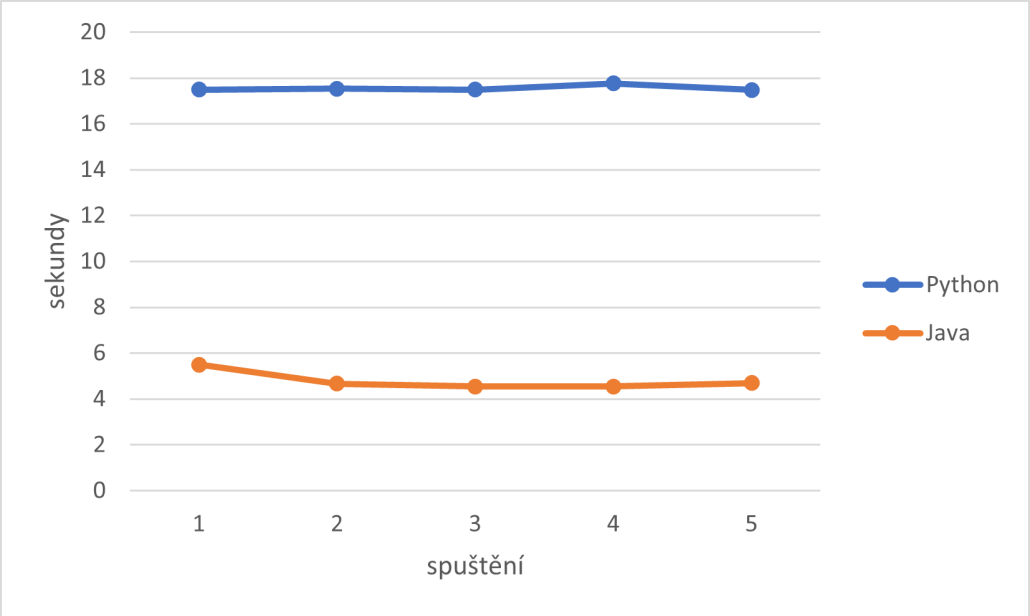
## 3.2 Java

Stejně jako Python, klient realizovaný přes jazyk Java také využívá Postgres. Java vyžaduje externí knihovnu `postgresql.jar` pro připojení k databázi a práci s daty.

## 3.3 Porovnání jazyků

Při výběru jazyka bylo hlavní rozhodovací pravidlo rychlost výběru dat z databáze a jejich opětovné zapsání. Testování rychlosti proběhlo nad tabulkou s 50 000 záznamy dvěma způsoby. Nejdříve se data vybrala z databáze, poté se přes quicksort seřadila vzestupně a následně se zpět zapsala do databáze, tento proces se několikrát opakoval ve smyčce. První způsob zapisoval každou řádku přes samostatný insert a druhý způsob zapsal všechny řádky najednou přes jediný insert. Srovnání rychlosti u prvního způsobu lze vidět na obrázku 3.1.

Testování proběhlo na počítači s Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz procesorem a 16 GB RAM paměti.




Obrázek 3.1: Srovnání rychlosti jazyků Python a Java

Při detailním výpisu času 3.1 je vidět, že hlavním důvodem rozdílu je zápis dat, který je v Pythonu výrazně pomalejší.

| | **Python** | **Java** |
|---|---|---|
| **Select** | 0.12 | 0.03 |
| **Sort** | 0.19 | 0.11 |
| **Insert** | 17.44 | 4.59 |
| **Celkem** | 17.5 | 4.68 |

Tabulka 3.1: Detailní výpis průměrného času v sekundách

U druhého způsobu jsem narazil na problém, kdy pokus zápisu všech dat jednou akcí v Pythonu trval moc dlouho a zápis skončil neúspěšně kvůli vypršení času pro jednu akci, tedy timeout. Naopak průměrný čas u Javy se značně zmenšil na cirka 0.8 s pro zápis a 1.6 s pro celý proces, jak lze vidět v tabulce 3.2.

| | **Python** | **Java** |
|---|---|---|
| **Insert** | 15+ | 0.8 |
| **Celkem** | 15+ | 1.6 |

Tabulka 3.2: Průměrný čas v sekundách pro druhý způsob

Z testování rychlosti lze tedy vidět, že Java je výrazně rychlejší a tudíž jsem jí zvolil jako jazyk pro realizaci klienta.

# 4 Struktura vzorečků a dat

## 4.1 Přístupy k uživatelským výpočtům

K rešení uživatelských výpočtů nad time-series databází se nabízí několik možností:

1. Přímá aplikace
2. Parser příkazů
3. Interpretovaný jazyk
4. Linkování kompilovaných knihoven
5. N-zpracovávajících procesů

Přímá aplikace znamená psaní vzorečků ručně přímo nad daty v databázi, například přes webovou konzoli. Parser příkazů zpracuje uživatelem zadaný vzoreček a vypíše vzorec v podobě, který je aplikovatelný pro vybraná data. Vzoreček tedy bude jednorázově zpracován a použit. Interpretovaný jazyk zpracuje uživatelem zadaný vzoreček a vygeneruje kód, kterým lze vzoreček aplikovat na daná data v databázi. Výstupem tedy bude soubor napsaný v programovacím jazyce, který je ekvivalentem zadaného vzorečku. Linkování kompilovaných knihoven využívá knihovny napsané pro každý stroj, kde při změně stroje se přelinkují knihovny, aby byly schopné pracovat s danými daty. Každé zařízení tedy bude mít vlastní knihovnu, která představuje programové zapsání vzorečků. N-zpracovávajících procesů využívá společnou knihovní funkci, která přes ostatní knihovny bude schopná pracovat se všemi specifickými daty, což představuje kombinaci obou předchozích způsobů.

Uživatelské vzorečky by měly být schopné nejen data vypsat, ale také jednotlivé výpočty zapsat zpět do databáze pro další výpočty. Tímto můžeme data sumarizovat či modifikovat. Z důvodu odlišnosti dat je nutné zvolit vhodný přístup ke způsobu ukládání dat. Jedna z možností je definovat univerzální schéma, které se bude snažit pokrýt různorodost dat, nebo data před uložením zpracovat, aby je bylo možné do schématu vložit. Druhá možnost je nechat schéma nedefinované a již uložená data modifikovat a rozšiřovat dle potřeby.

U testování zpracování uživatelských vzorečků se bude tedy nejen sledovat, zda vzorečky jsou zpracovány správně, ale také zda dokážou nalézt příslušná data. Také musí být schopné vypočtený výsledek uložit do záznamu, a ten zpět zapsat do databáze.

## 4.2 Výběr přístupu

Jak už bylo zmíněno, příchozí data se můžou výrazně lišit. Proto je nutné tabulky navrhnout tak, aby byly schopné si s odlišnostmi poradit. Jelikož se všechna nově příchozí data budou ukládat do stejné tabulky, vytvářet nový sloupec pro každou proměnnou by nebylo moc efektivní, ani přehledné. Tudíž jsem zvolil možnost univerzálního schématu.

Z důvodu velkého množství vzorečků, které se často liší jen v typu operací či obsahují jiné funkce, ale jsou zapsány ve stejném formátu a používají stejný seznam funkcí, jsem zvolil druhý přístup, tedy parser příkazů. Vytvářet nový soubor nebo knihovnu, kvůli malé změně ve vzorečku mi přijde neefektivní a zároveň to ztěžuje libovolné úpravy již existujících vzorečků. Parser nabízí větší flexibilitu pro úpravu vzorečků a díky existenci univerzálního schématu si bude schopný poradit s odlišnostmi vstupních dat.

