Platforma průmyslové spolupráce

CZ.1.07/2.4.00/17.0041

Název

Operátory pro zpracování proudů dat

Popis a využití

- práce s operátory v jazyce Esper pro Java
- Benchmarking
- výuka: pokročilá Java

Jazyk textu

- slovenský

Autor (autoři)

- Tomáš Hrušo

Oficiální stránka projektu:

- http://lasaris.fi.muni.cz/pps

Dostupnost výukových materiálů a nástrojů online:

- http://lasaris.fi.muni.cz/pps/study-materials-and-tools

Obsah

Úvod

Hneď ako začala výpočtová technika pracovať rýchlejšie ako človek, dávali sa jej úlohy na spracúvanie veľkého množstva dát a mala reagovať určitým spôsobom, ktorý záleží od prijatých dát. Odvtedy vznikli rôzne reprezentácie dát a prístupy ku ich spracovaniu. V tejto práci budem pracovať s dátami reprezentovanými ako udalosti a porovnám ich s databázovým systémom.

Hardvér už nie je taký drahý, ako býval. Firmy si môžu dovoliť najvýkonnejšie stroje. Čo sa týka úložného priestoru, tak vďaka virtuálizácii a ďalšiemu softvérovému vybaveniu môžeme teoreticky predpokladať, že máme neobmedzene miesta. Čo ma ale najviac v tejto práci zaujíma, je rýchlosť spracovania dát, takže odhliadneme od obmedzení v pamäti.

„Spracovanie dát je činnosť, ktorú vykonáva napríklad počítač alebo človek pri manipulácii s dátami s cieľom ich transformácie v rámci zadanej úlohy“ [7]. V tejto práci ma bude zaujímať samotný spôsob transformácie dát pomocou operátorov. Každý operátor je funkcia, ktorá mení vstup na výstup pomocou definovaných pravidiel. Existuje ich strašne veľa, tak som vybral niektoré, naviac používané. Kombináciou operátorov môžem prakticky z každého vstupu vytvoriť výstup aký chcem.

Prvá kapitola sa bude venovať teoretickému základu návrhu udalosťami riadenej sieti [3]. Pozriem sa konkrétne na každú zložku osobitne a zadefinujem pojmy súvisiace so spracúvaním dát.

V druhej kapitole porovnám systém riadenia toku dát(DSMS) a systém pre správu databáz (DBMS). Rozoberiem ich rozdiely a načrtnem ich využitie.

V tretej kapitole sa pozriem na kontexty a ich použitie pri spracúvaní dát. Rozoberiem rôzne druhy kontextov, nie len časové ale aj priestorové.

V štvrtej, poslednej kapitole popíšem formálne každý operátor, pridám príklady použitia a porovnám rýchlosť pri rôznych vstupných hodnotách.

Cieľom tejto práce je premerať operátory v spracúvaní dát. Zistiť pri akých podmienkach je operátor rýchlejší a pri ktorých naopak veľmi pomalý.

Kapitola 1

Spracúvanie dát

Spracúvanie dát je metóda sledovania a analyzovania toku udalostí/dát. Na základe analýzy toku môžeme následne vytvoriť novú udalosť, ktorú bude spracúvať iný tok udalostí. Týmto postupom vytvoríme komplexné spracovanie dát (Complex event processing - CEP). CEP analyzuje viacero tokov súčasne a na základe tejto analýzy môže vytvoriť novú udalosť a poslať ju do jedného alebo viacerých tokov dát. Tento postup je veľmi efektívny pre analýzu napríklad logu užívateľov, ktorý sa snažili prihlásiť do nejakého systému za určenú dobu, tzv. offline spracúvanie ale aj pre aplikácie, ktoré pracujú v reálnom čase(online) ako napríklad aplikácia, ktorá skúma počasie v Českej republike. Má k dispozícii niekoľko meteorologických staníc v Brne, Prahe a Olomouci. Každá stanica odošle každú minútu aktuálne informácie o počasí, ale aby sa nepreťažila sieť, každá inú sekundu. Aplikácia spracúva informácie tak, že prijme všetky udalosti a vnútorne ich logicky rozdelí a rozošle každú ku svojmu spracúvaniu, napríklad teplotu pošle stanici A, ktorá spočíta priemernú hodnotu a vypíše ju užívateľovi. Čiže, každú minútu si aplikácia pozbiera informácie od meteorologických staníc (od každej má jeden vstupný tok) a spočíta výsledné hodnoty, ktoré pošle ku vypísaniu na obrazovku (t.j. na výstupný tok).

1.1 Udalosť/Event

„Event je udalosť v rámci určitého systému alebo domény, je to niečo čo sa stalo, alebo sa uvažuje akoby sa stalo v tejto doméne. Slovo event sa tiež používa ako programovacia entita, ktorá reprezentuje udalosť vo výpočtovom systéme.“ [3]

Inými slovami, je to popísateľná množina dát, ktorá vznikla na základe nejakej zmeny v systéme, napríklad vytvorenie nového užívateľa alebo niečo jednoduché ako tik časovača alebo kliknutie myšou a ďalšie iné. Je to základná jednotka spracúvania udalostí podobná riadku v tabuľke databáze alebo objektu v OOP(objektovo orientovanom programovaní). Každý event musí obsahovať časové razítko jej vzniku.

Je možné že jedna udalosť v systéme môže vyvolať viac ako jeden event v aplikácii. Napríklad zrušenie dovolenky môže vygenerovať udalosť na odvolanie rezervácie v hoteli odoslanej na recepciu, ďalej na zrušenie pôžičky auta poslanej danej požičovni áut a samozrejme udalosť pre uvoľnenie rezervácie v lietadle.

Nemusí to byť reálna udalosť, ktorá sa stala v tomto svete, môže to byť udalosť, ktorá vznika vo virtuálnej realite alebo nejakej simulácii.

1.1.1 Typ udalosti

„Typ udalosti je špecifikácia pre množinu eventov, ktoré majú rovnaký sémantický zmysel a rovnakú štruktúru. Každý event je považovaný ako inštancia nejakého typu.“[3]

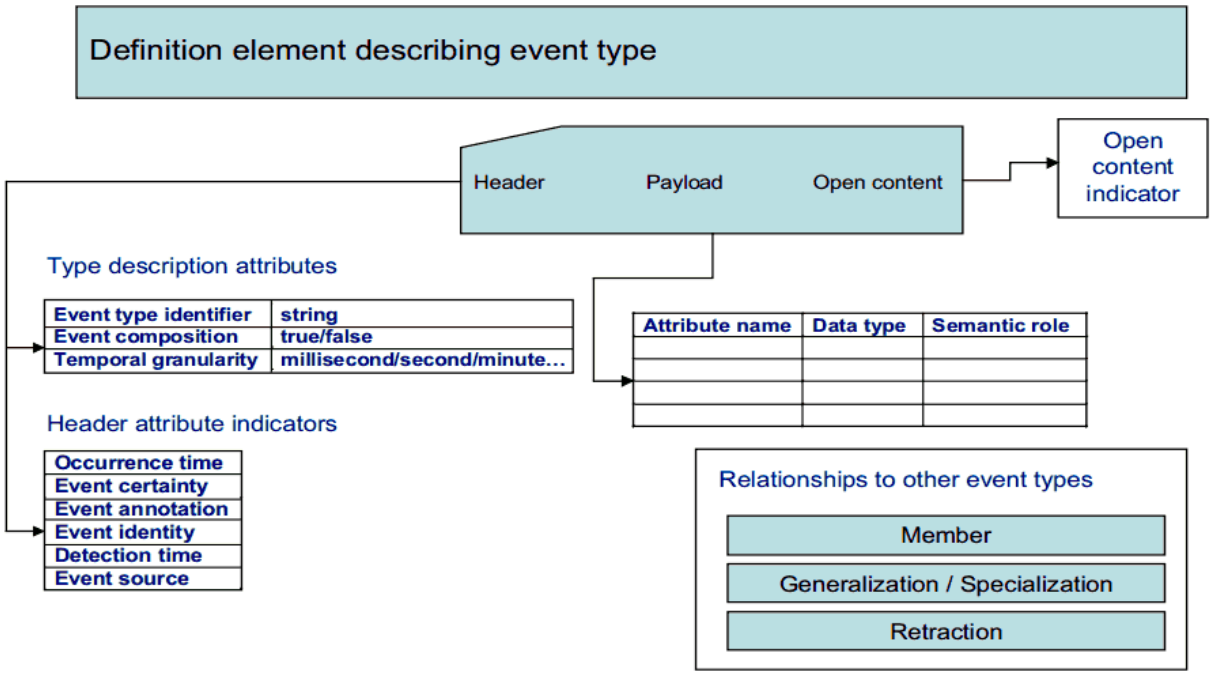
Keď máme nejakú veľkú aplikáciu, tak niekedy potrebujeme spracúvať iba niektoré udalosti. Napríklad ich môžeme rozdeľovať pomocou ich typu. V spomínanej aplikácii na skúmanie a meranie počasia máme niekoľko typov udalostí. Prvá je od meteorologickej stanice, typu Informácie o počasí, keď príde táto udalosť, aplikácia ju rozloží na menšie udalosti napríklad typu Teplota alebo Tlak vzduchu, a podobne.

1.1.2 Atribúty udalosti

„Atribút udalosti je komponenta štruktúry udalosti. Každý atribút má meno a dátový typ.“ [3]

Atribúty pomáhajú zisťovať príčinu udalostí a obohacujú udalosť o dôležité informácie. Každý atribút má dátový typ. Napríklad v spomínanej aplikácii na meranie teploty, teplomery posielajú udalosť s atribútom, ktorý má názov teplota a je dátového typu číslo.

1.1.3 Udalosť ako objekt



Obr. 1.1: Definičný element typu udalosti sa skladá s atribútov, ktoré opisujú obsah hlavičky a nákladu eventu, a listom vzťahov tohoto typu a ostatných typov udalostí. [3]

Obrázok 1.1 zobrazuje štruktúru definičného elementu. Ako môžeme vidieť, skladá sa z nasledujúcich informácií:

Atribúty opisujúce typ
: Podľa týchto atribútov vie detekčný systém určiť presný typ udalosti a skontrolovať či splňuje atribúty daného typu. **Identifikátor typu udalosti** obsahuje reťazec znakov, číže názov typu. **Kompozícia udalosti** je pravdivostná hodnota či je daný typ kompozíciou iného alebo nie. **Časové razítko a granularita** udáva v akých jednotkách sa bude merať čas. V spomínanej aplikácii na skúmanie počasia by tu bola hodnota sekundy.

Ukazovatele na atribúty v hlavičke
: Tieto atribúty slúžia na zistenie miesta a času vzniknutia udalosti, jej dôveryhodnosti, obsahuje taktiež volný text slúžiaci ako popis udalosti a jej unikátny identifikátor.

Definície atribútov
: Tu sa už nenachádzajú metadáta, ale dáta, ktoré podrobnejšie opisujú aktuálny výskyt udalosti. Každý takýto atribút musí byť pomenovaný (nejaký reťazec znakov), musí mať dátový typ (číslo, reťazec, pravdivostná hodnota, binárne dáta, dátum, lokácia, ...) a musí mať sémantickú rolu (normálny, bežný atribút alebo odkaz na inú udalosť).

Ukazovateľ na volný obsah
: S týmto ukazovateľom máme volnú ruku s formátovaním dát. Obsahuje dáta, ktorým formát môže určiť entita, ktorá udalosť vytvorila.

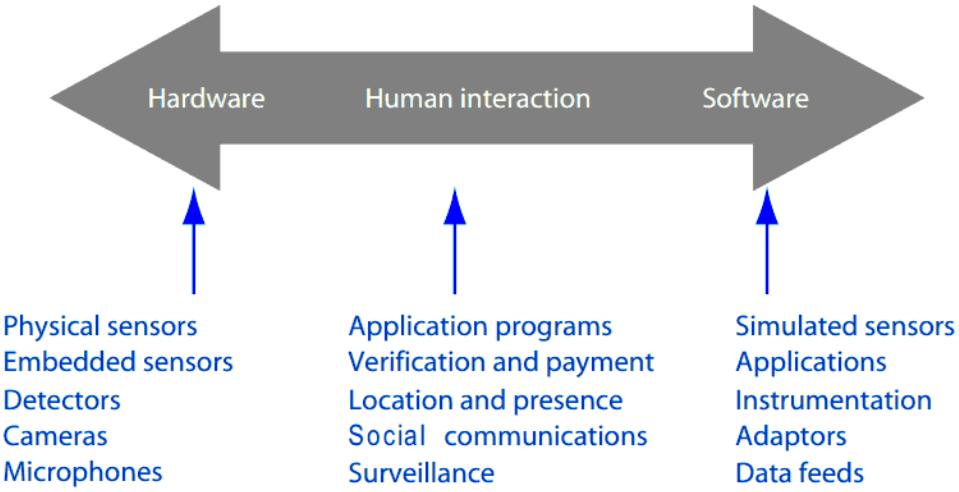
Vzťah s ostatnými typmi udalostí
: Určuje vzťah s inými typmi udalostí (ak vôbec nejaký existuje). **Člen** je vzťah hovoriaci že daná udalosť je člen inej zloženej udalosti. **Generalizácia a špecifikácia** je vzťah určujúci či je tento typ zovšeobecnením alebo špecifikáciou iného typu udalosti. **Stiahnutie** je vzťah ku druhému typu keď su navzájom logicky opačné. Napríklad ako udalosť typu *Zrušenie objednávky* je opakom udalosti typu *Zadanie objednávky*.

1.2 Stavebné kamene pre spracúvanie udalostí

Ako každý spôsob programovania, programovanie pomocou spracúvania udalostí je výnimočné. Samozrejme že sa dá simulovať v ostatných spôsoboch, napríklad v OOP (udalosť je objekt) alebo v databáze (udalosť je riadok tabuľky) ale táto architektúra programovania je založená na veľkej sieti rôznych objektov, spojenej kanálmi, ktoré prenášajú udalosti od jedného objektu ku druhému. Je veľmi jednoduché zmeniť funkčnosť vymenením jednej časti, časťou novou s rovnakým rozhraním. V tejto sekcii si postupne priblížime rôzne druhy týchto objektov a ich prepojení, stavebných kameňov pre spracúvanie dát.

1.2.1 Producenti udalostí

„Producent udalostí je entita na kraji systému pracujúceho s udalosťami, ktorá vkladá udalosti do systému.“ [3]



Obr. 1.2: Rôzne druhy producentov udalostí stretávajúce sa v aplikáciách na spracúvanie dát. [3]

Producent je dôležitá súčasť systému. Bez neho by nevznikali žiadne udalosti, ktoré by sa dali ďalej spracúvať. V spomínanej aplikácii na spracúvanie počasia, sú meteorologické stanice producenti udalostí. Ale nemusí to byť vždy tak jednoduché. Napríklad ak sú dve udalosťami riadené siete navzájom prepojené, tak tá druhá vidí tú prvú ako producenta. Existujú stovky príkladov, čo by mohlo byť producentom udalostí.

Ako môžeme vidieť na Obrázku 1.2 tak sa producenti dajú rozdeliť do troch skupín:

1. **Hardware**
   Hardwareový producenti sa používajú v strašne veľa odvetviach a základom je senzor. Ako napríklad senzor ktorý meră:

   - Teplotu
   - Vzduch
   - Vlhkosť
   - Tlak tekutiny
   - pH level

   Viac komplikovanejšie detektory využívajú viac senzorov naraz ako napríklad GPS lokátor, Siezmometer alebo policajný radar na meranie rýchlosti.

2. **Software**
   Aj keď software je často sprevádzaný hardwareovým producentom, existujú aj iba softwareový producenti. Napríklad simulované senzory. Sú to klasické hardwareové senzory, ktoré sú simulované vo virtuálnej realite alebo nejakom simulátore. Ďalej aplikácie, ktoré generujú udalosti na základe zadaných parametrov. Alebo generátory udalostí, ktoré nie sú vygenerované aplikáciou ale nepriamo nejakou monitorujúcou aplikáciou. Alebo takzvané adaptory, nie sú to aplikácie čo si „vymýšlajú" nové udalosti ale pozorujú okolie (napríklad viac hardwareových producentov) a na základe toho generujú nové udalosti, alebo len premieňajú rozhranie udalostí aby vyhovovali požiadavkám siete.

3. **Ľudská interakcia**
   Každá udalosť vygenerovaná ľudskou interakciou je vždy sprevádzaná kúskom hardwaru a softwaru. Väčšinou to je klikanie v grafickom rozhraní aplikácie alebo iná interakcia s užívateľským rozhraním. Pekným príkladom je sociálna komunikácia ako napríklad Facebook alebo Twiter.

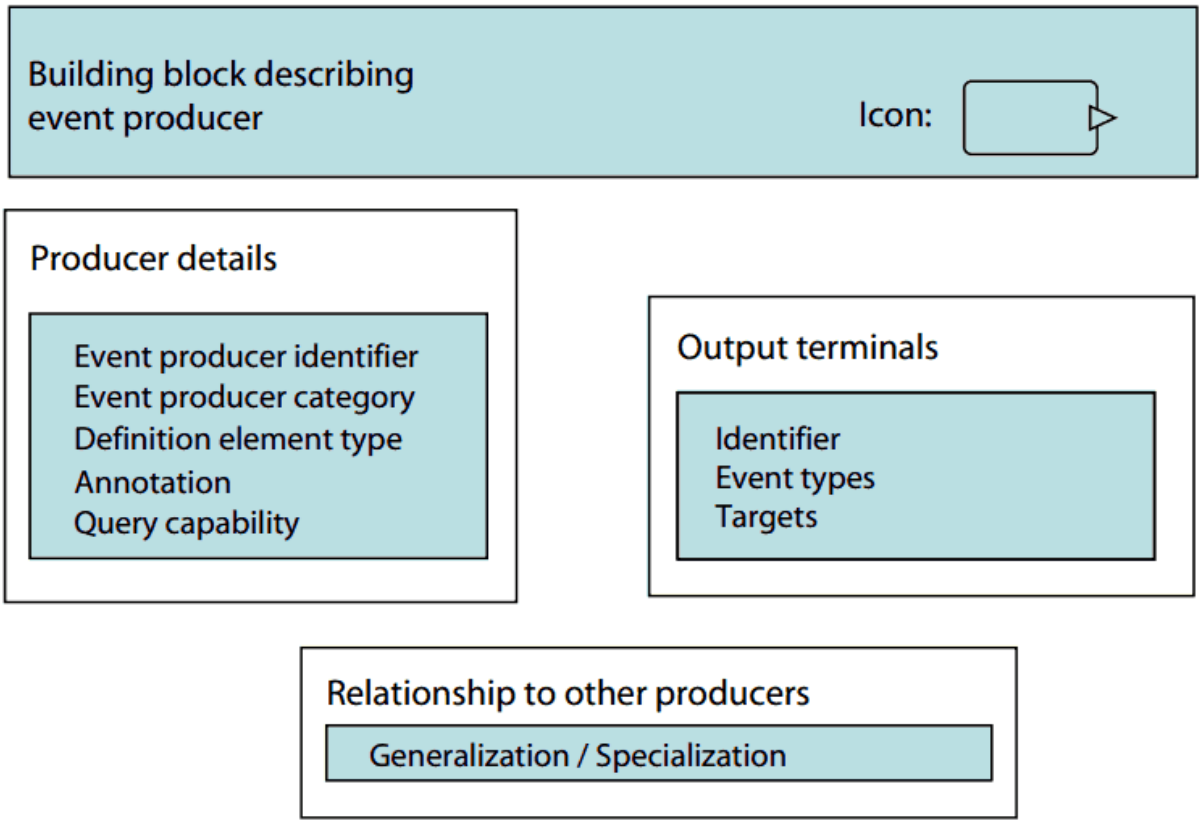
Všeobecne sa všetci títo producenti dajú zapísať pomocou definičného elementu producenta udalostí.

Ako môžeme vidieť na Obrázku 1.3, definičný element producenta udalostí sa skladá z nasledujúcich informácií:

Detaily o producentovi
Táto časť obsahuje informácie o producentovi. **Identifikátor** udáva meno producenta, ktoré môže byť odkazované z inej časti systému. **Kategória** udáva kategóriu producenta (napríklad *meteorologická stanica*). Nemá žiadny vplyv na spracovanie vygenerovaných udalostí, ale popisuje druh producenta. **Typ definičného elementu** obsahuje informáciu o definičnom elemente. Môže nadobudnúť jednu z hodnôt: abstraktný typ, trieda producentov alebo inštancia triedy producentov. **Anotácia** obsahuje užívateľské informácie o inštancii, triede alebo abstraktnom type. Je voliteľným atribútom. **Fronta** je logická hodnota indikujúca či producent má frontu alebo nie.

Výstupné terminály
Je to kolekcia výstupných terminálov kde pre každý jeden obsahuje tri informácie. **Identifikátor** sa používa na rozlíšenie výstupných terminálov v prípade že producent má viac ako jeden terminál. **Typy udalostí** sú kolekcia obsahujúca identifikátory typov udalostí, ktoré môže vyprodukovať cez daný výstupný terminál. Nie je to výhradná asociácia (viac výstupných terminálov môže produkovať rovnaký typ udalostí). **Ciele** sú list identifikátorov vstupných terminálov entít ktoré prijímajú udalosti z daného výstupného terminálu. Ak je producent abstraktný typ, môže mať aj nula cieľov. Viac výstupných terminálov môže mať rovnaký cieľ.

Vzťahy s ostatnými producentmi
Určuje vzťah ku ostatným producentom, môže sa jednať o zovšeobecnenie alebo o špeci-



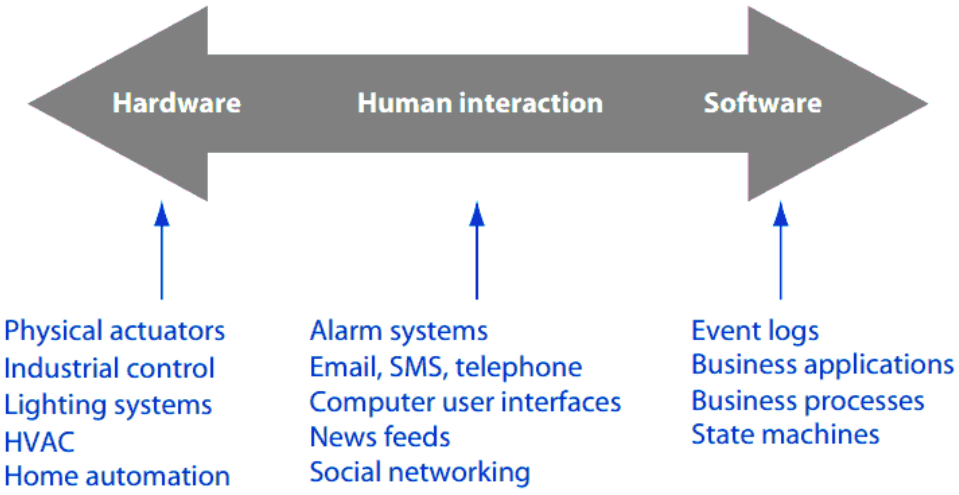
Obr. 1.3: Definičný element producenta udalostí obsahuje informácie o definičnom elemente samotnom, list výstupných terminálov, a list vzťahov ku ostatným definíciám producentov udalostí. [3]

fikáciu. Napríklad typ *Meteorologická stanica v Brne* je špecifikáciou všeobecnejšieho typu *Meteorologická stanica*.

1.2.2 Konzumenti udalostí

„Konzument udalostí je entita na kraji systému pracujúceho s udalosťami, ktorá prijíma udalosti zo systému.“ [3]

Konzument je zložka systému bez ktorej by samotné spracúvanie nemalo žiaden zmysel. Je to komponenta, ktorá spracúva (napríklad zobrazuje) udalosti ktoré sieť vygeneruje. V spomínanej aplikácii na spracúvanie počasia, je monitor užívateľa konzument udalostí. Ale nie vždy to musí byť tak jednoduché. Napríklad keď sú dve udalosťami ovládané siete navzájom prepojené, tak prvá vidí tú druhú ako svojho konzumenta. Existuje veľa príkladov, čo by mohlo byť konzumentom udalostí.



Obr. 1.4: Druhy konzumentov udalostí stretávajúce sa v aplikáciách na spracúvanie dát. [3]

Ako môžeme vidieť na Obrázku 1.4 tak sa konzumenti dajú rozdeliť do troch skupín:

1. **Hardware**
   Hardwareový konzument je často označovaný ako aktivátor. Keď dostane udalosť, tak vykoná nejakú akciu v reálnom svete. Napríklad zamykanie a odomykanie dverí, kontrolovanie výhybiek vlaku, a ďalšie iné.

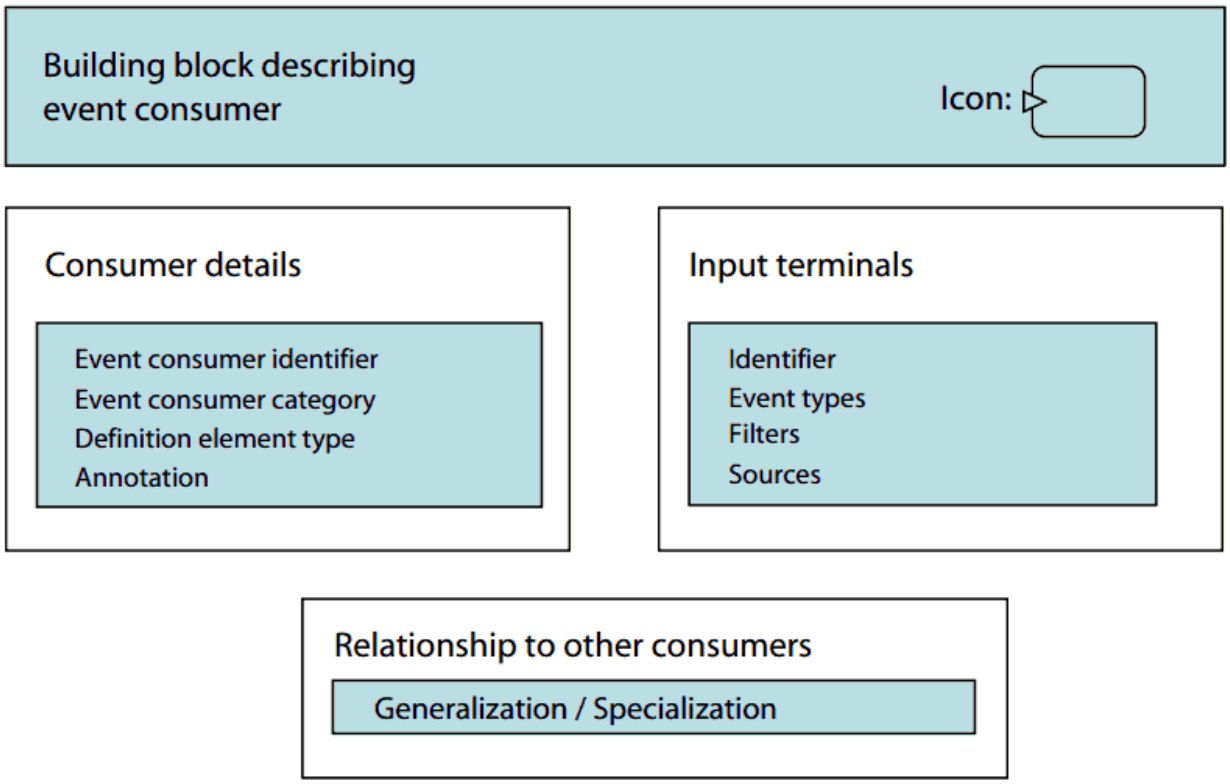
2. **Ľudská interakcia**
   Človek ako konzument udalostí potrebuje vizualizáciu dát, napríklad monitorom. Samotná vizualizácia je prevádzaná softwareom alebo hardwareom, ktorý vizualizáciu podporuje. Napríklad spomínaný monitor, aplikácia si navrhne užívateľské prostredie a na základe prichádzajúcich udalostí ho mení. Napríklad v aplikácii na skúmanie a meranie počasia, keď príde udalosť, ktorá ukazuje že v Brne prší, aplikácia zmení obrázok pri Brne na mrak s kvapkami. Ale ak následne príde informácia že v Olomouci svieti slnko, v sekcii pre Olomouc sa zmení obrázok na slnko. Výborným príkladom sú Ambient prístroje [1], tie reagujú na prichádzajúcu udalosť zmenou farby.