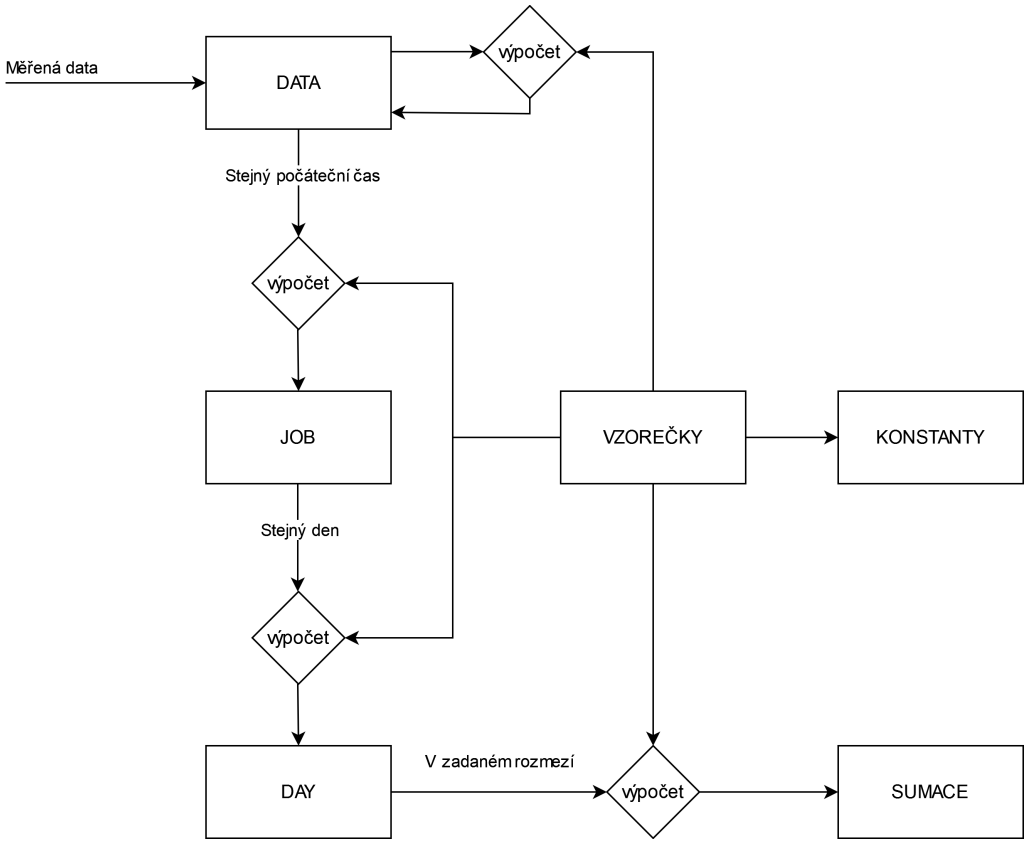
## 4.3 Úrovně výpočtů

Výpočty nad daty jsou rozděleny do čtyř úrovní, kde každá úroveň, kromě sumací, představuje jednu tabulku z databáze:

1. Data
2. Job
3. Day
4. Summary

Všechny výpočty jsou provedeny jen nad daty se stejným `id`. To platí i pro rozdělení do úrovní. Úroveň data jsou nejnižší úrovní. Představují vstupní naměřená data, která přichází do databáze. Zde jsou následně přepočtena a vložena zpět. Tato data jsou poté sdružena do další úrovně job. Job představuje jeden ucelený běh nebo operaci. Rozmezí jobu je definováno startovní časovou značkou dat, tedy pokud více dat stejného typu má stejný počáteční čas, tak patří do stejného jobu. Úroveň day představuje všechny joby

v daném dni. Takže pokud více jobů časově spadá do stejného dne, jsou sdruženy do jednoho záznamu. Poslední úroveň summary vytváří sumace z dnů za libovolné časové období. Úrověň summary slouží jen k výpisu informace, výpočty se neukládají zpět do tabulky. Ilustraci struktury lze vidět na obrázku 4.1.




Obrázek 4.1: Struktura dat a výpočtů

## 4.4 Definice schématu

Jelikož jsem zvolil možnost univerzálního schématu, tak tabulky data, job a day budou mít stejnou strukturu.

| **datetime** | **data_time** | **id** | **state** | **user** | **gps** | **recalc** | **conjoined** |
|---|---|---|---|---|---|---|---|
| timestamp | timestamp | int | int | string | string | int | string |

Tabulka 4.1: Názvy a datové typy sloupců tabulek data, job a day

Pro tabulku data 4.1 mají sloupce datetime a data_time jiný význam než pro job a day. Datetime v datech označuje čas vzniku dat a data_time

obsahuje čas spuštění čili čas od začátku operace. Pro job a day tabulku datetime a data_time představují častový rozsah, tedy startovní a konečný čas. Další sloupce jako id slouží k identifikaci zařízení, state udává stav zařízení, user obsahuje název uživatele a gps představuje gps souřadnice. Recalc je pomocný sloupec, který udává zda záznam je nutný přepočítat nebo zda je už ve finálním stavu. Sloupec conjoined je sloučením všech ostatních hodnot, kterých by data mohla nabývat. Tímto způsobem můžeme odlišná data sjednotit do stejné podoby. Formát sloupce conjoined se skládá z názvu proměnné následované znakem rovno a dané hodnoty. Všechny proměnné jsou oddělené středníkem.

Pro realizaci výpočtů je nutné ještě definovat tabulky konstant 4.2 a vzorečků 4.3. Konstanty jsou uloženy v tabulce constants a vzorečky jsou v tabulce equations.

| **id** | **name** | **value** |
|---|---|---|
| int | string | string |

Tabulka 4.2: Názvy a datové typy sloupců tabulky constants

Sloupec id slouží k přiřazení konstanty k zařízení, name udává jméno konstanty a value obsahuje hodnotu dané konstanty.

| **type** | **id** | **name** | **eq** |
|---|---|---|---|
| int | int | string | string |

Tabulka 4.3: Názvy a datové typy sloupců tabulky equations

Sloupec type představuje na jaké úrovni výpočtů bude vzoreček použit. Type je rozdělen do úrovní:

- 1 = měřená, tedy nově příchozí data
- 2 = počítaná data
- 4 = job
- 5 = day
- 6 a 7 = summace

Další sloupce jako id opět slouží k přiřazení vzorečku k zařízení, name obsahuje název vzorečku, čili pod jakým názvem se proměnná uloží a eq je sloupec obsahující samotný vzoreček.

## 4.5 Syntaxe vzorečků

Vzorečky jsou napsané v prefixové čili polské notaci. To znamená, že operátory jsou zapsané vlevo před operandy. Například výraz `( 1 + 2 )` by tedy byl zapsán jako `( + 1 2 )`. Ve vzorcích se vyskytují výrazy:

- číslo, desetinná čísla jsou oddělena tečkou
- řetězec znaků, řetězce jsou uvozeny a uzavřeny apostrofy
- funkce, zapsané názvem funkce
- proměnná či kanál, uvozené znakem dolaru `$`
- konstanta, zapsané názvem a uvozeny křížkem `#`

Proměnné jsou navíc dále rozděleny podle úrovně vzorečku, ve kterém jsou obsaženy, počtem uvozujících dolarů a zda je název zapsán velkými či malými písmeny. Rozdělení lze vidět v tabulce 4.4.

| | **data** | **job** | **day** | **summary** |
|---|---|---|---|---|
| **$kanal** | aktuální zrovna počítané dato | pole dat aktuálního jobu | pole jobů v aktuálním dni | pole denních dat v počítaném rozsahu |
| **$$kanal** | předchozí dato | pole dat předchozího jobu | pole jobů v předchozím dni | denní výsledek před počítaným rozsahem |
| **$KANAL** | aktuální job | aktuální zrovna počítané jobové dato | aktuální zrovna počítané denní dato | aktuální zrovna počítané sumační dato |
| **$$KANAL** | předchozí job | výsledek předchozího jobu | výsledek předchozího dne | |
| **$d$kanal** | | | pole dat v aktuálním dni | |
| **$j$kanal** | | | | pole jobů v počítaném rozsahu |

Tabulka 4.4: Rozdělení významu proměnných

Vzoreček, tedy může vypdadat například takto:

```
IF > $tmp #TMAX 1 0
```

Tento vzoreček sleduje, zda hodnota kanálu `$tmp`, tedy v tomto případě teplota aktuálního data, je vyšší než než konstanta `#TMAX`. Pokud je podmínka splněna, výsledkem je 1, jinak 0.

## 4.6 Popis funkcí a pseudokonstant

Jelikož se většina vzorečků skládá z pojmenovaných funkcí, je nutné funkce definovat a ručně implementovat. Pseudokonstanty představují speciální konstanty, které jsou dostupné pro všechna odlišná data a zařízení. Význam pseudokonstant se také liší na aktuální úrovni výpočtu.