3. **Software**
   Softwareovým konzumentom udalostí je napríklad log aplikácie kde sa zapisujú výsledky alebo iné operácie, ktoré aplikácia vykonala. Môže sa hocikedy pozrieť na ľubovolný bod v čase a zistiť čo aplikácia vtedy vykonávala a podla toho sa zachovať.

   Na základe udalosti môže nejaký systém zmeniť svoj stav. Napríklad webovej stránke, ktorá ponúka veľa webových aplikácií, príde udalosť o veľkej zátaži serveru, ktorú zanalyzuje a na základe toho zmení svoj stav na *zaťažená* a bude poskytovať iba 50% aplikácií.

1. `http://www.ambientdevices.com/`

Všeobecne sa všetci títo konzumenti dajú zapísať pomocou definičného elementu konzumenta udalostí.



Obr. 1.5: Definičný element konzument udalostí obsahuje detaily o definičnom elemente samotnom, list vstupných terminálov, a list vzťahov ku ostatným definíciám konzumentov udalostí. [3]

Ako môžeme vidieť na Obrázku 1.5, definičný element konzumenta udalostí sa skladá z nasledujúcich informácií:

Detaily o konzumentovi
: Táto časť obsahuje informácie o konzumentovi. **Identifikátor** udáva meno konzumenta, ktoré môže byť odkazované z inej časti systému. **Kategória** udáva kategóriu konzumenta (napríklad *log*). Nemá žiaden vplyv na spracovanie udalostí, ale popisuje druh konzumenta. **Typ definičného elementu** obsahuje informáciu o definičnom elemente. Môže nadobudnúť jednu z hodnôt: abstraktný typ, trieda konzumentov alebo inštancia triedy konzumentov. **Anotácia** obsahuje užívateľské informácie o inštancii, triede alebo abstraktnom type. Je voliteľným atribútom.

Výstupné terminály
: Je to kolekcia vstupných terminálov kde pre každý jeden obsahuje štyri informácie. **Identifikátor** sa používa na rozlíšenie vstupných terminálov v prípade že konzument má viac

ako jeden terminál. **Typy udalostí** sú list obsahujúci identifikátory typov udalostí, ktoré sú akceptované daným vstupným terminálom. Nie je to výhradná asociácia (viac vstupných terminálov môže akceptovať rovnaký typ udalostí). **Filtre** sú rozšírením a spresnením predchádzajúcej kolekcie. Špecifikujú aké udalosti môžu byť akceptované týmto vstupným terminálom. **Zdroje** sú list identifikátorov výstupných terminálov entít ktoré odosielajú udalosti do daného výstupného terminálu. Ak je konzument abstraktný typ, môže mať aj nula zdrojov.

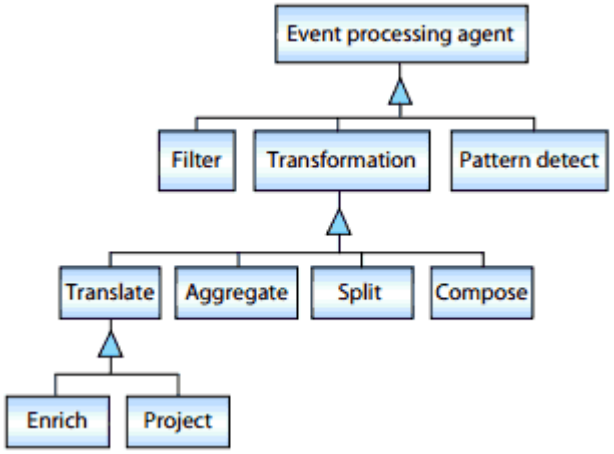
Vzťahy s ostatnými konzumentmi

Určuje vzťah ku ostatným konzumentom, môže sa jednať o zovšeobecnenie alebo o špecifikáciu. Napríklad typ *log záťaže systému* je špecifikáciou všeobecnejšieho typu *log*.

1.2.3 Agenti spracúvania udalostí

„Agent spracúvania udalostí je softwareový modul ktorý spracúva udalosti.“ [3]

Agent spracúvania udalostí tvorí jadro udalosťami riadenej architektúry. Vykonáva 3 základné funkcie: filtrovanie, porovnávanie a odvodzovanie.

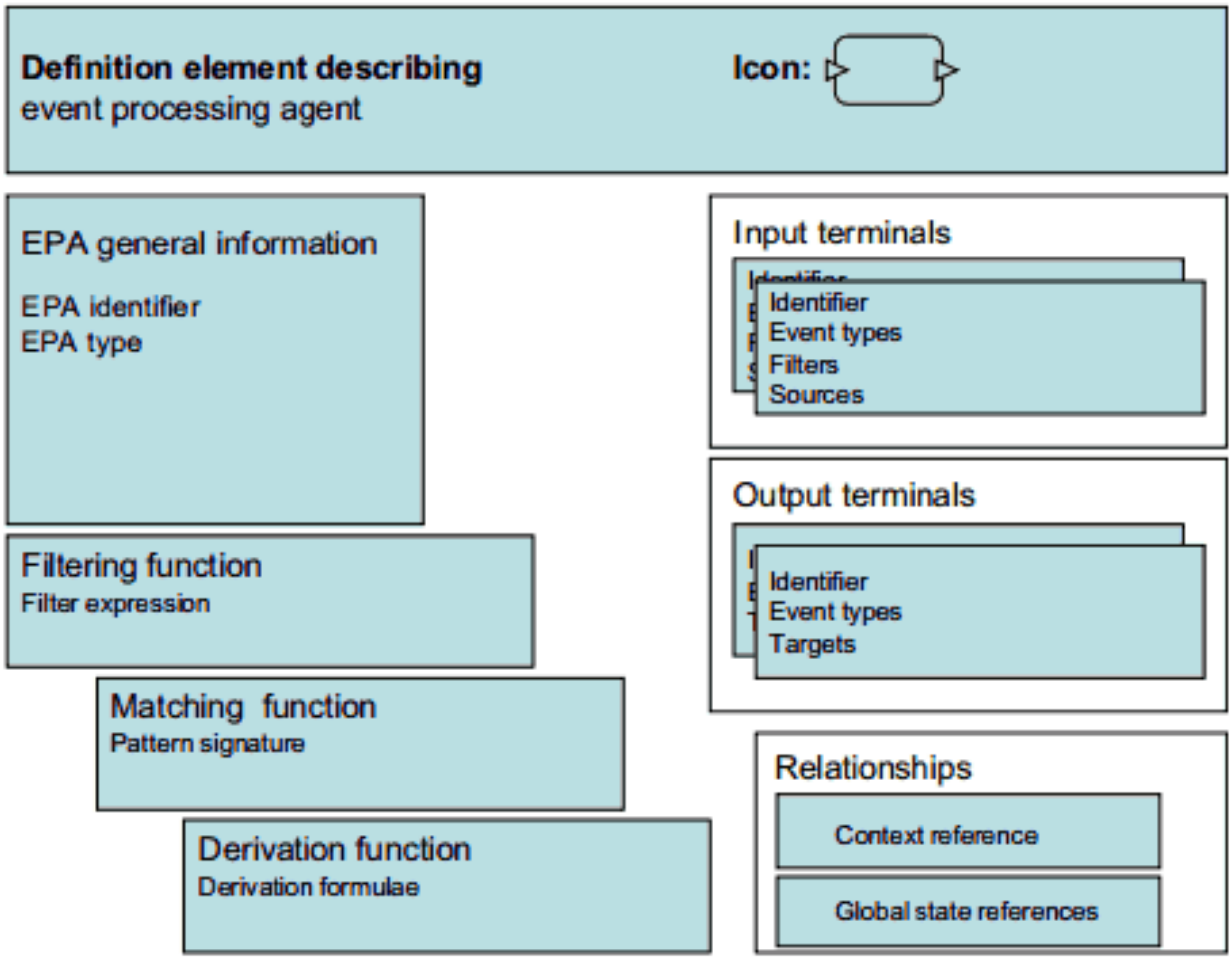


Obr. 1.6: Typy agentov. Tento diagram ukazuje dedičnú hierarchiu, napríklad, Obohatenie a Projekcia sú špeciálne prípady Prekladania, ktoré je špeciálnym prípadom Transformácie. [3]

V agentovi najskôr prebieha filtrovacia funkcia, ktorá zahodí všetky udalosti, ktoré nespĺňajú požadované vlastnosti. Potom nastane porovnávanie, ktoré hľadá zadaný vzor vo vstupných udalostiach, a nakoniec sa na výstupe predchádzajúceho kroku spustí funkcia na odvodenie finálnej verzie udalostí, ktoré sú vypustené výstupným terminálom ku ďalšiemu spracúvaniu. Ako uvidíme ďalej, nie každý typ agenta musí využívať všetky 3 funkcie.

Všeobecne sa agenti dajú zapísať pomocou definičného elementu agenta spracúvania udalostí.

Ako môžeme vidieť na Obrázku 1.7, definičný element agenta sa skladá z nasledujúcich informácií:



Obr. 1.7: Všeobecný definičný element agenta spracúvania udalostí. [3]

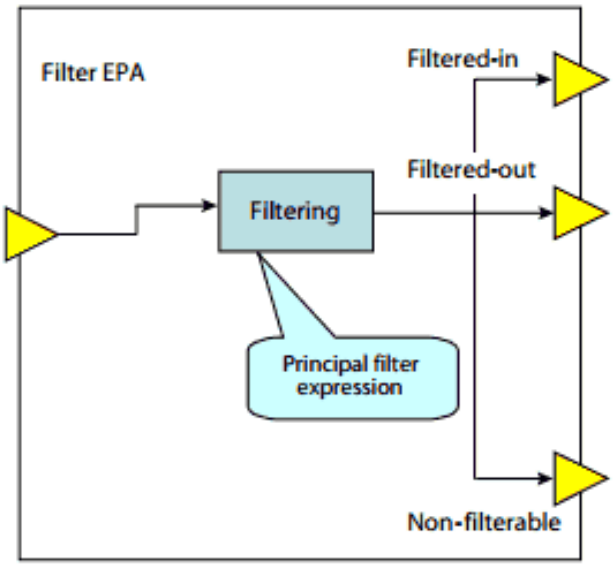
Všeobecné informácie
Táto časť obsahuje **identifikátor**, ktorý reprezentuje unikátne meno agenta, ktorého je to definičný element. Ďalej obsahuje **typ agenta**, tento atribút zaraďuje agenta do nejakej zo skupín na Obrázku 1.6.

Vstupné/výstupné terminály
Zoznam/list vstupných terminálov s rovnakým rozhraním ako 1.5. Zoznam/list výstupných terminálov s rovnakým rozhraním ako 1.3.

Filtrujúci agenti

Keď máme obrovskú aplikáciu obsahujúcu veľa producentov udalostí, tak sa môže stať že potrebujeme iba určitý druh udalostí. Ku tomuto výsledku sa môžeme dopracovať viacerými prostriedkami. Ako sme videli už v kapitole 1.2.2, tak sa dá filtrovať na vstupných termináloch pomo-

cou listu typov udalostí, ktoré môžu byť prijaté. Podobne môžeme na vstupnom terminály zadať filtrovací výraz, ktorý ak sa vyhodnotí na PRAVDA, tak je udalosť prijatá, a ak sa vyhodnotí na NEPRAVDA, tak sa udalosť zahodí.



Obr. 1.8: Filtrovací agent ukazujúci vstupný terminál a 3 výstupné. [3]

Niekedy je ale vhodné spracúvať aj udalosti, ktoré neprejdú filtrovacím výrazom. Na to sa používajú špeciálny filtrovací agenti. Ako môžeme vidieť na obrázku 1.8, tento druh agentov má 3 výstupné terminály. Nielen že môžete spracúvať aj udalosti, ktoré neprejdú filtrovacím výrazom, ale môžete spracúvať aj udalosti na ktoré sa nedá aplikovať filtrovací výraz (napríklad sú iného typu, alebo niektorý z atribútov nie je definovaný, a podobne).

Ďalší a posledný typ filtrovania spočíva v kontextoch, kedy spracúvame len niektoré udalosti podľa časového razítka alebo pozície. Ku kontextom sa dostanem neskôr.

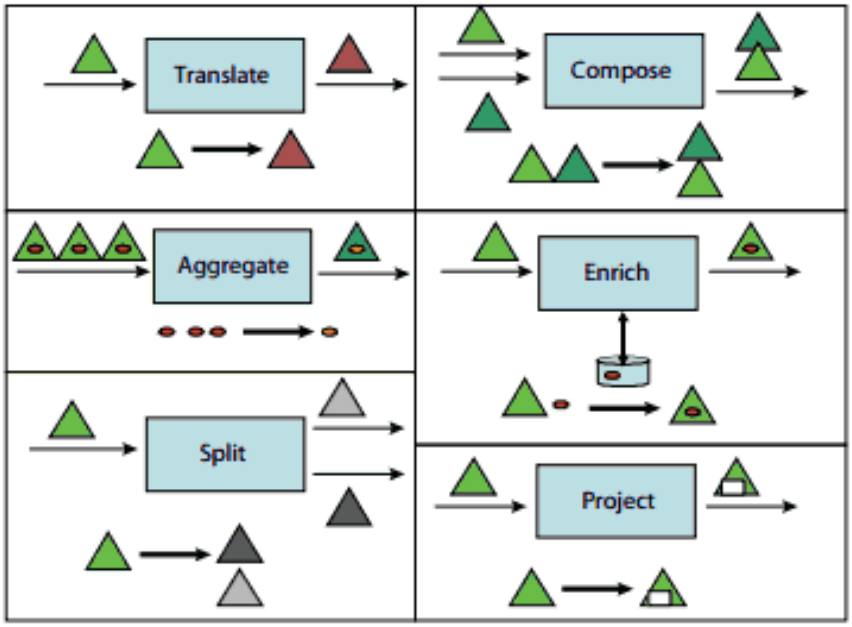
Transformační agenti

Keď už máme odfiltrované nepotrebné udalosti, chceme s nimi pracovať. Môžeme použiť niekoľko techník, ktoré sú zobrazené na obrázku 1.9.

Rozlišujeme dva druhy transformačných agentov, bez stavu a so stavom. Bez stavu sú takí, že každú udalosť spracúvajú rovnako (projekcia, obohatenie a preklad). Transformační agenti so stavom spočívajú v tom, že výstupná udalosť je odvodená z viacerých vstupných udalostí (kompozícia, agregácia a rozdelenie).