Funkce jsou rozděleny do tří typů podle vstupu a výstupu:

- Skalární funkce - vstup číslo nebo řetězec, výstup číslo nebo řetězec
- Sumační funkce - vstup vektor, výstup číslo nebo řetězec
- Vektorové funkce - vstup vektor, výstup vektor

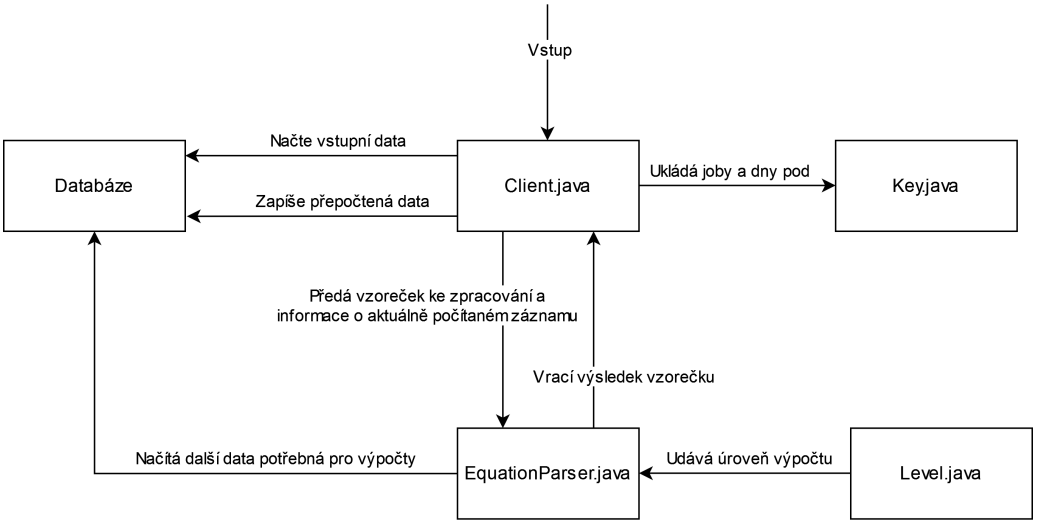
Seznam implementovaných pseudokonstant a funkcí lze nalézt v sekci Přílohy.

# 5 Implementace klienta

Jako řešení přístupu k uživatelským výpočtům jsem zvolil klienta napsaného v javě. Jelikož schéma je definované tak, že vzorečky lze ukládat přímo do databáze, tak parser vzorečků bude zpracovávat vzorečky podle aktuálně počítaného data, kde z databáze načte příslušný vzoreček, který rovnou zpracuje a výsledek zapíše zpět do databáze. Výsledný program se skládá ze čtyř tříd:

- Client.java
- EquationParser.java
- Key.java
- Level.java

Diagram popisující vazby mezi třídami a databází lze vidět na obrázku 5.1.




Obrázek 5.1: Vazby mezi třídami a databází

## 5.1 Client.java

Hlavní třída programu, která představuje vstupní bod, zda se budou počítat data nebo sumace určuje, jestli se program spustil bez argumentů nebo s dvěma argumenty. Pokud bez argumentů, tak se nejdříve načtou data z

databáze, která jsou označena pro přepočet, tedy ve sloupci `recalc` mají hodnotu vyšší než 0. Pokud jsou načteny i joby nebo dny, jsou načtena i data, ze kterých se počítala. Poté se data přepočtou a rozdělí do jobů. Vypočtené joby se následně rozdělí do dní pro finální výpočet. Tento proces se opakuje ve smyčce, dokud se nedojde do bodu, kdy je vše spočteno nebo se už více s aktuálními daty spočítat nedá. Přepočtená data jsou na závěr vložena zpět do databáze. Při zadání dvou argumentů, které musí být ve formátu `yyyy-MM-dd`, se spočtou sumace v zadaném rozsahu. Výsledek sumace je na konci vypsán. Všechny výsledky jsou spočteny skrze třídu `EquationParser.java`. Tato třída také ukládá všechna dodatečná data, která se načtou během výpočtů pro opakované použití. Všechna počítaná data také mají pomocný seznam, který určuje zda daný záznam je už ve fínální podobě nebo je ho třeba znovu přepočítat.

Třída obsahuje následující proměnné:

**`EquationParser equationParser`** - parser vzorečků.

**`Connection connection`** - spojení s databází.

**`static List<HashMap<String, Object>> new_data`** - nová data načtená z databáze.

**`static List<HashMap<String, Object>> recalcedData`** - přepočtená dat.

**`static HashMap<Key, List<HashMap<String, Object>>> jobs`** - přepočtená data rozdělená do jobů.

**`static HashMap<Key, HashMap<String, Object>> recalcedJobs`** - přepočtené joby.

**`static HashMap<Key, List<HashMap<String, Object>>> daysFromJobs`** - přepočtené joby rozděleny do dnů.

**`static HashMap<Key, List<HashMap<String, Object>>> daysFromData`** - přepočtená data rozdělena do dnů.

**`static HashMap<Key, HashMap<String, Object>> recalcedDays`** - přepočtené dny.

**static HashMap<Integer, List<HashMap<String, Object>>> summs** - dny v zadaném rozsahu sumace.

**static HashMap<Integer, HashMap<String, Object>> calcedSumms** - vypočtené sumace.

**static List<Boolean> new_data_finished** - pomocný list, který udává zda je dané dato nutné přepočítávat.

**static HashMap<Key, Boolean> jobs_finished** - pomocná hashmapa, která udává zda je daný job nutné přepočítat.

**static HashMap<Key, Boolean> daysFromJobs_finished** - pomocná hashmapa, která udává zda je daný day nutné přepočítat.

**static HashMap<Integer, Boolean> summ_finished** - pomocná hashmapa, která udává zda je danou sumaci nutné přepočítat.

**static HashMap<Key, HashMap<String, Object>> jobsFromDB** - uchovává joby, které bylo nutné během výpočtů dodatečně načíst z databáze.

**static HashMap<Key, List<HashMap<String, Object>>> jobDataFromDB** - uchovává data jobů, které bylo nutné během výpočtů dodatečně načíst z databáze.

**static HashMap<Key, HashMap<String, Object>> daysFromDB** - uchovává dny, které bylo nutné během výpočtů dodatečně načíst z databáze.

**static HashMap<Key, List<HashMap<String, Object>>> dayDataFromDB** - uchovává data dnů, které bylo nutné během výpočtů dodatečně načíst z databáze.

**static HashMap<Integer, List<HashMap<String, Object>>> jobsFromDB_SUM** - uchovává data jobů, které bylo nutné během výpočtů dodatečně načíst z databáze. Rozděleny podle id pro sumace.

**static HashMap<Integer, HashMap<String, Object>> daysFromDB_SUM** - uchovává data jobů, které bylo nutné během výpočtů dodatečně načíst z databáze. Rozděleny podle id pro sumace.

**static HashMap<String, HashMap<String, Object>> constants** - uchovává konstanty načtené z databáze.

**HashMap<Integer, List<HashMap<String, Object>>> equations** - uchovává vzorečky načtené z databáze.

**static int recalcsFailedCount** - počítadlo. Udává kolik proměnných se nepodařilo vypočítat během jednoho cyklu.

**static Timestamp startStamp** - startovní rozsah pro sumace.

**static Timestamp stopStamp** - konečný rozsah pro sumace.

**private final int dataLimit** - udává maximální počet řádek dat, kterých bude načteno z databáze.

**private final int jobLimit** - udává maximální počet řádek jobů, kterých bude načteno z databáze.

**private final int dayLimit** - udává maximální počet řádek dní, kterých bude načteno z databáze.

A následující metody:

**void connect()** - metoda, která je první volána při spuštění programu. Vytvoří spojení s databází.

**void closeConnection()** - poslední metoda, která je v programu volána. Ukončí spojení s databází.

**void selectData()** - Načte všechna relevantní data z databáze. To zna-

mená všechna data, joby a dny které mají ve sloupci recalc hodnotu vyšší než 0 a označí je nutné pro přepočet v pomocných proměnných. Maximální počet načtených záznamů udávají výše zmíněné proměnné. Pokud je nutné přepočítat job nebo den, z databáze se načtou data, z kterých výpočet předtím probíhal a v pomocných proměnných jsou označeny za spočtené. Pokud tato data nejsou nalezena tak job/den nelze přepočítat, tudíž program skončí a informuje uživatele.

**`void selectDaysBetween(Timestamp start, Timestamp end)`** - načte dny do `new_data` v zadaném rozsahu pro sumace.

**`List<HashMap<String, Object>> getEquations(int id)`** - načte z databáze vzorečky pro zadané id, uloží je do `equations`, kde klíč je dané id a vrátí je v listu.

**`void recalcData()`** - přepočte data v `new_data` přes `equationParser` a uloží je do `recalcedData`. Pokud je daný záznam označen v `new_data_finished` za hotový, přeskočí se.

**`void recalcJobs()`** - přepočte joby v `jobs` přes `equationParser` a uloží je do `recalcedJobs`, kde klíč je stejný klíč jako v `jobs`. Pokud je daný záznam označen v `jobs_finished` za hotový, přeskočí se.