Obohatenie/Enrich

Tento transformačný agent obohatí vstupnú udalosť o nejakú doplňujúcu informáciu. Ako napríklad pri objednávke, ktorá obsahuje adresu, sa podľa adresy nájde meno osoby, ktorá tam býva a pridá sa ako nový atribút.



Obr. 1.9: Šesť podtypov transformačného agenta ukazujúcich vstupné a výstupné terminály a typ transformácie. [3]

Projekcia/Project
Opačný agent ako predchádzajúci. V tomto prípade nejakú informáciu zo vstupnej udalosti odoberáme. Ako napríklad nepotrebné identifikátory, čísla vo výslednom produkte.

Preklad/Translate
Je kombináciou predchádzajúcich dvoch. Môže odoberať aj pridávať nové atribúty do vstupnej udalosti. Výstupná udalosť sa vôbec nemusí podobať vstupnej. Tento agent môže slúžiť ako adaptér, ktorý prekladá z jedného rozhrania do druhého.

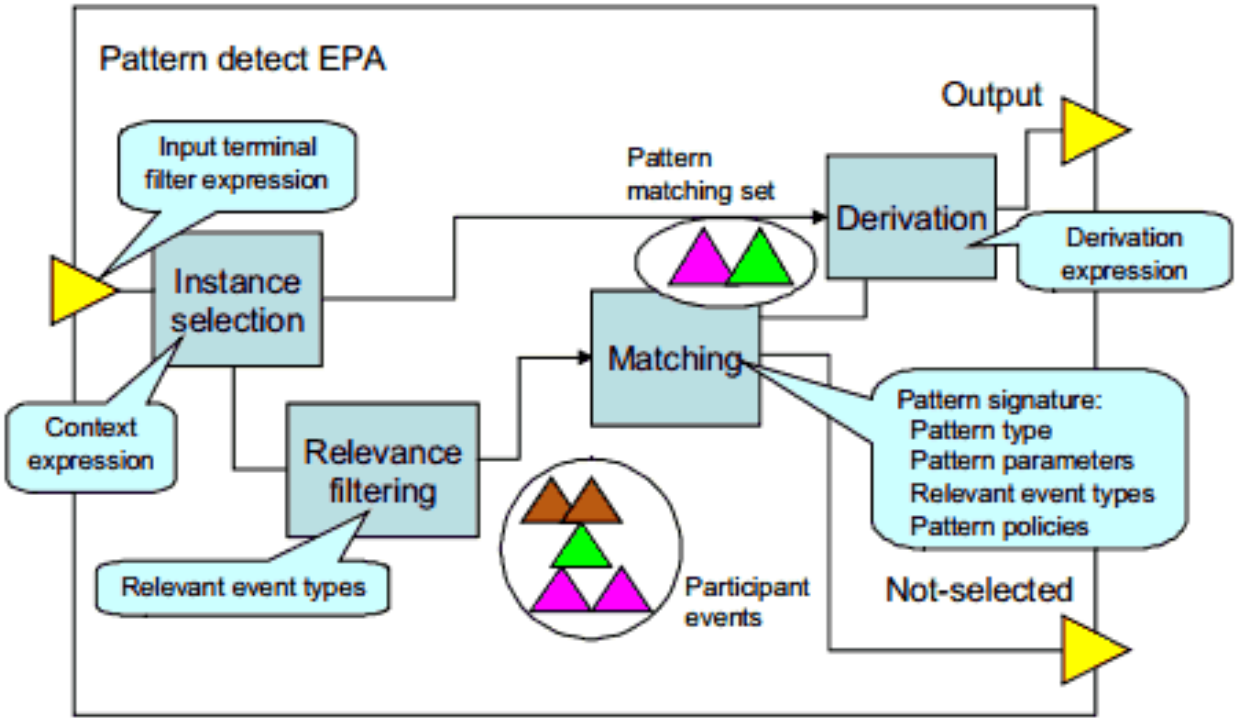
Kompozícia/Compose
Ako názov napovedá, tento agent komponuje, skladá viacero udalostí do jednej pomocou nejakého daného pravidla. Podobne ako operátor JOIN, ktorý rozoberiem neskôr.

Rozdelenie/Split
Tento agent je opak predchádzajúceho. Rozdeľuje udalosti podľa nejakého daného pravidla. Podobne ako v aplikácii na meranie počasia, ktorá rozdeľuje udalosť od meteorologickej stanice na menšie, ktoré rozposiela špeciálnym agentom na ďalšie spracúvanie. Podobne ako operátor GROUP BY, ktorý rozoberiem neskôr.

Agregácia/Aggregate
Agent slúžiaci na prevádzanie agregačných funkcií ako počet, priemerná hodnota, atď. Ako v spomínanej aplikácii, kde počítame priemernú hodnotu teploty.

Detekční agenti

Detekční agenti sú dôvod, prečo sa používa udalosťami orientované programovanie. Nielen že môžeme filtrovať a transformovať udalosti, vďaka týmto agentom je možné eventy kombinovať a detegovať vzory.



Obr. 1.10: Logická štruktúra detekčného agenta, ukazujúca tri logické časti s „podpisom vzoru“,ktorý riadi proces detegovania vzoru. Trojuholníkové objekty reprezentujú inštancie udalosti. [3]

Ako je vidieť na Obrázku 1.10, detekčný agent používa všetky tri zložky (filtrovaciu, porovnávaciu aj odvodzovaciu).

Existujú dva druhy typov vzorov, základný a dimenzionálny. Základné vzory sú napríklad:

All pattern
: Tento vzor obsahuje list typov udalostí. Ukladá si udalosti a pri každej prichádzajúcej skontroluje, či má už takú udalosť uloženú, ak nie „vyškrtne“ si ju zo zoznamu, ak hej, čaká na ďalšie. Keď prijme všetky typy udalostí, splní vzor a vygeneruje výslednú udalosť.

Any pattern
: Tento vzor funguje podobne ako predchádzajúci, akurát je spokojný ak mu príde aspoň jedna udalosť so zoznamu.

Absence pattern
Vzor, ktorý sa využíva najčastejšie ako časovač. Keď mu priradíme temporálny kontext, o ktorom bude reč neskôr, tak bude spokojný ak za daný čas nepríde ani jedna udalosť.

Treshold patterns
Sú to vzory, ktorým dáme číslo a názov atribútu. Prehľadávajú prichádzajúce udalosti a kontrolujú či napríklad počet udalostí sa nerovná tomu číslu(count pattern), alebo či hodnota nie je väčšia/menšia ako zadané číslo(value min/max/average pattern). Môžeme mu priradiť aj funkciu, ktorá niečo počíta s atribútmi a porovnáva s výslednou hodnotou.

Subset selection pattern
Tieto vzory pracujú s podmnožinami udalostí ako napríklad hľadajú určitý počet najväčších/najmenších hodnôt(the relative n highest/lowest values pattern).

Okrem týchto základných existujú ešte aj dimenzionálne, ktoré pracujú s časom, priestorom alebo ich kombináciou. Nasledujúce vzory su dimenzionálne:

Sequence pattern
Tento vzor deteguje sekvencie udalostí. Je spokojný ak prišla aspoň jedna udalosť z každého typu udalostí, ktoré má na zozname, a prišli v rovnakom poradí. Tento vzor budem brať ako operátor a rozoberiem ho neskôr.

First/Last n pattern
Tieto vzory redukujú veľký počet udalostí na n alebo menej udalostí. Ak ich je menej ako n, vybrané sú všetky. *First n pattern* deteguje prvých n udalostí a *Last n pattern* opačne posledných n udalostí.

Trend patterns
Tieto vzory pozorujú určitý atribút udalostí a porovnávajú ich medzi sebou. Napríklad *Increasing pattern* deteguje či každá nová udalosť má daný atribút väčší ako predchádzajúca. *Decreasing pattern* deteguje či je daný atribút vždy menší. *Non-increasing/non-decreasing pattern* detekujú nestúpanie/neklesanie atribútu.

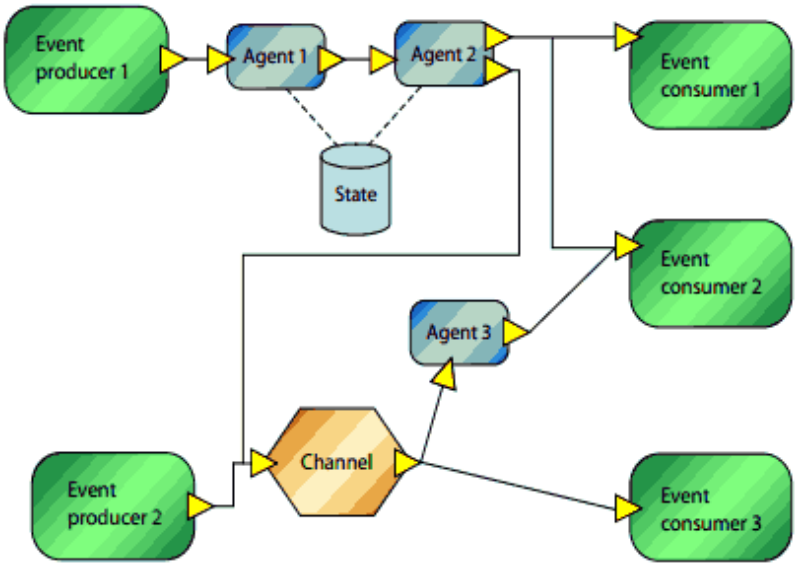
Spatial patterns
Tieto vzory pracujú s GPS lokáciou udalosti. Kontrolujú vzdialenosť udalosti od určiteho bodu(min/max/average pattern) alebo vzdialenosti všetkých udalostí po dvojiciach(relative min/max/average pattern)

Spatiotemporal patterns
Ako posledný typ sú detekční agenti, ktorí pracujú s GPS lokáciou a časom zároveň. Pozorujú zmenu polohy udalosti s priebehom času a testujú zmenu jej smeru. Napríklad či stojí na mieste alebo sa pohybuje (ne)rovno.

1.3 Sieť spracúvania udalostí

Skombinovaním predchádzajúcich stavebných kameňov, prepojených kanálmi vytvoríme sieť na spracúvanie udalostí. Od producentov sú do nej posielané udalosti, ktoré sieť pomocou rôznych agentov spracúva, transformuje a deteguje rôzne vzory. Nakoniec spracované udalosti pošle konzumentom.

Ako vidíme na Obrázku 1.11 sieť môžeme reprezentovať grafickou notáciou, pospájaním stavebných kameňov šípkami v smere posielania udalostí. Na obrázku môžeme vidieť zatiaľ nespomenuté stavebné kamene ako kanál (Channel) a globálny stavový element (State). **Kanál** uľahčuje grafickú notáciu tým, že rozdeľuje M:N reláciu na M:1 a 1:N relácie, čo sa ľahšie zakresľuje. **Globálny stavový element** slúži na zachovávanie informácií bez ohľadu na spracúvanú udalosť. Interne sa môže podobať klasickej relačnej databáze, ktorá mení svoj stav na základe prijatia rôznych udalostí. Napríklad môže mať uložený jazyk, v ktorom sa vykonáva spracúvanie udalostí a na základe tejto informácie sa môžu udalosti preposielať cez prekladového agenta. Ak užívateľ zmení jazyk, je globálnemu stavovému elementu zaslaná udalosť na zmenenie jazyku.



Obr. 1.11: Ukážka siete spracúvania udalostí ukazujúca grafickú reprezentáciu a ilustruje niektoré vlastnosti siete. [3]

Keďže sieť môže byť veľmi komplikovaná, môžeme ju rozdeliť na menšie časti, ktoré nakoniec spojíme do jednej veľkej. Pozrieme sa na ňu ako na obyčajného agenta, ktorý má počet vstupných terminálov rovnaký ako ma podsieť producentov a výstupných terminálov rovnako veľa, ako má podsieť konzumentov.

Táto kapitola obsahuje základné teoretické základy spracúvania udalostí. Ďalej sa pozriem na spôsoby spracúvania dát/udalostí pomocou systému riadenia toku dát (DSMS) a systému pre správu databáz (DBMS).

Kapitola 2

Spôsoby spracúvania udalostí

V tejto kapitole sa pozriem na základy spracúvania dát pomocou systému riadenia toku dát(DSMS) a systému pre správu databáz(DBMS). DSMS je rozšírením DBMS o dôležité koncepty pri spracovaní dát. Existujú tam kontinuálne dotazy, ktoré sa nevykonajú raz ako v databáze, ale existujú tam permanentne. Dalo by sa to simulovať v databáze pomocou spúšťačov(triggerov) pri vkladaní do tabuľky, ale nie je to rovnako výkonné. Spúšťač vždy pracuje s celou tabuľkou, naproti tomu, kontinuálny dotaz si zaznamenáva informácie postupne, a pracuje iba s dátami, ktoré práve potrebuje. Napríklad dotaz na spočítanie udalostí v priebehu minúty. Spúšťač si pri každej novej udalosti vždy prepočíta celú tabuľku, plus musí odfiltrovať udalosti, ktoré nepatria do minútového intervalu, čo je časovo veľmi náročné. Kontinuálny dotaz pri každej novej udalosti pripočíta jednotku ku počítadlu a odčíta počet udalostí, ktoré sú staršie ako minúta(tie sa vyhodia ako už nepotrebné).

Systém pre správu databáz	**Systém riadenia toku dát**
Perzistentné dáta	Nestály dátový prúd
Náhodný prístup	Sekvenčný prístup
Dotazy na jedno použitie	Kontinuálne dotazy
(Teoreticky) neobmedzené sekundárne úložisko	Limitovaná hlavná pamäť
Iba momentálny stav je relevantný	Treba uvažovať ako je zoradený vstup
Relatívne malá úroveň obnovovania	Potencionálne extrémne veľká úroveň obnovovania
Malinké alebo žiadne časové nároky	Nároky v reálnom čase
Predpokladajú sa presné údaje	Predpokladajú sa zastaralé/nepresné údaje
Plánované spracovanie dotazu	Variabilné príchody dát a ich charakteristiky

Tabuľka 2.1: Rozdiely medzi DBMS a DSMS[8]

V Tabuľke 2.1 vidíme hlavné rozdiely medzi týmito dvoma systémami. Jeden som už trochu začal rozoberať, ale v priebehu kapitoly sa z oboch pohľadov pozriem na každý problém osobitne.