**`void recalcDays()`** - přepočte dny v `daysFromJobs` přes `equationParser` a uloží je do `recalcedDays`, kde klíč je stejný klíč jako v `daysFromJobs`. Pokud je daný záznam označen v `daysFromJobs_finished` za hotový, přeskočí se.

**`void calcSumTot()`** - vypočte sumaci z `summs` přes `equationParser` a uloží je do `calcedSumms`, kde klíč je id. Pokud je daný záznam označen v `summ_finished` za hotový, přeskočí se.

**`void getJobs()`** - přepočtená data z `recalcedData` rozdělí do jobů podle sloupce data_time, tedy začátek jobu. Rozdělená data uloží do `jobs`, kde klíč je id a data_time.

**`void getDays()`** - přepočtené joby z `recalcedJobs` rozdělí do dnů podle toho zda joby proběhly ve stejný den. Každé zařízení má jinak definovaný zlom dne, tuto hodnotu zjistíme z konstanty `#WB`. Rozdělené joby uloží do `recalcedData`, kde zdrojová data jsou z `recalcedJobs` a do `recalcedDays`,

kde zdojová data jsou z `jobs`. Klíč je stejný pro obě a je to stejný klíč jako v `recalcedJobs`.

**`void categorizeById()`** - nově načtené záznamy z `new_data` rozdělí podle id a uloží do `summs`, kde klíč je dané id. Použité v sumacích.

**`void insertToDB()`** - zapíše všechna vypočítaná data z `recalcedData`, `recalcedJobs` a `recalcDays` zpět do příslušných tabulek v databázi.

**`static void main(String[] args)`** - vstupem jsou dva nepovinné argumenty. Pokud nejsou zadány, tedy program je spuštěn bez argumenů, proběhne přepočet dat, kde se postupně volají metody `selectData()`, `recalcData()`, `getJobs()`, `recalcJobs()`, `getDays()` a `recalcDays()`. Poté se ve smyčce tyto metody opakují dokud `recalcsFailedCount` není nula nebo není stejný jako v předešlé iteraci, tedy nic nového se už spočítat nemůže. Ve smyčce se `selectData()` znovu nevolá, `new_data` jsou přepočtená data z předchozí iterace. Po dokončení smyčky se data zapíší zpět do databáze a program končí. Pokud jsou oba argumenty zadány a oba argumenty jsou ve formátu `yyyy-MM-dd`, tak se načtou dny ze zadaného rozmezí a pro každé id se vypočtou sumace, které na konci budou vypsány.

## 5.2 EquationParser.java

Tato třída slouží ke zpracování vzorečků a jejich aplikaci na načtené záznamy. Od třídy `Client.java` dostane vzoreček, informace o aktuálně počítaném záznamu a na jaké úrovni má výpočet proběhnout. Zadaný vzoreček se zpracovává rekurzivně a hodnoty proměnných se zjišťují podle aktuální úrovně výpočtu. Pokud jsou nutná další data pro provedení výpočtu, tak jsou načtena z databáze a uložena ve třídě `Client.java`. Vypočtený výsledek se aplikuje na daný záznam, kde se uloží pod jménem vzorečku a je ukončen středníkem. Pokud nelze vzoreček spočítat, výsledek je prázdný řetězec.

Obsahuje následující proměnné:

**`final Connection connection`** - připojení do databáze.

**`Object currRow`** - aktuální řádek. Může být číslo, pokud počítáme záznamy v listech nebo může být klíč Key pokud počítáme záznamy uložené v hashmapě.

**`Level currLevel`** - udává aktuální úroveň výpočtu.

A následující metody:

**`EquationParser(Connection connection)`** - konstruktor, inicializuje `connection` na předanou hodnotu. **`String applyEquation(String conjoined, HashMap<String, Object> eq, Object row, Level level)`** - vstupem je sloupec `conjoined` aktuálně počítaného záznamu, vzoreček, který chceme zpracovat, aktuální řádek a úroveň výpočtu. Nejdříve se nastaví `currRow` a `currLevel` na předané hodnoty. Poté se z předaného vzorečku načte sloupec `eq`, který obsahuje samotný vzoreček, a rozdělí se do částí, kde oddělovačem je mezera. Rozdělený vzoreček se předá metodě `parseFunction(String[] eq)`, která vrátí vypočtenou hodnotu. Pokud je vrácená hodnota prázdný řetězec nebo chybného tvaru, zvedne se o jedna počítadlo `recalcsFailedCount` ze třídy Client. Následně se z předaného vzorečku načte sloupec `name`, pod kterým se do předaného conjoined proměnná s vypočtenou hodnotou uloží. Pokud jsou všechny proměnné v conjoined spočteny, označí se řádek za spočtený v pomocných proměnných ve třídě Client, tedy `new_data_finished`, `jobs_finished` atd. Zápis do proměnné je určen podle aktuální úrovně. Metoda vrací přepočtený conjoined.

**`String parseFunction(String[] eq)`** - zpracuje zadaný vzoreček. Jelikož je vzoreček napsán v prefixové notaci je zpracován po částech pomocí rekurze, kde každá operace má zadaný počet parametrů. Každý parametr představuje jednu pozici od operátoru doprava. Tato metoda obsahuje switch na všechny implementované pojmenované funkce. Pokud aktuálně zpracovávaná část není funkcí, zavolá se příslušná metoda `getvariables` podle aktuální úrovně. Metoda vrací hodnotu vypočtenou ze vzorečku nebo prázdný řetězec pokud výpočet selhal.

**`String getVariablesForData(String eq)`** - vstupem je buď název proměnné nebo konstanty. Metoda nahradí název získanou hodnotou. Místo, odkud se hodnota zjistí, závisí na formátu vstupu, který lze vidět v tabulce 4.4. Metoda je volána jen na úrovni dat.

**`String getVariablesForJobs(String eq)`** - vstupem je buď název proměnné nebo konstanty. Metoda nahradí název získanou hodnotou. Místo, odkud se hodnota zjistí, závisí na formátu vstupu, který lze vidět v tabulce 4.4. Metoda je volána jen na úrovni jobů.

**`String getVariablesForDays(String eq)`** - vstupem je buď název proměnné nebo konstanty. Metoda nahradí název získanou hodnotou. Místo, odkud se hodnota zjistí, závisí na formátu vstupu, který lze vidět v tabulce 4.4. Metoda je volána jen na úrovni dnů.

**`String getVariablesForSumTot(String eq)`** - vstupem je buď název proměnné nebo konstanty. Metoda nahradí název získanou hodnotou. Místo, odkud se hodnota zjistí, závisí na formátu vstupu, který lze vidět v tabulce 4.4. Metoda je volána jen na úrovni sumací.

**`String getConstantsForData(String eq)`** - kontroluje zda zadaná konstanta nepatří do seznamu pseudokonstant na úrovni dat. Pokud ano, vrátí příslušnou hodnotu, jinak předá konstantu metodě `getTableConstants`.

**`String getConstantsForJobs(String eq)`** - kontroluje zda zadaná konstanta nepatří do seznamu pseudokonstant na úrovni jobů. Pokud ano, vrátí příslušnou hodnotu, jinak předá konstantu metodě `getTableConstants`.

**`String getConstantsForDays(String eq)`** - kontroluje zda zadaná konstanta nepatří do seznamu pseudokonstant na úrovni dnů. Pokud ano, vrátí příslušnou hodnotu, jinak předá konstantu metodě `getTableConstants`.

**`String getConstantsForSumTot(String eq)`** - kontroluje zda zadaná konstanta nepatří do seznamu pseudokonstant na úrovni sumací. Pokud ano, vrátí příslušnou hodnotu, jinak předá konstantu metodě `getTableConstants`.

**`HashMap<String, Object> getPreviousJob(int id, Key curr)`** - pokusí se nalézt job, který proběhl před zadaným jobem. Nejprve prohledá lokálně uložené joby, pokud zde žádný nenajde, pokusí se najít v databázi. Načtený job z databáze uloží do `jobsFromDB` ze třídy Client.

**`List<HashMap<String, Object>> getPreviousJobArray(int id, Key curr)`** - pokusí se nalézt data, ze kterých se počítal předchozí job, který proběhl před zadaným jobem. Nejprve prohledá lokálně uložené joby, pokud zde žádný nenajde, pokusí se najít v databázi. Načtená data z databáze uloží do `jobDataFromDB` ze třídy Client.