2.1 Systém pre správu databáz

V databázovom systéme môžeme uchovávať udalosti ako riadky v tabuľke. Problém spočíva v tom, že databáza očakáva stále dáta, ktoré užívateľ uloží a potom ich opakovane číta a používa. Udalosti sú ale na jedno použitie. Reprezentujú okamih v čase, ktorý nastal na základe nejakej zmeny v systéme. Keď sa zmena spracuje, pokračuje sa ďalšou udalosťou. To nás privádza ku potrebe sekvenčného prístupu, ktorý databáza nemá. Pristupuje ku dátam náhodne, čo môže byť menej optimálne pri postupne prichádzajúcich udalostiach, ktoré sa spracúvajú sekvenčne. Náhodný prístup môže byť naopak efektívnejší pri veľkom okne s použitím konkrétneho operátora, ku tomu sa ale dostanem neskôr.

Každá nová udalosť je spracúvaná alebo upravovaná(záleží na aplikácii). V databázovom systéme sa dá pracovať s každou novou udalosťou pomocou spúšťačov, ktoré pri každom vyvolaní, musia prejsť celú tabuľku, pretože spolu nezdieľajú žiadne dáta. Sú to dotazy na jedno použitie.

Veľká výhoda databázového systému je v teoreticky neobmedzenej pamäti keďže nikdy dopredu nevieme, koľko udalostí bude potrebné uložiť pri ich spracúvaní. Ale s veľkým množstvom dát, prichádza časová náročnosť ich spracúvania. Databáza sa snaží optimalizovať dotazy, ktoré sú na ňu dotazované aby prebehli čo najrýchlejšie, ale všeobecne na databázový systém nie sú časové nároky vyhrotené do takých vysokých hodnôt ako potrebuje ku svojmu rýchlemu prevozu spracúvanie dát. To znamená ze dotaz by sa mal vyhodnotiť do príchodu následujúcej udalosti.

Databázový systém nie je navrhnutý tak, aby sa staral o zoradenie vstupu, ktoré môže byť porušené pomalou linkou(udalosť, s časovým razítkom τ sa dostane ku agentovi neskôr ako udalosť s časovým razítkom $\tau + 1$). Dôležitý je iba momentálny stav databázy nad ktorou sa spustí dotaz, ktorý si musí sám zabezpečiť zoradenie spracúvaných dát.

Použitie databázového systéme slúži hlavne na uschovávanie trvalých, presných a nemeniacich sa dát. Ak je nad dátami spustená nejaká procedúra, môže sa naplánovať, čo pri udalostiach neplatí, pretože nevieme kedy nastanú.

Systému pre správu databáz chýbajú základné prvky spracúvania dát, ale stále sa dá použiť na túto činnosť. Každá operácia sa dá simulovať pomocou procedúr a spúšťačov. Schéma(atribúty) udalosti reprezentujú stĺpce v tabuľke, ktorá reprezentuje prúd udalostí. Obtiažné, ale nie nemožné, sú na simuláciu kontexty, ku ktorým sa dostanem v nasledujúcej kapitole, pretože sa musia zisťovať pri každej novej udalosti, čo môže byť časovo náročné. Teraz ešte na porovnanie rozoberiem systém určený pre spracúvanie udalostí, DSMS.

2.2 Systém riadenia toku dát

Projektov na zdokonalenie spracúvania dát existuje niekoľko [2]. Ale všetky musia bojovať so základnými výzvami samotného spracúvania udalostí. Musia predpokladať nestály dátový prúd. Neexistuje možnosť predpovedať kedy udalosť nastane a dopredu sa na ňu nejako špecificky pripraviť. Môžu využívať informácie, že udalosti prichádzajú jedna za druhou a nastaviť spracúvanie na sekvenčné, čo môže výrazne zvýšiť výkon aplikácie. Dotazy musia byť natoľko rýchle aby zvládali veľmi rýchly prísun nových udalostí v reálnom čase. Z toho vyplýva že môže byť potrebná extrémne vysoká úroveň obnovovania.

Ako som už spomenul v kapitole 2.1, udalosti nemusia vďaka rýchlosti siete prísť v presnom poradí. S týmto problémom sa dá vysporiadať rôznymi spôsobmi ako vzorkovaním(sampling), dávkovým spracovaním(batch processing) alebo pomocou kĺzajúcich okien(sliding windows). Tieto spôsoby rozoberiem neskôr v priebehu kapitoly ale ani jeden nemôže byť natvrdo implementovaný v jadre systému, pretože užívateľ ich nemusí chcieť vždy využiť. Pomalá linka, udalosti ktoré neprišli v správnom poradí, nespôsobujú rizikové problémy v samotnom spracúvaní, ale znižujú presnosť výsledných dát. Užívateľ môže využiť spomínaných systémov na spresnenie výsledku svojej aplikácie.

Zoradenie udalostí nemusí byť iba pomocou časového razítka ich vzniku. Užívateľ môže definovať iné usporiadanie s ktorým sa systém musí nejako vysporiadať. Napríklad môže brať ako dôležitý čas príchodu udalosti ku agentovi, takže pomalé linky už nebudú spôsobovať problémy lebo z pohľadu agenta pôjdu po poradí.

Jednou s najväčších výziev pri spracúvaní udalostí je uloženie potencionálne neobmedzeného dátového prúdu do limitovanej hlavnej pamäte. Pokiaľ dotaz, ktorý sa vyhodnocuje s príchodom každej novej udalosti, trvá veľmi dlho, ďalšie udalosti sa pomaly nahromadia a dátový prúd môže narásť do enormných rozmerov. Inak povedané, ak nové udalosti stále prichádzajú, aj keď staré ešte nie sú spracované, hodnota výpočtu na jednu udalosť musí byť malá, inak bude latencia výpočtu moc veľká a algoritmus nebude stíhať s dátovým prúdom. Preto sa zaujímame o algoritmy ktoré sú spokojné s hlavnou pamäťou a nepristupujú na disk. Týmto sa dostávame ku ďalšiemu problému, ktorým sú kontinuálne dotazy. Tento problém som už naznačil v úvode kapitoly a je výnimočný pre spracúvanie dát. S každou novou udalosťou sa prepočíta výsledná hodnota algoritmu. Keďže s rýchlo prichádzajúcimi udalosťami rastie ich počet, výpočtová hodnota algoritmu sa predlžuje a môže sa preplniť pamäť. Aby sme urýchlili beh algoritmu, môžeme aproximovať odpoveď a tým znížiť presnosť výsledku, ktorá je väčšinou dostačujúca. Existujú rôzne spôsoby aproximácie, ktoré teraz postupne vysvetlím. [2]

Kĺzajúce okná

Jedna z techník na aproximáciu výsledku je nepočítať výsledok zo všetkých dát ale iba z blízkej minulosti. V reálnych aplikáciách sa tento druh aproximácie využíva veľmi často, pretože zastaralé dáta sú menej dôležité ako nové a preto môžu byť zanedbané. Týmto prístupom nie len že zredukujeme čas behu algoritmu, keďže bude počítať z obmedzeného počtu dát, ale aj vyriešime problém neobmedzeného prúdu dát.

SQL-podobné programovacie jazyky stačí rozšíriť a kľúčové slovo, ktoré vytvorí, spravuje a manipuluje s kĺzajúcim oknom. Napríklad v jazyku CQL [1], okno s veľkosťou 1 minúty, ktoré sa posúva po 10 sekundách vyzerá následovne:

```
SELECT Istream Count(*) FROM
C2XMessage [ Range 1 Minute Slide 10 s~]
WHERE
Speed > 30.0
```

Ku kĺzajúcim a iným druhom okien sa vrátim v ďalšej kapitole. [2]

Dávkové spracovanie

Tento spôsob sa hodí ak je kontinuálny dotaz pomalý a pridanie novej udalosti do prúdu je rýchle. Spočíva v tom, že prichádzajúce udalosti su buffrované, postupne dané do menších dávok, ktoré sú následne spracované kontinuálnym dotazom. Pokiaľ je kontinuálny dotaz vykonávaný, nové udalosti sa odkladajú do ďalšej dávky. Tento spôsob je výhodný, pretože nespôsobuje veľké nepresnosti v odpovedi, ale odpoveď dostaneme v neskoršom čase. Preto sa využíva v aplikáciách, ktoré očakávajú presnosť výsledku ale nevyžadujú instantnú odpoveď. [2]

Vzorkovanie

Vzorkovanie sa využíva ak kontinuálny dotaz prebehne za krátku dobu, ale pridanie novej udalosti trvá dlhšie ako priemerná doba príchodu ďalšej udalosti. Je nemožné aby boli využité všetky dáta, pretože prichádzajú skôr ako môžu byť spracované. Preto musí algoritmus počítať iba s niektorými udalosťami, čoho výsledkom je aproximácia výslednej hodnoty. Tento prístup môže v niektorých prípadoch(hlavne ak kontinuálny dotaz obsahuje operátor JOIN) poskytnúť veľmi slabú presnosť. Na algoritme, ktorý by poskytoval presnejšiu odpoveď sa stále pracuje. [2]

Ďalším problémom su blokujúce operátory, ktoré pre svoj výpočet musia poznať celý vstup. Napríklad operátor zoradenia alebo agregačné operátory ako SUM, COUNT, MIN, MAX alebo AVG. Keďže nie je možné aby operátor čakal na celý vstup pretože nemôže odhadnúť kedy je koniec, musí to systém pre spracúvanie dát nejako vyriešiť. Jedna možnosť je produkovať odpoveď po každej novej udalosti(ako to napríklad robí Esper). Celkovo je každá odpoveď aproximácia výslednej, čo je lepšie ako žiadna odpoveď. Ďalšou možnosťou je nahradzovať blokujúce operátory ich neblokujúcimi variantami, ktoré robia približne rovnakú činnosť.

V tejto kapitole som sa pozrel na rozdiely medzi systémom na spracúvanie udalostí a systémom na správu databáz. Ďalej priblížim problematiku kontextov používaných pri spracúvaní dát.

Kapitola 3

Kontexty

Niekedy potrebujeme rozlišovať podobné udalosti na základe nejakej hodnoty, napríklad času alebo miesta vzniku, alebo chceme počítať len s podmnožinou všetkých udalostí. Aby sme to nemuseli všetko písať ručne, stále dokola do agentov, boli vytvorené kontexty, ktoré sa využívajú najčastejšie. Môžeme mať aplikáciu, ktorá prijíma udalosti iba od 8 do 10 hodiny rannej. Takže jej nastavíme fixný temporálny kontext, ktorý trvá iba dané hodiny. Udalosti, ktoré prídu v iných hodinách budú ignorované, pretože nepatria do kontextu.

Na Obrázku 3.1 vidíme rôzne druhy kontextov aj s ich parametrami. V tejto kapitole sa budem zaoberať temporálnymi a priestorovými kontextami. O stavovo a segmentovo orientovaných kontextoch je viac informácií v [3, kapitola 7].

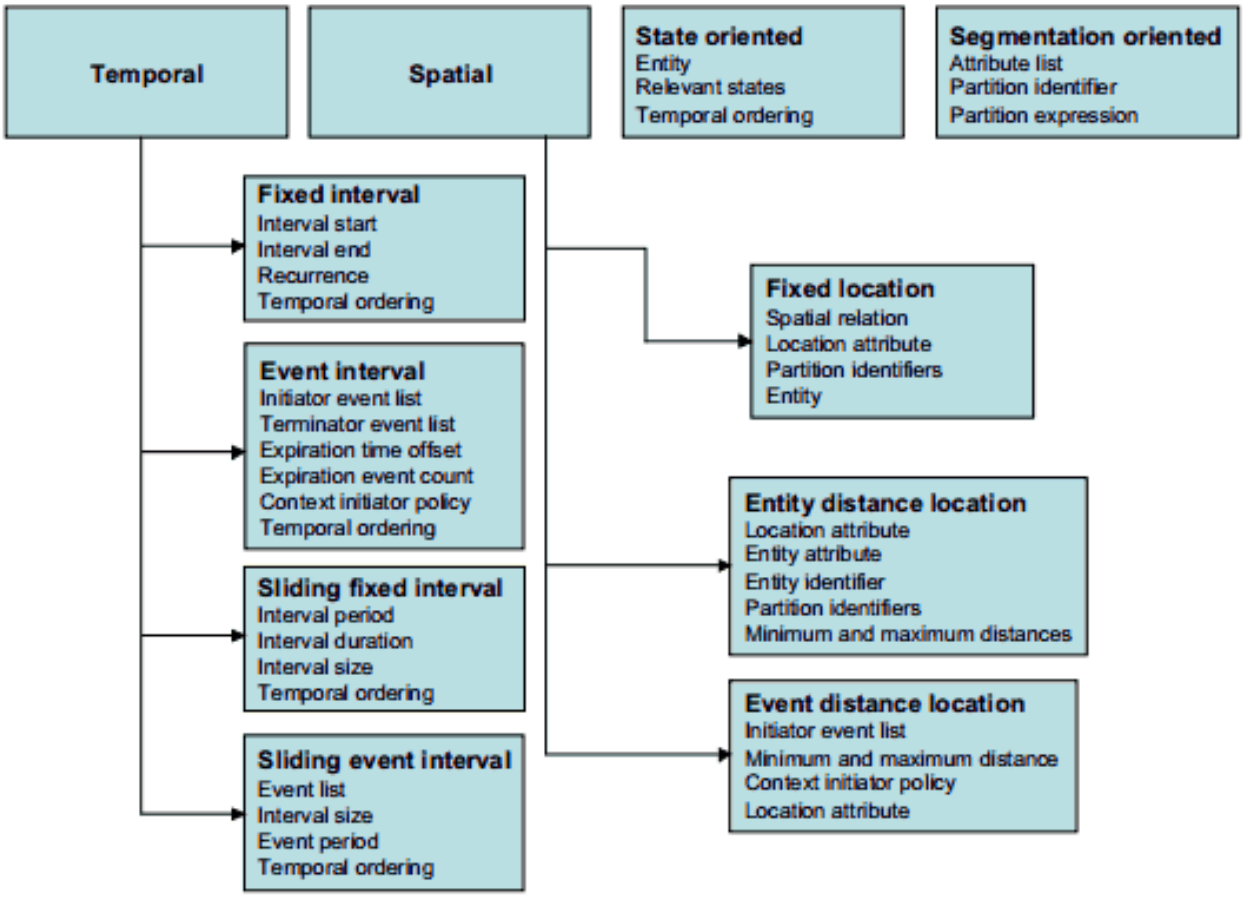
3.1 Temporálny kontext

Existujú 4 typy temporálnych kontextov, kde každý pracuje s rozdelením času na ohraničené intervaly, ktorým sa taktiež hovorí okná(windows). Ako vidíme na Obrázku 3.2, tieto okná môžu byť disjunktné(neprekrývajúce sa), kontinuálne(nadväzujúce na seba) alebo sa môžu prelínať. Ich začiatok a koniec môže byť vyvolaný buď určitým časovým okamihom, alebo konkrétnou udalosťou.

V tejto podkapitole sa budem zaoberať iba intervalmi, ktoré sa spúšťajú a končia časovým okamžikom, pretože intervaly, ktoré závisia na udalostiach pracujú analogicky. Obsahujú list typov udalostí, pri ktorých výskyte sa okno otvorí a list typov udalostí, pri ktorých(opakovanom, podľa nastavenia okna) výskyte sa okno zatvorí. Otvorenie nového okna záleží na počte udalostí, ktoré sa vyskytli v priebehu času kedy bolo okno otvorené. Pre spresnenie analógie medzi týmito dvoma druhmi intervalov, si môžeme predstaviť systém, kde sa vyskytne nová udalosť každú sekundu(sekunda je najmenšia časová jednotka, ktorú použijem), takže fixný časový interval veľkosti 2 minút bude rovnako veľký ako fixný interval udalostí, ktorý sa zatvorí po príchode 120 udalostí od jeho otvorenia.

3.1.1 Fixný interval

Fixný časový interval reprezentuje určitú časovú dobu, podľa ktorej sú udalosti rozdeľované do odlišných skupín, ktoré sú individuálne spracúvané. Môže to byť konštantná doba, bez opakovania, alebo sa môže opakovať. Atribúty tohoto kontextu sú následovné:



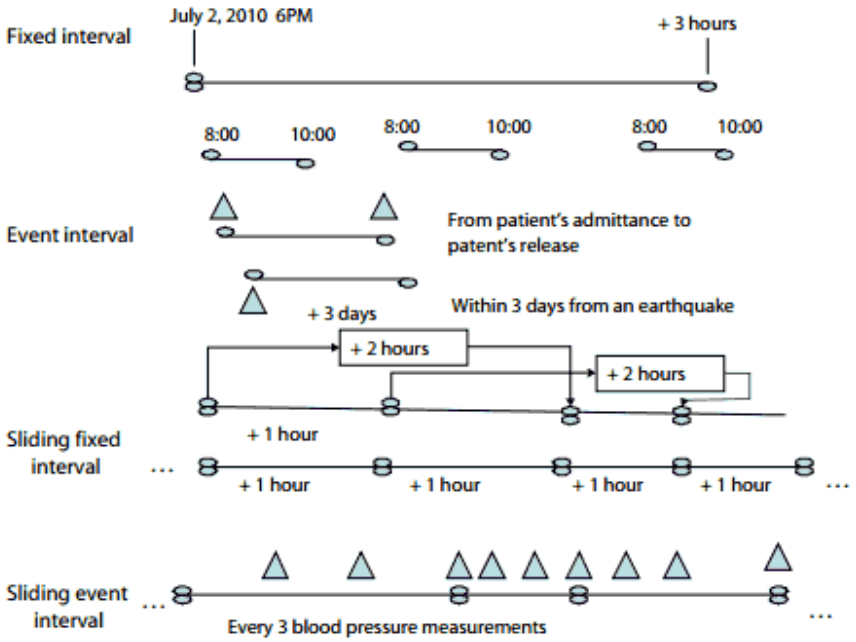
Obr. 3.1: Rozdielne typy kontextov ukazujúce svoje parametre. Tento diagram tiež ukazuje základné dimenzie priradené ku typom, napríklad typ fixná lokácia je braná ako priestorový kontext. [3]

Štart intervalu
: Reprezentuje časovú jednotku, kedy interval začne. Môže byť typu Dátum alebo DátumČas. Ak je dátového typu Dátum, interval začne o polnoci daného dátumu.

Koniec intervalu
: Reprezentuje časovú jednotku, kedy prvý interval skončí, alebo obsahuje dĺžku intervalu. Môže byť typu Dátum alebo DátumČas. Ak je dátového typu Dátum, interval skončí o polnoci daného dátumu.

Rekurzia
: Reprezentuje ako často, a či vôbec, sa interval opakuje.

Temporálne zoradenie
: Tento atribút určuje, ktorá časová stopa sa má použiť(či čas detegovania alebo výskytu).



Obr. 3.2: Rozdielne typy temporálnych kontextov s ich kontextovými segmentami. [3]

Na Obrázku 3.2 vidíme 2 fixné časové intervaly. Prvý môže znázorňovať aukčnú aplikáciu, kde aukcia začne 2 Júla 2010 o 18 hodine a trvá 3 hodiny. Akceptujú sa iba udalosti v tomto intervale. Druhý interval môže reprezentovať aplikáciu, ktorá skenuje trh každý pracovný deň od 8 do 10 hodiny rannej.

Taktiež sa to dá predstaviť ako polootvorený interval $[T_s, T_e)$ kde udalosť s časovým razítkom τ patrí do intervalu ak $T_s \leq \tau < T_e$.

3.1.2 Kĺzajúci interval

Kĺzajúci časový interval reprezentuje, laicky povedané, pohybujúci sa fixný časový interval. Taktiež rozdeľuje udalosti do rozdielnych skupín, akurát tento interval sa môže aj prekrývať, takže jedna udalosť môže patriť do viacerých intervalov zároveň. Atribúty tohoto kontextu sú nasledovné:

Perióda intervalu
: Reprezentuje časový úsek od otvorenia okna, po ktorom bude otvorené okno nové.

Trvanie intervalu
: Časový úsek, za ktorý ostáva okno otvorené.

Veľkosť intervalu
: Maximálny počet udalostí, ktorý ak je prekročený, okno sa zavrie aj keď ešte nevypršalo trvanie intervalu.

Temporálne zoradenie

Tento atribút určuje, ktorá časová stopa sa má použiť(či čas detegovania alebo výskytu).

Ako je vidieť na Obrázku 3.2, prvý kĺzajúci interval trvá 2 hodiny, a nové okno je otvorené každú hodinu t.j. perióda = 1 hodina a trvanie = 2 hodiny. Tu je viditeľné že ak je perióda intervalu menšia ako jeho trvanie, intervaly sa prelínajú. Druhý príklad je príklad kontinuálneho intervalu kde sa perióda a trvanie rovnajú. Posledný, na obrázku neznázornený interval je keď je perióda väčšia ako trvanie. Vtedy nie je intervalmi pokrytá celá časová os.

Rovnako ako pri fixnom časovom intervale sa to dá predstaviť ako polootvorený interval $[T_s, T_e)$ takže udalosť s časovým razítkom τ patrí do intervalu ak $T_s \leq \tau < T_e$.

V programovacom jazyku Esper, s ktorým som pracoval, som sa stretol s implementáciou kĺzajúceho okno, ktorá spájala časové okno s oknom riadeným udalosťami. Funguje to tak, že s každou novou udalosťou sa ako keby zavrie fiktívne, dopredu neotvorené okno. To znamená že s každou novou udalosťou s časovým razítkom τ sú v minútovom okne všetky udalostí, ktoré majú časové razítko väčšie alebo rovné $\tau - 60$ sec.

3.2 Priestorový kontext

Priestorový kontext rozdeľuje udalosti podľa ich geografickej lokácie. Aby sme mohli zistiť kde udalosť vznikla, musí obsahovať atribúty reprezentujúce jej geografickú lokáciu, napríklad bod by obsahoval geografickú šírku a výšku. Miesto vzniku udalosti môže byť taktiež reprezentované názvom nejakej geografickej entity, napríklad názov mesta.

Na správne určenie miesta a zaradenie do správneho kontextu potrebujeme externú databázu podľa ktorej sa atribúty preložia do geografických súradníc, takémuto postupu sa hovorí *geocoding*.

Udalosť nemusí reprezentovať len bod v 2-D priestore, môže to byť aj priamka/úsečka, napríklad cesta, ktorá vytvorila udalosť *zápcha*. Môže to byť taktiež aj plocha, príde udalosť že vybuchla bomba a jej umiestnenie bude area výbuchu.

Existujú 3 typy priestorového kontextu. Fixná lokácia(Fixed location), Vzdialenosť od entity (Entity distance location) a Vzdialenosť od udalosti(Event distance location). Každý jeden následne povrchne rozoberiem.

Fixná lokácia

Tento kontext využijeme ak nás zaujímajú udalosti len z nejakého miesta, napríklad chceme prijímať len udalosti, ktoré vznikli v Amerike. Existuje viac typov relácií, ktoré môže vzniknutá udalosť zdielať s definovanou a to: *je obsiahnutá, obsahuje, presahuje, prelína sa, rovná sa* alebo *dotýka sa*. Myslím že ich význam je zrejmý akurát dodám že je to brané z pohľadu entity, takže entita je obsiahnutá v udalosti, entita obsahuje udalosť, atd.

Vzdialenosť od entity

Tento kontext rozdeľuje udalosti podla ich vzdialenosti od určitej entity. Teoreticky vytvára nekonečne veľa rozdelení, keďže vzdialenosť je jednotka dĺžky a každá dĺžka sa dá rozdeliť na nekonečne veľa častí, ale prakticky máme nejakú najmenšiu jednotku dĺžky,

definujme si ju ako meter, tak potom vzdialenosť od entity veľkosti 1 km, rozdelí danú oblasť na 1000 rozdelení.

Vzdialenosť od udalosti

Tento kontext pracuje podobne ako *Vzdialenosť od entity* ale navyše má list udalostí, od ktorých sa vzdialenosť merá. Ak sa niekde vyskytne udalosť, ktorá sa nachádza na liste, tak sa vytvárajú rozdelenia na základe vzdialenosti od spomenutej udalosti.

Kapitola 4

Príkazy

Už som spomenul celú teóriu spracúvania udalostí, teraz sa na to pozriem z viac praktickej časti. Budem hovoriť konkrétne o programovacom jazyku Esper, ktorého výkon a správanie operátorov som premeral. Esper je open-source softvér pod licenciou GNU General Public Licence(GPU) [6], ktorého syntax sa podobá, je rozšírením, SQL jazyka. Operátory pracujú podobne aj v ostatných programovacích jazykoch.

Operátor v spracúvaní dát, je funkcia, ktorá zoberie žiadnu, jednu alebo viac udalostí na vstupe a vráti jednu alebo viac nových udalostí, číslo alebo iný dátový typ záležiac od operátora. Taktiež nemusí vrátiť nič. Operátory sú väčšinou úzko späté s kontextami, pretože dostávajú na vstup všetky udalosti, ktoré patria do daného kontextu, nie je to však podmienka.

Aby som to spojil s teóriou, funkčnosť všetkých agentov sa dá zapísať pomocou príkazov, napríklad zo spomínaného Esperu. Pre lepšiu predstavu, filtrovací agent, ktorému sú posielané udalosti typu *Prihlásenie užívateľa*, odosiela ďalej iba udalosti, ktorých prihlásenie bolo neúspešné(atribút success je nepravda). Dotaz, ktorý vykonáva takúto úlohu môže vyzerať napríklad takto:

```
INSERT INTO ďalejSpracúvanýPrúd
SELECT * FROM
prúdPrihlásenieUžívateľov
WHERE success = false;
```

Pri tomto dotaze som syntakticky využil iba kľúčové slová SQL jazyka. Ak by v databáze existovala tabuľka s názvom *ďalejSpracúvanýPrúd* a *prúdPrihlásenieUžívateľov* tak by tento dotaz prešiel aj v databázovom systéme. Rozdiel je v tom, že v databáze by sa vykonal raz a skončil by, no v svete spracúvania dát je to kontinuálny dotaz, ktorý je priradený prúdu *prúdPrihlásenieUžívateľov*. Takže pri každej novej udalosti v tomto prúde sa hneď vyhodnotí či ju poslať ďalej do prúdu *ďalejSpracúvanýPrúd* alebo zahodiť.

Meranie časových hodnôt prebieha offline. Pomocou javovského Random [1] si vygenerujem náhodné listy udalosti(so semienkom 1, 2 a 3). Postupne tieto udalosti posielam agentovi na spracovanie. Agent obsahuje vždy jeden dotaz na konkrétny operátor. Z každého merania mám jednu časovú hodnotu v milisekundách, z ktorých vypočítam aritmetický priemer a výsledok zapíšem do tabuľky, z ktorej vykreslím graf pomocou online nástroju [10].

V tejto kapitole sa budem držať nasledujúcich označení. Pole, ktoré vygenerujem má značku EL a dĺžku n. Funkcia, ktorá vráti udalosť z pola môže byť buď $EL(i)$, kde i je číslo(index, poradie udalosti v liste), alebo $EL(E)$, kde E je udalosť, ktorú chceme z pola dostať na výstup.

1. http://docs.oracle.com/javase/6/docs/api/java/util/Random.html

Taktiež, niektoré operátory očakávajú na vstup atribút každej udalosti v EL, toto budem značiť ako $EL(A)$. Udalosť má atribúty $timestamp$, ktorý je dátového typu long [2] a reprezentuje vznik udalosti, časové razítko, ďalej obsahuje atribút $test$, ktorý je taktiež dátového typu long a podla tohoto atribútu sa bude testovať. Posledný atribút je D, ktorý je dátového typu reťazec(string), ale obsahuje len jedno písmeno, buď l alebo r. Tento atribút bol pridaný z dôvodu operátoru JOIN a rozhoduje do ktorého prúdu dát má byť udalosť poslaná. Takže udalosť budem formálne zapisovať ako $E(\tau, t, d)$, kde τ je časové razítko, t je hodnota atribútu test a d je hodnota atribútu D. Každý atribút dostanem funkciou $E(A)$. Kontext, v ktorom bude operátor vyhodnocovaný, bude prevažne kĺzajúce okno, ktoré rozdelí vstup do m rôznych nedisjunktných častí.