**`List<HashMap<String, Object>> getJobsInRange()`** - vrátí joby, které spadají do zadaného rozmezí sumací. Nejprve prohledá lokálně uložené joby,

pokud je zde nenajde, pokusí se je najít v databázi. Načtené joby z databáze uloží do `jobsFromDB_SUM` ze třídy Client.

**`HashMap<String, Object> getPreviousDay()`** - pokusí se nalézt den, který proběhl před aktuálním dnem. Nejprve prohledá lokálně uložené dny, pokud zde žádný nenajde, pokusí se najít v databázi. Načtený den z databáze uloží do `daysFromDB` ze třídy Client.

**`List<HashMap<String, Object>> getPreviousDayArray()`** - pokusí se nalézt joby, ze kterých se počítal předchozí den, který proběhl před aktuálním dnem. Nejprve prohledá lokálně uložené dny, pokud zde žádný nenajde, pokusí se najít v databázi. Načtené joby z databáze uloží do `daysFromJobs` ze třídy Client.

**`HashMap<String, Object> getDayBeforeRange()`** - pokusí se nalézt den, který proběhl před zadaným rozsahem pro sumace. Nejprve prohledá lokálně uložené dny, pokud zde žádný nenajde, pokusí se najít v databázi. Načtený den z databáze uloží do `daysFromDB_SUM` ze třídy Client.

**`String getTableConstants(String eq, String id)`** - zpracuje zadaný název konstanty a předá metodě `getConstant`. Vrácenou hodnotou nahradí název a výsledek vrátí.

**`HashMap<String, Object> getConstant(String id, String name)`** - pokusí se nalézt konstantu podle zadaného jména a `id`. Nejprve prohledá lokálně uložené konstanty, pokud jí zde nenajde, pokusí se najít v databázi. Načtenou konstantu z databáze uloží do `constants` ze třídy Client.

**`int checkForBrackets(String[] eq, int pos)`** - pomocná metoda, která kontroluje, zda na zadané pozici je levá závorka. Pokud ano, najde pozici odpovídající pravé závorky a pozici vrátí.

**`List<Double> numberListFromArrayString(String string)`** - pomocná metoda, která ze zadaného řetězce vytvoří list čísel. Čísla v řetězci musí byt oddělena čárkou.

**`List<String> stringListFromArrayString(String string)`** - pomocná metoda, která ze zadaného řetězce vytvoří list řetězců. Řetězce odděluje podle čárky.

`String doubleListToString(List<Double> list)` - pomocná metoda, která převede list desetinných čísel na řetězec. Hodnoty jsou v řetězci odděleny čárkou.

`String intListToString(List<Integer> list)` - pomocná metoda, která převede list celých čísel na řetězec. Hodnoty jsou v řetězci odděleny čárkou.

`String removeLeadingAndEndingFromVariable(String var)` - pomocná metoda, která ze zadaného řetězce odstraní znaky: `=, _` a `;`.

`boolean isNumeric(String str)` - pomocná metoda, která určí zda zadaný řetězec je číslo nebo ne.

`boolean isAllUpperCase(final String s)` - pomocná metoda, která určí zda se zadaný řetězec skládá jen z velkých písmen. Metoda ignoruje čísla, tedy pokud zadaný řetězec obsahuje jen velká písmena a číslice, vrací `true`.

`List<Double> parseHistogramAxis(String input)` - pomocná metoda, která ze zadaného řetězce vytvoří list hodnot osy histogramu. zadaný řetězec musí být v jednom ze dvou formátů: čísla oddělené čárkou nebo hodnoty oddělené čárkou, kde hodnoty jsou minimum a maximum rozmezí oddělené dvojtečkou. Pokud zadaný řetězec neobsahuje dvojtečku, minima a maxima rozsahu se vypočtou ze zadaných čísel.

`String checkVariableName(String variable)` - pomocná metoda, která kontroluje, zda zadané jméno proměnné neodkazuje na sloupec tabulky. Pokud ano, vrátí název sloupce, jinak vrátí název sloupce `conjoined`.

## 5.3 Key.java

Pomocná třída, která představuje klíč, pod kterým se ukládají záznamy do jobů a dnů. Klíč se skládá z počátečního času a textttid zařízení.

Obsahuje dvě proměnné:

`final Timestamp data_time` - časová značka.

`final int id` - id zařízení.

A následující metody:

**`Key(Timestamp x, int y)`** - konstruktor, inicializuje proměnné `data_time` a `id` na zadané hodnoty.

**`boolean before(Key key)`** - pokud klíč v parametru má stejné id a jeho časová značka je po čase než je čas v klíči, na kterém byla metoda zavolána, vrací true, jinak false.

**`boolean after(Key key)`** - pokud klíč v parametru má stejné id a jeho časová značka je před časem než je čas v klíči, na kterém byla metoda zavolána vrací true, jinak false.

**`String toString()`** - přepisuje nadřazenou metodu. Vypíše `data_time` a `id` oddělené dvojtečkou.

**`boolean equals(Object o)`** - přepisuje nadřazenou metodu. Pokud má parametr stejnou časovou značku `data_time` a `id` vrací true, jinak false.

**`int hashCode()`** - jelikož přepisujeme metodu `equals`, tak musíme přepsat i tuto metodu, která vypočítává hash, pod kterým se klíč ukládá.

**`Timestamp getData_time()`** - vrátí `data_time`.

**`int getId()`** - vrátí `id`.

## 5.4 Level.java

Pomocná enum třída, která udává úroveň výpočtu. Může nabývat hodnot:

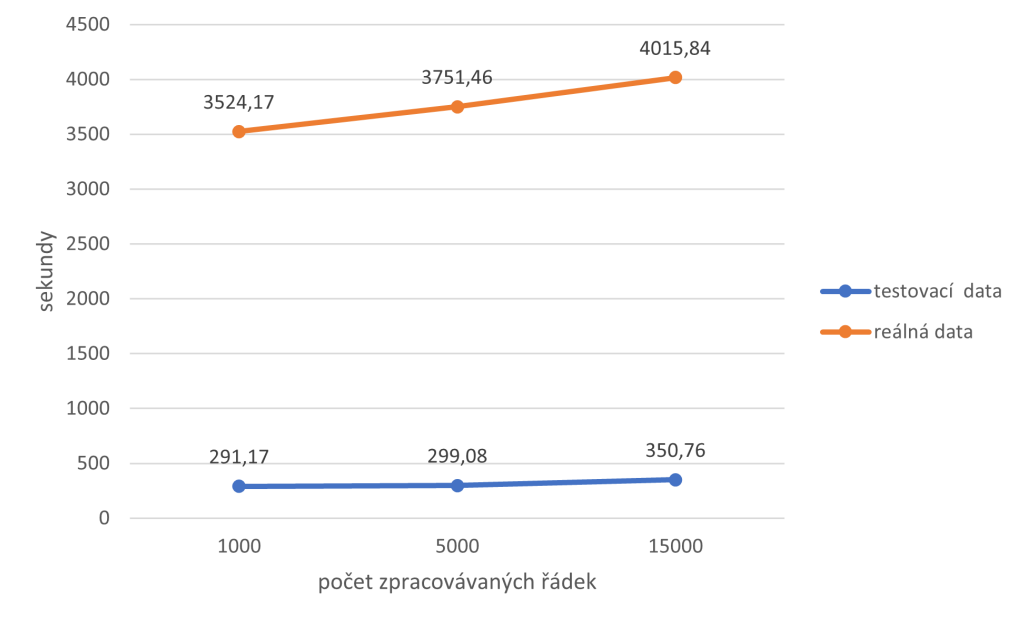
- DATA
- JOB
- DAY
- SUM

Využito ve třídě textttEquationParser.java, která podle zadané úrovně zpracovává vzorečky příslušným způsobem.

# 6 Testování

Důležitou součástí práce bylo také testování. Testování probíhalo nad triviálními, tedy testovacími daty, a nad daty reálnými. Měřen byl celý běh aplikace nad různým počtem zpracovávaných řádků a nad odlišnými typy dat. Testovací data obsahují jednodušší a menší počet vzorečků. Tato data jsou navržena tak, že vždy lze spočítat všechny hodnoty a program končí ve stavu, kdy je vše spočteno. Oproti tomu reálná data obsahují více složité vzorečky a obsahují funkce, které nelze s dostupnými daty spočítat. Program tedy skončí ve stavu, kde další iterace už více spočítat nemohou, ale vše spočteno není. Reálná data také obsahují mnohem více proměnných a počítají se nad vetším počtem vzorečků. Reálná data, která byla použita pro toto testování, obsahují stonásobek vzorečků a přibližně třicetinásobek záznamů v tabulkách oproti datům testovacím.

Testování proběhlo na počítači s Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz procesorem a 16 GB RAM paměti.

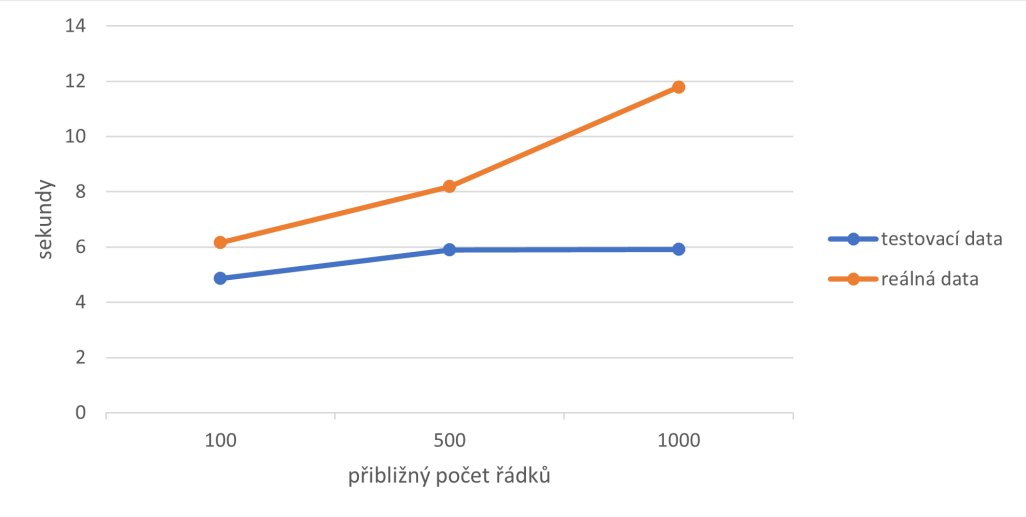



Obrázek 6.1: Rychlost výpočtů nad testovacími a reálnými daty

Z grafu 6.1 vidíme, že rozdíl mezi reálnými a testovacími daty je výrazný, a že i přes velký nárůst počtu zpracovávaných řádků se výsledný čas příliš neliší. Hlavním důvodem je, že většinu výsledného času program stráví za-

pisováním dat zpět do databáze. U reálných dat, kde běh průměrně dosáhl jedné hodiny, je okolo padesáti minut stráveno na zápisu dat. Pro testovací data zápis představuje zhruba polovinu či více celkového času. To, že i přes rozdílný počet zpracovávaných řádků jsou výsledné časy velmi stejné je způsobeno tím, že finální čas více záležel na počtu dat v databázi než na počtu zpracovávaných řádků. Příčinou je, že QuestDB zatím nepodporuje SQL příkazy UPDATE a DELETE. Tudíž pro zápis dat je nutné dané tabulky vytvořit znovu bez dat, která počítáme a až poté se mohou vložit vypočtená data. Tento proces vybrání všech dat, kromě dat počítaných, přes příkaz SELECT je velice náročný, obzvlášť pokud máme veliký počet složitých dat. Tedy pro zápis dat je nutné provést příkaz SELECT nad všemi daty v databázi, což není použitelné řešení pro velký počet záznamů, tedy pro data reálná.

Fakt, že program stráví většinu času u zápisu dat, můžeme ověřit testováním rychlosti u sumací, kde se spočtený výsledek nikam neukládá, pouze se vypíše.




Obrázek 6.2: Rychlost sumací nad testovacími a reálnými daty

V grafu 6.2 lze vidět, že výsledné časy jsou mnohokrát kratší a rozdíl mezi testovacími daty a daty reálnými není příliš veliký. Počet řádků je pouze průměrný, protože stejné časové rozpětí obsahovalo odlišný počet řádků, podle toho, na kterých datech výpočet probíhal. Časová rozmezí však byla zvolena tak, aby počet řádků byl přibližně stejný. Důvodem rozdílného počtu zpracovávaných řádků v obrázku 6.1 a 6.2 je, že obrázek 6.1 představuje počet datových řádků a obrázek 6.2 představuje počet denních řádků. Data

použitá v 6.1 a 6.2 jsou ale stále stejná, výpočet jen proběhl na jiné úrovni. Denních řádků je zpravidla méně než datových a jeden denní řádek může odkazovat na několik jobů nebo dat, tedy finální počet řádků použitých při výpočtu může být vyšší. Počet zpracovaných řádků záleží na typu použitých vzorečků.

Ověření, že výsledný čas více závisí na počtu záznamů v tabulce, a ne na počtu záznamů ve výpočtu lze vidět v tabulce 6.1.

| | **tabulka s 6000 záznamy** | **tabulka s 18000 záznamy** |
|---|---|---|
| **Select** | 12.89 | 9.62 |
| **Výpočet** | 4.06 | 3.72 |
| **Insert** | 133.95 | 1068.40 |
| **Celkem** | 151.23 | 1082.04 |

Tabulka 6.1: Čas strávený v jednotlivých sekcí programu v sekundách

Obě měření proběhla nad daty testovacímy, kde bylo vybráno 5000 záznamů pro výpočet. Z tabulky 6.1 tedy lze vidět, že i pres stejný počet záznamů, kterých bylo použito pro výpočet, se výsledný čas výrazně liší. S vyšším počtem dat v tabulkách tedy čas pro zápis dat extrémě narůstá.

Důvod, proč zápis dat a tedy tvorba nových tabulek je tak časově náročná je, jak už bylo zmíněno, že se skládá z velice náročného zápisu příkazu SELECT. Při velkém počtu netriviálních dat čas pro vykonání příkazu značně narůstá. Při testování rychlosti běžných výpočtů bez provedení finálního kroku, tedy zápisu, byla finální rychlost značně menší, pro reálná data se dokonce čas přibližně srovnal s daty testovacími. Ale stále, pokud byl zvolen veliký počet dat k přepočítání, obzvláště pří větším počtu jobů nebo dní k přepočítání, tak nejdelší část běhu program stráví u prvního kroku, kdy načítá data z databáze.

Bude tedy nutné počkat až vyjde přidání podpory SQL příkazů UPADTE a DELETE. Naštěstí vydání podpory je plánované na příštích pár měsíců. Po vydání podpory stačí přepsat metodu `insertToDB()`. Důvodem vybrání QuestDB databáze i přes tento nedostatek je, že zprvu vydání podpory bylo plánováno dříve, ale nakonec se datum vydání posunulo. Tímto jsem byl nucen implementovat dočasné řešení nebo si vybrat jinou databázi. Z důvodu časového omezení jsem zvolil implementaci dočasného řešení. To se ale nakonec ukázalo být více neefektivní, než jsem čekal. S vydáním této podpory tedy bude program použitelný i pro vyšší počet dat a pro reálná data, jelikož

je zápis dat hlavním důvodem dlouhých časů běhu programu.

Testování, zda program vzorečky zpracovává správně, proběhlo odlišně podle typu dat. Pro data testovací se očekávaný výsledek dá ručně spočítat a pak porovnat s výsledky programu. Pro data reálná byl použit způsob, kde byl porovnán výsledek programu a výsledek již předem spočítaných reálných dat, tedy pokud se data rovnala, tak výsledek programu byl správný.

Pokud program nedokáže vzoreček zpracovat, nebo obsahuje neimplementované funkce či je chybně zapsán, tak třída `EquationParser` vždy vrací prázdný řetězec. Pokud ale jsou načteny joby nebo dny pro přepočet a v databázi se nepodaří najít data, z jakých se počítala, tak program skončí v prvím kroku a informuje uživatele. Jestliže během běhu programu dojde ke zrátě připojení k databázi z jakéhokoliv důvodu, tak program také skončí, jelikož bez databáze nemůže fungovat.

Systémové nároky programu a datáze jsou hlavně ve velikosti souborů na disku a paměťové náročnosti. Tabulky databáze zabírají na disku průměrně 500 MB pro 5000 záznamů a až 11 GB pro 500 000 záznamů. Jelikož je nutné data načíst z databáze pro výpočet, tak program může být i paměťově náročný. Z testování vyšlo, že náročnost na pamět se pohybuje od 260 MB pro 5000 záznamů do 3 GB pro 100 000 záznamů. Při pokusu zpracování více dat hrozí, že náročnost přesáhne maximální hranici, která u JVM většinou bývá nastavena na 4 GB. Pro navýšení tohoto maxima je nutné program spustit s argumentem `-Xmx<size>`, tedy například `-Xmx6g` pro zvednutí maximální hranice na 6 GB.