Generovanie udalostí prebieha tak, že každá 250-ta udalosť má o 1 ms väčší $timestamp$ ako predchádzajúca, takže mám virtuálne prostredie kde sa za sekundu vytvoria 250 000 udalosti. Funkcia, ktorá toto pole generuje, dostane na vstup informácie *počet udalostí*, *rozsah testu* a *semienko*. *Počet udalostí* je zrejmé čo znamená, podla *semienka* sa náhodne generuje a *rozsah testu* obsahuje číslo, ktoré reprezentuje maximálnu hodnotu atribútu test. Takže napríklad keby má hodnotu 6, atribút test obsahuje hodnoty z intervalu $\langle 0, 5\rangle$. Atribút je vždy, pokiaľ nie je explicitne povedané inak, generovaný náhodne, pomocou spomínaného semienka. Atribút D, ktorý využijem iba pri operátore JOIN, sa generuje striedavo, t.j. každá párna udalosť má hodnotu „r“ a každá nepárna hodnotu „l“.

Najviac vypovedajúca hodnota bude väčšinou pri počte udalostí 2 000 000, pretože je to jediné množstvo, ktoré je väčšie ako 10 sec okno.

4.1 Agregačné operátory

Niekedy môže byť centrum záujmu jednoduchá matematická operácia, ktorá ako vstup dostane nejaký atribút viacerých logicky rovnakých udalostí, väčšinou číslo, a ako výsledok vráti číslo, ktoré je nejako odvodené zo vstupu. Príkladom môže byť funkcia MAX, ktorá zoberie na vstup nejaký číselný atribút udalostí a vráti najväčšiu nameranú hodnotu. Jedno z využití agregačných operátorov môže byť pri štatistických úlohách. Môže nás zaujímať priemerný počet prihlásených užívateľov za deň v hodinách. Tak vytvoríme kĺzajúce okno s periódou rovnou trvaniu intervalu, ktorá bude mať dĺžku 1 hodinu. Vždy keď sa okno uzavrie, spočítame počet prihlásení a pošleme túto informáciu ako udalosť ďalšiemu agentovi, ktorý z nich bude počítať priemernú hodnotu raz za deň. Môže to vyzerať následovne:

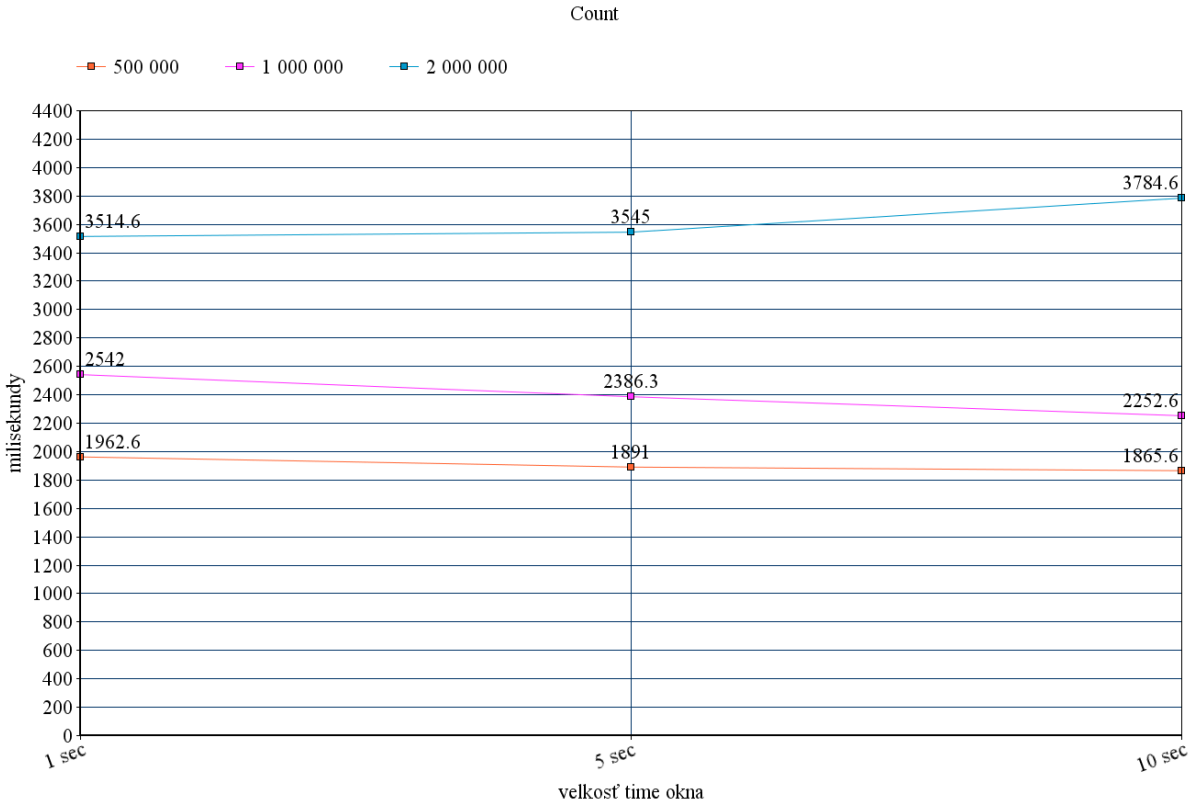
```
INSERT INTO prúdPočtuPrihlásení
SELECT COUNT(*) AS pocet FROM
prúdPrihásenіUžívateľov.win:time_batch(1 hod);
SELECT AVG(pocet) FROM
prúdPočtuPrihlásení.win:time_batch(24 hod);
```

Syntax <názovprúdu>.win:time_batch(časový úsek) vytvorí spomínané kĺzajúce okno s rovnakým trvaním a periódou intervalu dĺžky časového úseku. Teraz sa presuniem konkrétnejšie na jednotlivé agregačné operátory postupne.

2. http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

4.1.1 COUNT

Jednoduchá operácia, ktorá reprezentuje funkciu, ktorá na vstupe očakáva EL a vráti jeho dĺžku n.



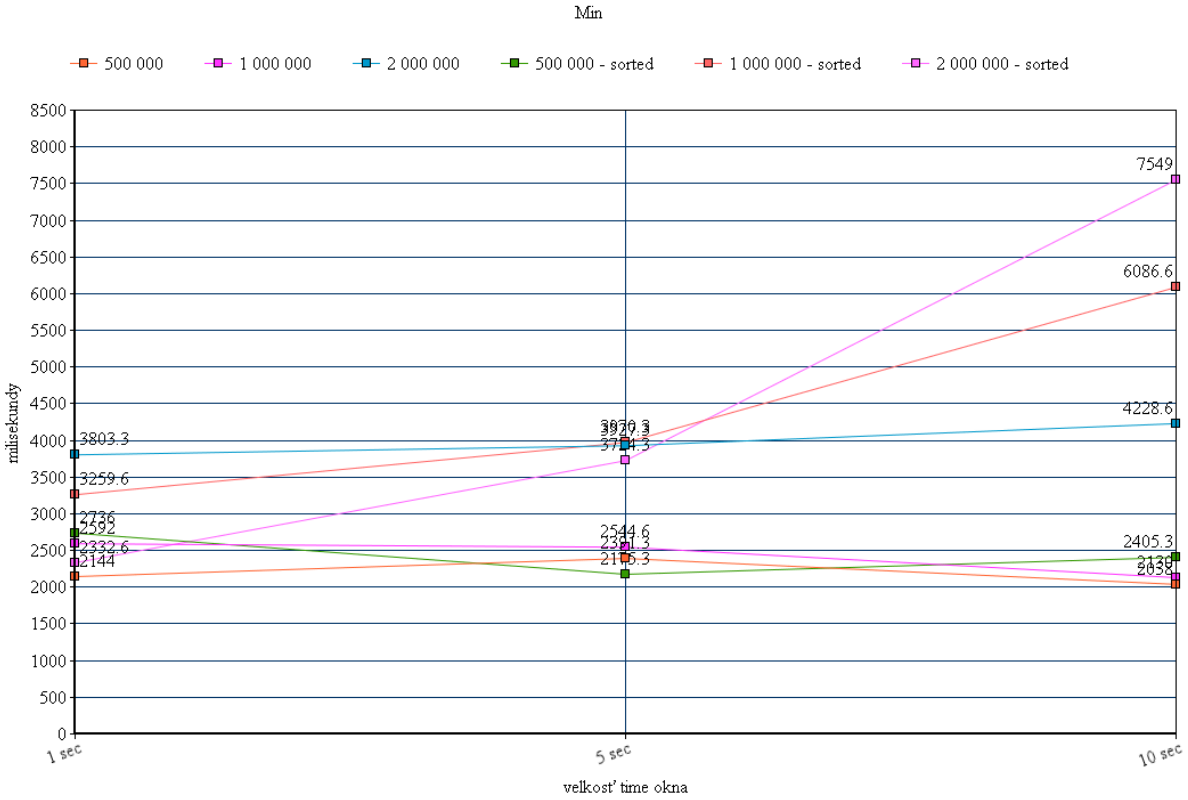
Obr. 4.1: Tento graf reprezentuje chovanie agregačného operátora COUNT pri rozdielnych veľkostiach veľkostí kĺzajúceho okna.

Ako môžeme vidieť z grafu na Obrázku 4.1, zmena veľkosti okna žiadno výrazne nezmení rýchlosť výpočtu tohoto operátora. Všetky výraznejšie zmeny sú spôsobené nejakým prerušením procesu alebo iným vedľajším efektom. Je viditeľné menšie spomalenie pri 10 sec okne, ale to je pochopiteľné, keďže operátor dostáva naraz väčší vstup.

Využitie tohoto operátora je veľmi rozsiahle. Existuje veľa aplikácií, ktoré na svoju prácu potrebujú mať vypočítaný počet niečoho, alebo dĺžku pola. Konkrétny príklad som spomenul už v úvode tento podkapitoly.

4.1.2 MIN

Minimum je agregačný operátor, ktorý reprezentuje funkciu, ktorá na vstupe očakáva $EL(A)$, ktoré sú typu číslo a vráti číselný atribút udalosti $E(A)$, pre ktorý bude platiť že $\forall x \in EL(A).E(A) \leq x$.



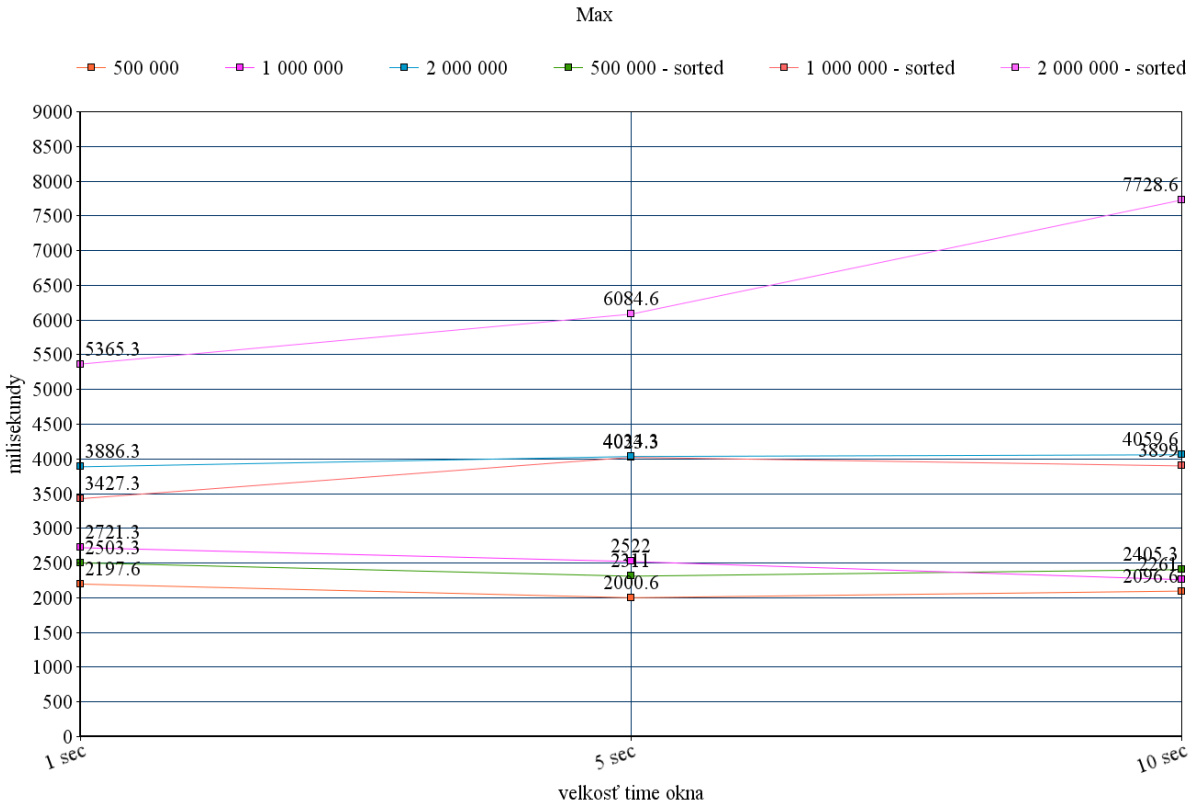
Obr. 4.2: Tento graf reprezentuje chovanie agregačného operátora MIN pri rozdielnych veľkostiach veľkostí kĺzajúceho okna. Sorted(zoradené) znamená že atribút udalostí test je klesajúci.

Ako môžeme vidieť z grafu na Obrázku 4.2, zmena veľkosti okna žiadno výrazne nezmení rýchlosť výpočtu tohoto operátora, až na zoradený vstup. Všetky výraznejšie zmeny sú spôsobené nejakým prerušením procesu alebo iným vedľajším efektom. Samozrejme že zoradený vstup sa bude so zväčšovaním okna spomaľovať pretože každá nová udalosť je nové minimum a všetko sa musí prepočítavať.

Využitie tohoto operátora je si myslím celkom zrejmé. Napríklad chceme pozorovať cenu notebooku, tak prijímame udalosti, ktoré sú vyvolané zmenou ceny vybratého modelu. Nakoniec sa pozrieme kedy bol najlacnejší, kedy sa ho oplatilo kúpiť.

4.1.3 MAX

Maximum je agregačný operátor, ktorý reprezentuje funkciu, ktorá na vstupe očakáva $EL(A)$, ktoré sú typu číslo a vráti číselný atribút udalosti $E(A)$, pre ktorý bude platiť že $\forall x \in EL(A).E(A) \geq x$.