# 7 Budoucí vylepšení

S vydáním podpory SQL příkazů UPDATE a DELETE bude nejen program schopný pracovat s daty reálnými, ale také bude možné upravovat uložené vzorečky dle potřeby, nebo ruční označování dat pro přepočet tím, že se změní hodnota v `recalc` sloupci tabulek.

Další možné rozšíření programu je implementace nových funkcí dle potřeby. Pro implementaci nových vzorečků stačí vzoreček vložit do databáze. Pro zlepšení celkové doby běhu programu se nabízí co nejvíce optimalizovat příkazy SELECT, jelikož jsou nejnáročnější částí programu.

# 8 Závěr

Cílem bakalářské práce bylo vytvořit program, který bude schopný aplikovat vzorečky na odlišná data, spočtený výsledek poté zapsat zpět do databáze, a nebo spočítat a vypsat sumace. Nejprve bylo nutné vybrat si v jaké time-series databázi se bude pracovat, poté bylo potřeba zvolit v jakém jazyce bude program napsán a nakonec otestovat program nad reálnými daty.

Výsledkem je klient pro QuestDB time-series databázi, který je napsaný v jazyce Java. Kvůli opoždění vydání podpory SQL příkazů UPDATE a DELETE je výsledný program použitelný pouze nad testovacími daty. Po vydání této podpory a přepsání metody pro zápis dat zpět do databáze, bude ale program schopný pracovat i s daty reálnými.

# Přehled zkratek

- IoT – Internet of Things - síť fyzických zařízení
- SQL – Structured Query Language - jazyk pro komunikaci s databázi
- RAM – Random-Access Memory - paměť s přímým přístupem
- TICK – Telegraf, InfluxDB, Chronograf and Kapacitor - typ time-series databázové architektury
- SIMD – Single Instruction, Multiple Data - typ počítačové architektury
- CPU – Central processing unit - Centrální procesorová jednotka v počítači
- XML – Extensible Markup Language - značkovací jazyk
- CSV – Comma-separated values - souborový formát
- JSON – JavaScript Object Notation - objekt programovacího jazyku JavaScript
- JVM – Java Virtual Machine - virtuální stroj programovacího jazyku Java

# Literatura


[1] Collard, T. *Aggregating billions of rows per second with SIMD* [online]. questdb, 2021. [cit. 2021/12/30]. Dostupné z: https://questdb.io/blog/2020/04/02/using-simd-to-aggregate-billions-of-rows-per-second/.

[2] Hwang, Y. *Comparing InfluxDB, TimescaleDB, and QuestDB Timeseries Databases* [online]. towardsdatascience, 2021. [cit. 2021/11/20]. Dostupné z: https://towardsdatascience.com/comparing-influxdb-timescaledb-and-questdb-timeseries-databases-c1692b9327a5.

[3] Ilyushchenko, V. *How databases handle 10 million devices in high-cardinality benchmarks* [online]. questdb, 2021. [cit. 2021/12/30]. Dostupné z: https://questdb.io/blog/2021/06/16/high-cardinality-time-series-data-performance.

[4] InfluxDB. *InfluxDB Documentation* [online]. influxdata, 2021. [cit. 2021/12/30]. Dostupné z: https://docs.influxdata.com/influxdb/cloud/reference/key-concepts/data-elements/.

[5] InfluxDB. *InfluxDB Documentation* [online]. influxdata, 2021. [cit. 2022/1/9]. Dostupné z: https://docs.influxdata.com/flux/v0.x/spec/types/.

[6] Mike Freedman, A. S. *TimescaleDB vs. InfluxDB: Purpose built differently for time-series data* [online]. timescale, 2020. [cit. 2021/12/31]. Dostupné z: https://blog.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/.

[7] PostgreSQL. *PostgreSQL Documentation* [online]. postgresql, 2021. [cit. 2022/1/9]. Dostupné z: https://www.postgresql.org/docs/13/datatype.html.

[8] QuestDB. *QuestDB Documentation* [online]. questdb, 2021. [cit. 2022/1/9]. Dostupné z: https://questdb.io/docs/reference/sql/datatypes/.

[9] QuestDB. *QuestDB Documentation* [online]. questdb, 2021. [cit. 2021/12/30]. Dostupné z: https://questdb.io/docs/develop/insert-data.

[10] QuestDB. *QuestDB Documentation* [online]. questdb, 2021. [cit. 2021/12/31]. Dostupné z: https://questdb.io/docs/concept/storage-model/.

[11] QuestDB. *Benchmarking database performance with time series workloads* [online]. questdb, 2021. [cit. 2021/12/30]. Dostupné z: https://questdb.io/time-series-benchmark-suite/.

[12] QuestDB. *QuestDB Documentation* [online]. questdb, 2021. [cit. 2021/12/30]. Dostupné z: https://questdb.io/docs/develop/query-data.

[13] Rathee, K. *QuestDB vs. TimescaleDB* [online]. towardsdatascience, 2021. [cit. 2022/1/9]. Dostupné z: https://towardsdatascience.com/questdb-vs-timescaledb-38160a361c0e.

[14] TimescaleDB. *Timescale gives you the time-series performance you need for your critical workloads* [online]. timescale, 2021. [cit. 2022/1/9]. Dostupné z: https://www.timescale.com/compare.

# Přílohy

# A Seznam implementovaných funkcí a pseudokonstant

## Pseudokonstanty

Seznam pseudokonstant a jejich význam podle úrovně výpočtu:

`#GID` - id zařízení. Pro všechny úrovně stejné.

`#MT` - pro data představuje rozdíl časů mezi aktuálním datem a datem předešlým. Pro joby představuje rozdíl mezi start a stop časem, tedy sloupci `datetime` a `data_time`. U sumací reprezentuje rozsah sumace. Čas je vždy vrácen v sekundách.

`#VMT` - pro data představuje vektor konstant `#MT` a pro sumace rozíly mezi start a stop časem pro data v rozmezí sumace.

`#JT` - pro data představuje čas od začátku jobu a pro sumace celkový čas jobu.

`#REC` - číslo záznamu v jobu. Jen na úrovni dat.

`#RECS` - pro joby představuje počet záznamů v daném jobu. Pro data počet jobů v adném dni a pro sumace počet dní v zadaném rozsahu.

`#JB` - pro data představuje čas začátku joby, tedy sloupec `data_time`. Pro sumace reprezentuje první záznam v zadaném rozsahu.

`#JE` - stejné jako `#JB`, ale představuje čas konce jobu pro data, či poslední záznam pro sumace.

`#WB` - určuje kdy je definovaný zlom dne pro dané zařízení. Tato konstanta je povinná pro výpočet dní.

`#WE` - určuje čas zlomu pro další pracovní den. Tato konstanta je vypočtena z konstanty `#WB`.

`#NULL` - prázdný parametr. Pro všechny úrovně stejné.

# Funkce

Funkce jako sčítání, odčítání atd. můžou být zapsané jak operátorem, tak pojmenovaným ekvivalentem. Každá funce má předem definovaný počet vstupních parametrů. Dále se na tyto parametry bude odkazovat jako `a`, `b`, `c`, `d`, `e` a `f`, kde písmeno představuje pořadí parametru. Tedy `a` je první parametr atd. Parametry funkcí mohou být čísla, značeno `N`, řetězec `S`, vektor čísel `A` nebo vektor řetězců `T`.

### Skalární funkce

| **Funkce** | **Parametry** | **Popis** |
|---|---|---|
| + (ADD) | N N | součet `a` + `b` |
| - (SUB) | N N | rozdíl `a` - `b` |
| * (MUL) | N N | součin `a` * `b` |
| \(DIV) | N N | podíl `a` \`b` |
| ABS | N | absolutní hodnota `a` |
| MOD | N N | zbytek po dělení `a`%`b` |
| MAX | N N | maximum z `a` `b` |
| MIN | N N | minimum z `a` `b` |
| AND | N N | aritmetické AND mezi `a` `b` |
| OR | N N | aritmetické OR mezi `a` `b` |
| SHL | N N | binární rotace `a` doleva o `b` |
| SHR | N N | binární rotace `a` doprava o `b` |
| = (EQ) | N N | zda `a` je rovno `b` |
| EQS | S S | zda `a` je rovno `b` mezi řetězci |
| > (GT) | N N | zda je `a` větší než `b` |
| >= (GET) | N N | zda je `a` větší nebo rovno `b` |
| < (LT) | N N | zda je `a` menší než `b` |
| <= (LET) | N N | zda je `a` menší nebo rovno `b` |

| | | |
|---|---|---|
| POW | N N | mocnina `a` na `b` |
| IF | N S S | je-li podmínka `a` splněna je výsledek `b` jinak `c` |
| INR | N N N | je-li `a` v rozsahu <`b`;`c`> |
| TOR | N N N | je-li `a` v rozsahu <`b`;`c`> tak výsledek je `a` jinak nejbližší mez `b` čí `c` |
| IFNN | S N | není-li `a` číslo je výslekdem `b` jinak `a` |
| IFNA | S S | je-li `a` prázdný řetězec tak výsledkem je `b` jinak `a` |
| ISN | S | zda je `a` číslo |
| ISA | S | zda je `a` neprázdný řetězec |
| DDIF | S S | vrací rozdíl mezi časem `a` a časem `b` v sekundách |
| DADD | S N | vrací čas `a` ke kterému je přičteno `b` sekund |
| DWEEK | S | vrací den v týdnu podle `a` |
| GPSDIF | S S | spočte vzdálenost mezi hps body `a` `b` v metrech |
| GPSST | N S S | je-li změna mezi gps souřadnicí `b` a `c` menší než `a` je výsledkem `c` jinak `b` |
| ADDS | S S | spojí řetězce `a` `b` do jednoho |
| I2S | N | převede číslo `a` na řetězec |

| | | |
|---|---|---|
| GOTH | N N N N | sleduje průchod čísel `b` `c` číselm `d`, `a` udává jestli sledujeme zda jsou čísla klesající, rostoucí nebo bez podmínky |
| ROUND | N | zaokrouhlí `a` na celá čísla |
| ROUNDX | N N | zaokrouhlí `a` na `b` |
| ENUM | N S | vrátí `a`-tý záznam z `b`, prvky jsou odděleny znakem `&` |
| TOE | S S | přidá `a` do seznamu `b`, prvky jsou odděleny znakem `&` |
| TABLE | N S | převede číslo `a` dle tabulky `b` |
| RND | | náhodné číslo 0-1 `a` |
| RNDN | | náhodné číslo -1 až 1 při normálním rozdělení `a` |
| TRANS | N N N N | přechod čidel `a` `b` skrze rozsah `c` `d` |
| HILO | S N | `a` popisuje osu histogramu, `b` je index, funkce vrací spodní hodnotu na daném indexu |
| HIHO | S N | `a` popisuje osu histogramu, `b` je index, funkce vrací horní hodnotu na daném indexu |
| HIIX | S N | `a` popisuje osu histogramu, `b` je hodnota, funkce vrací index prvku, do kterého patří daná hodnota |

Tabulka A.1: Popis skalárních funkcí

**Sumační funkce**

| Funkce | Parametry | Popis |
|---|---|---|
| SNRM | A | průměr z `a` |
| SMED | A | medián z `a` |
| SMIN | A | minimum z `a` |
| SMAX | A | maximum z `a` |
| SSUM | A | suma `a` |
| SSUMX | T | suma `a` kde nečísla jsou vynechána |
| SSUMN | T | suma `a` kde všechny prvky musí být číslo |
| SBEG | T | první dato z `a` |
| SEND | T | poslední dato z `a` |
| SBEGN | N T | `a`-té dato od začátku z `b` |
| SENDN | N T | `a`-té dato od konce z `b` |
| SBEGNR | N A N N S | `a`-té dato od začátku z `b`, dato musí být v rozsahu <`c`;`d`>, jinak `e` |
| SENDNR | N A N N S | `a`-té dato od konce z `b`, dato musí být v rozsahu <`c`;`d`>, jinak `e` |
| SSUMD | A | suma diferencí z `a` |
| SSUMP | A | suma kladných diferencí z `a` |
| SSUMM | A | suma záporných diferencí z `a` |
| SCNT | T | počet dat v `a` |
| SCNT0 | T | počet prázdných dat v `a` |
| SCNTS | T | počet neprázdných dat v `a` |
| SCNTNR | A N N | počet dat v `a`, musí být v rozsahu <`b`;`c`>, pokud `b` > `c` tak počet dat mimo rozsah |

| | | |
|---|---|---|
| SDATCTRL | N @A A N | funkce pro kontrolu dat, `a` udává typ kontroly, `b` je označeno znakem @ protože představuje výstup, `c` jsou vstupní data a `d` je parametr |
| SLIST | N T | vytvoří seznam z `b`, položky jsou odděleny čárkou, `a` je typ seznamu |
| SA | T | převede vektor `a` na řetězec |

Tabulka A.2: Popis sumačních funkcí

**Vektorové funkce**

| **Funkce** | **Parametry** | **Popis** |
|---|---|---|
| VSEL | T A | vytvoří podmnožinu z `a` pro kterou platí `b` |
| VISR | T N N | vytvoří vektor hodnot 0 a 1 podle podmínky zda `a` je v rozsahu <`b`;`c`> |
| VAND | A A | operace AND mezi `a` `b` |
| VOR | A A | operace OR mezi `a` `b` |
| VNOT | A | negace nad `a` |
| VBEX | A N | pokud je `b` $> 0$ tak rozšíří bitové pole `a`, jinak pole `a` zmenší |
| VLAD | A A N | vytvoří vektor podle podmínky `a` $<$ (`b` - `c`) |
| VBEG | N T | vytvoří subvektor od začátku z `b` do `a` |
| VEND | N T | vytvoří subvektor od konce z `b` do `a` |

| | | |
|---|---|---|
| VIDX | N N T | vytvoří subvektor z `c` od `a` do `b` |
| VMID | N N T | vytvoří subvektor ze středu z `c` v rozsahu <střed + `a`; střed + `b`> |
| VNUM | T | vytvoří číselný vektor z `a`, nečísla jsou vynechána |
| VMUL | A A | vytvoří vektor který je součin mezi `a` `b` |
| VDIV | A A | vytvoří vektor který je podíl mezi `a` `b` |
| VADD | A A | vytvoří vektor který je součet mezi `a` `b` |
| VMULC | A A | vynásobí `a` číslem `b` |
| VADDC | A N | přičte k `a` číslo `b` |
| VSSS | A N | posune `a` o `b` a odečte od `a` |
| VSHIFT | A N | posune `a` o `b` |
| VT | T | vytvoří textový vektor z řetězce, položky odděleny podle čárky |
| VA | T | vytvoří číselný vektor z řetězce, položky odděleny podle čárky |
| VAN | T T | vytvoří vektor z řetězce, položky odděleny podle čárky |
| HISTS | T N | suma histogramu `a` a dělitele `b` |
| HISTD | S S N | podíl mezi histogramy `a` `b` a vydělen `c` |

| | | |
|---|---|---|
| HIST | S A A | vytvoří histogram podle osy `a`, `b` jsou hodnoty a počet ve skupinách se zvyšuje o `c` pokud je `c` definováno |
| HIST2 | S S A A A | vytvoří 2D histogram, kde osa X je `a`, osa Y je `b`, `c` a `d` jsou hodnoty a počet ve skupinách se zvyšuje o `c` pokud je `c` definováno |
| HISTIF | S A A A | vytvoří histogram podle osy `a`, `b` jsou hodnoty které splňují podmínku `d` a počet ve skupinách se zvyšuje o `c` |

Tabulka A.3: Popis vektorových funkcí

# B Uživatelská příručka

## Instalace nastavení databáze

V příloze je obsažena QuestDB databáze. Pro instalaci stačí jen extrahovat soubory do libovolné složky. Databáze se spouští přes soubor `bin/questdb.exe`. Pro vyvoření tabulek a vložení testovacích dat je možné spustit přiložený `SetupTestData.jar` soubor. Ten vytvoří všechny potřebné tabulky a vloží data do databáze ze souborů `data.txt`, `job.txt`, `day.txt` a `equations.txt`. Ke spuštěné databázi také lze přistoupit přes prohlížeč přes adresu: `localhost:9000`.

## Ovládání programu

Samotný klient se spouští přes `questDB_client.jar`. Soubor lze spustit bez argumentů pro přepočet dat z databáze nebo s dvěma argumenty pro výpočet sumací v zadaném rozsahu. Oba argumenty musí být ve tvaru `yyyy-MM-dd`, tedy například:

```
java -jar questDB_client.jar 2022-04-20 2023-06-25
```

Pro zvýšení limitu maximálně zpracovávaných záznamů je nutné přepsat danou proměnnou v souboru `Client.java`.