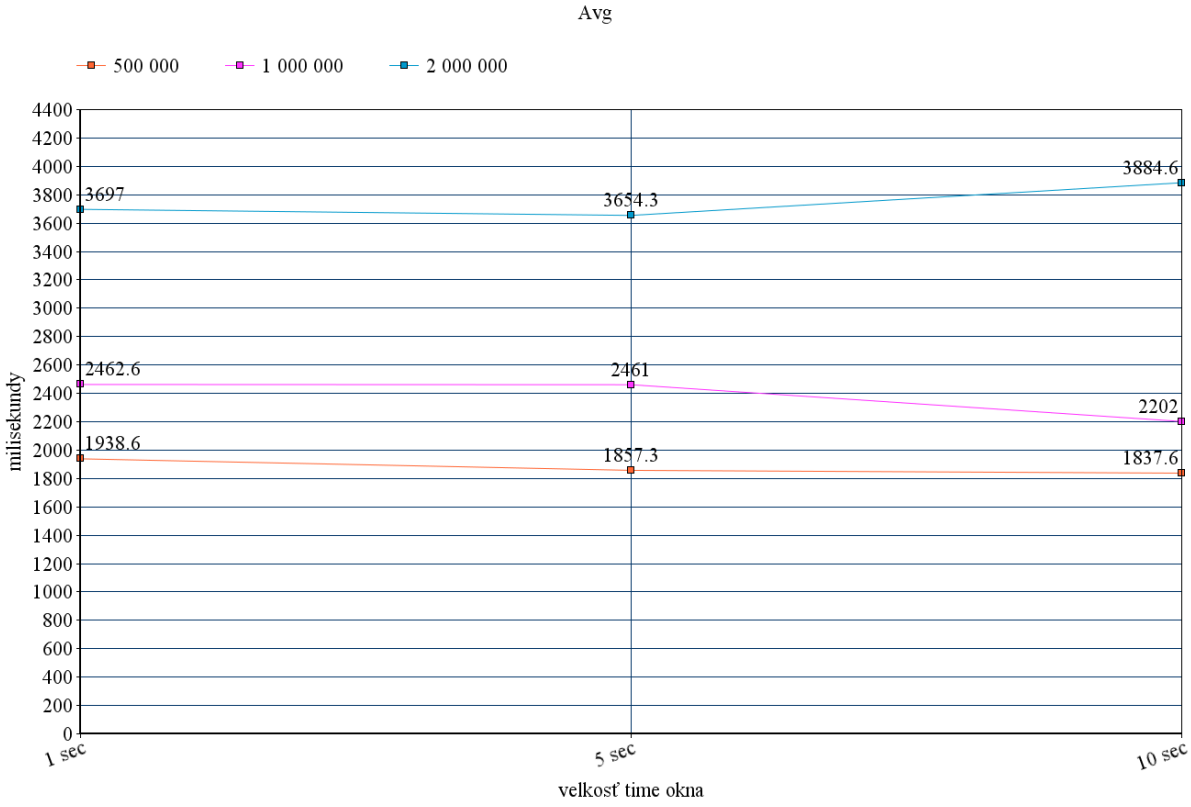
Obr. 4.3: Tento graf reprezentuje chovanie agregačného operátora MAX pri rozdielnych veľkostiach veľkostí kĺzajúceho okna. Sorted(zoradené) znamená že atribút udalostí test je rastúci.

Ako môžeme vidieť z grafu na Obrázku 4.3, zmena veľkosti okna žiadno výrazne nezmení rýchlosť výpočtu tohoto operátora, až na zoradený vstup. Všetky výraznejšie zmeny sú spôsobené nejakým prerušením procesu alebo iným vedľajším efektom. Čo je ale zaujímavé je napríklad porovnanie 2 000 000 náhodne vygenerovaných udalostí a 1 000 000 usporiadaných udalostí, sú skoro rovnaké a pritom jedno je dvojnásobkom druhého.

Využitie tohoto operátora je si myslím celkom zrejmé. Podobne ako pri MIN akurát si nakoniec pozrieme najväčšiu hodnotu.

4.1.4 AVG

Priemerná hodnota je agregačný operátor, ktorý reprezentuje funkciu, ktorá na vstupe očakáva $EL(A)$, ktoré sú typu číslo a vráti číslo x, pre ktoré platí že $x = \frac{\sum_{i=0}^{n} EL(i)(A)}{n}$.



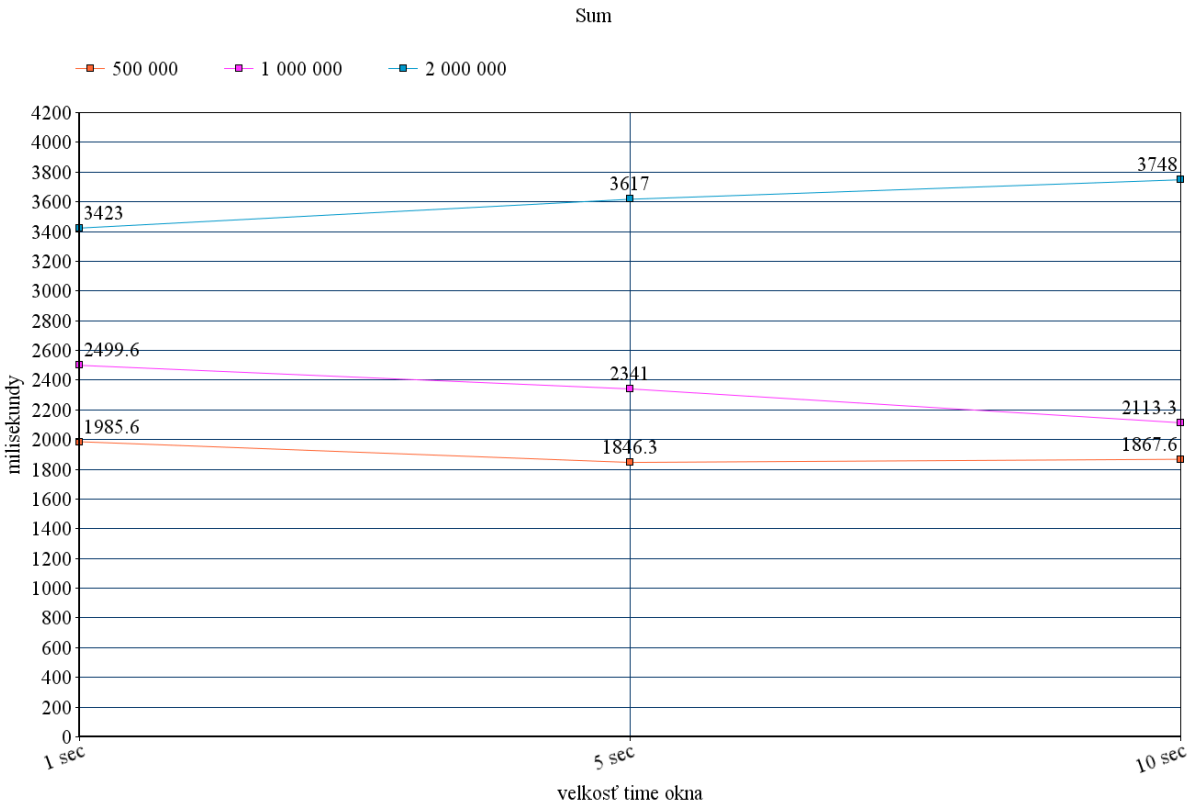
Obr. 4.4: Tento graf reprezentuje chovanie agregačného operátora AVG pri rozdielnych veľkostiach veľkostí kĺzajúceho okna.

Ako môžeme vidieť z grafu na Obrázku 4.4, zmena veľkosti okna žiadno výrazne nezmení rýchlosť výpočtu tohoto operátora. Všetky výraznejšie zmeny sú spôsobené nejakým prerušením procesu alebo iným vedľajším efektom. Keďže zmena rýchlosti nie je tak výrazná ako som predpokladal, tak z toho usudzujem že Esper si po skupinkách predpočítava priemerné hodnoty a potom s nimi pracuje ďalej. Pri menšom okne mu to predpočítavanie moc nefunguje, pretože v okne sú stále iné udalosti. Ale pri väčšom okne, kde sa zmestí 250 000 udalostí, využije týchto výsledkov na zrýchlenie výpočtu.

Využitie tohoto operátora ocenia hlavne ľudia, ktorý robia štatistiky. Praktický príklad som uviedol v úvode podkapitoly.

4.1.5 SUM

Suma je agregačný operátor, ktorý reprezentuje funkciu, ktorá na vstupe očakáva $EL(A)$, ktoré sú typu číslo a vráti číslo x, pre ktoré platí že $x = \sum_{i=0}^{n} EL(i)(A)$.



Obr. 4.5: Tento graf reprezentuje chovanie agregačného operátora SUM pri rozdielnych veľkostiach veľkostí kĺzajúceho okna.

Ako môžeme vidieť z grafu na Obrázku 4.5, zmena veľkosti okna žiadno výrazne nezmení rýchlosť výpočtu tohoto operátora. Všetky výraznejšie zmeny sú spôsobené nejakým prerušením procesu alebo iným vedľajším efektom. Ako je vidieť pri najväčšom vstupe, s veľkosťou okna rastie aj doba trvania algoritmu. Nie je to práve veľký rozdiel ale je opodstatnený rozdielom veľkosti vstupu(počet udalostí v okne).

Využitie tohoto operátora, ako aj ostatných môžeme využiť napríklad v štatistike. Vypočítame si celkový súčet všetkých udalostí a pošleme ďalšiemu agentovi na spracúvanie.

4.2 GROUP BY

Tento operátor sa skoro vždy využíva so spomenutými agregačnými operátormi, dá sa použiť aj bez nich ale výsledok je potom mätúci. Výsledok vyzerá rovnako ako v predchádzajúcej kapitole, akurát operátor GROUP BY udalosti rozdelí do viac skupín/grúp podľa nejakého atribútu.

Syntax tohoto operátoru vyzerá následovne [4]:

```
GROUP BY aggregate_free_expression [, aggregate_free_ex-
pression] [, ...]
```

Ako je vidieť zo syntaxe, môžeme vytvárať grupy pomocou viacerých atribútov. Nemusí to byť iba atribút udalosti, taktiež to môže byť výsledok nejakého výrazu, napríklad súčin dvoch atribútov.

Všeobecne, tento operátor reprezentuje funkcia, ktorá na vstup zoberie EL a ako výstup vráti udalosti rozdelené do $g \in N$ grúp. Ak sa grupuje podľa jedného atribútu A_1, $g = DISTINCT(A_1, EL)$, kde funkcia $DISTINCT(A_1, EL)$ vráti veľkosť množiny, obsahujúcej všetky hodnoty atribútu A_1 z listu udalostí EL(čiže počet rôznych hodnôt atribútu A_1 v EL). Ak sa grupuje podla viacerých atribútov(A_1, A_2), tak sa každá grupa rozdelí do $h \in N$ podgrúp, kde $h = DISTINCT(A_2, G_i)$. Takže počet grúp je $\sum_{i=0}^{g} DISTINCT(A_2, G_i)$, kde G_i je grupa pre hodnotu atribútu rovnú i. Do grupy G_i patria udalosti, ktorých atribút A je rovný hodnote i(majú vlastnosť i).

Napríklad ak hodnoty atribútu sú [$E_1(1)$, $E_2(2)$, $E_3(1)$, $E_4(3)$, $E_5(5)$, $E_6(1)$, $E_7(2)$, $E_8(4)$], tak počet grúp je 5 a prvá grupa obsahuje [E_1, E_3, E_6], druhá [E_2, E_7], tretia [E_4], štvrtá [E_8] a piata [E_5].

Ako som spomínal, operátor GROUP BY sa používa s agregačnými operátormi, takže výsledok nie je g grúp, ale Esper pre každú grupu vygeneruje jednu udalosť. Napríklad výraz

```
SELECT symbol, SUM(price) FROM StockTickEvent.win:
time(30 sec) GROUP BY symbol
```

pre každý symbol vygeneruje udalosť obsahujúcu hodnotu symbolu a súčet cien za 30 sekúnd.

Esper vyžaduje nasledujúce omedzenia pre operátor GROUP BY:[4]

- výraz v GROUP BY nesmie obsahovať agregačné funkcie
- atribúty udalostí, ktoré sú použité ako vstup pre agregačnú funkciu v SELECT klauzule, nesmú byť použité v GROUP BY výraze
- Keď grupujeme neohraničený prúd, t.j. nie je zadaný žiaden kontext, mali by sme sa uistiť že GROUP BY výraz nevráti neobmedzené množstvo grúp. Napríklad keby sa grupovalo podľa časového razítka(ktoré je teoreticky pre každú udalosť rôzne), výrazne by to zasiahlo voľnú pamäť.

Esper podporuje keď jeden alebo viac atribútov v GROUP BY výraze nie je spomenutých v SELECT liste. Nasledujúca ukážka počíta štandardnú odchýlku pre každý $symbol$ a $tickDataFeed$ a generuje jednu udalosť pre grupu obsahujúcu $symbol$ a štandardnú odchýlku ceny za 30 sekúnd. Keď sa $tickDataFeed$ nenachádza vo výsledku, môže to byť potencionálne mätúce [4].

```
select symbol, stddev(price)
from StockTickEvent.win:time(30 sec)
group by symbol, tickDataFeed
```

Môžeme taktiež skombinovať WHERE klauzulu s GROUP BY. Funguje to tak že sa najskôr odfiltrujú udalosti, pri ktorých sa podmienka vyhodnotí na $false$ a ostatné sú spracúvané operátorom GROUP BY.

Otestoval som funkčnosť tohoto operátora v Esperi, či odpovedá definícii a naozaj vytvára grupy ako má. Výsledok je trochu rozdielny od databázy ale spĺňa definíciu.

Pre vstup [$E_1(0, 2, l)$, $E_2(250, 2, l)$, $E_3(500, 1, l)$, $E_4(750, 1, l)$, $E_5(1000, 1, l)$, $E_6(1250, 1, l)$, $E_7(1500, 1, l)$, $E_8(1750, 1, l)$, $E_9(2000, 1, l)$] a výraz **SELECT** timestamp, **COUNT**(*), test **FROM** Test.win:time(1 sec) **GROUP BY** test; je vygenerovaný výsledok následovný(rovnaký výsledok dostaneme aj pre okno .win:length(4), pretože odpovedá časovému oknu):

prišla udalosť	časové razítko	počet	test
E_1	0	1	2
E_2	250	2	2
E_3	500	1	1
E_4	750	2	1
E_5	1000	3	1
E_6	1250	4	1
E_7	1500	4	1
E_8	1750	4	1
E_9	2000	4	1

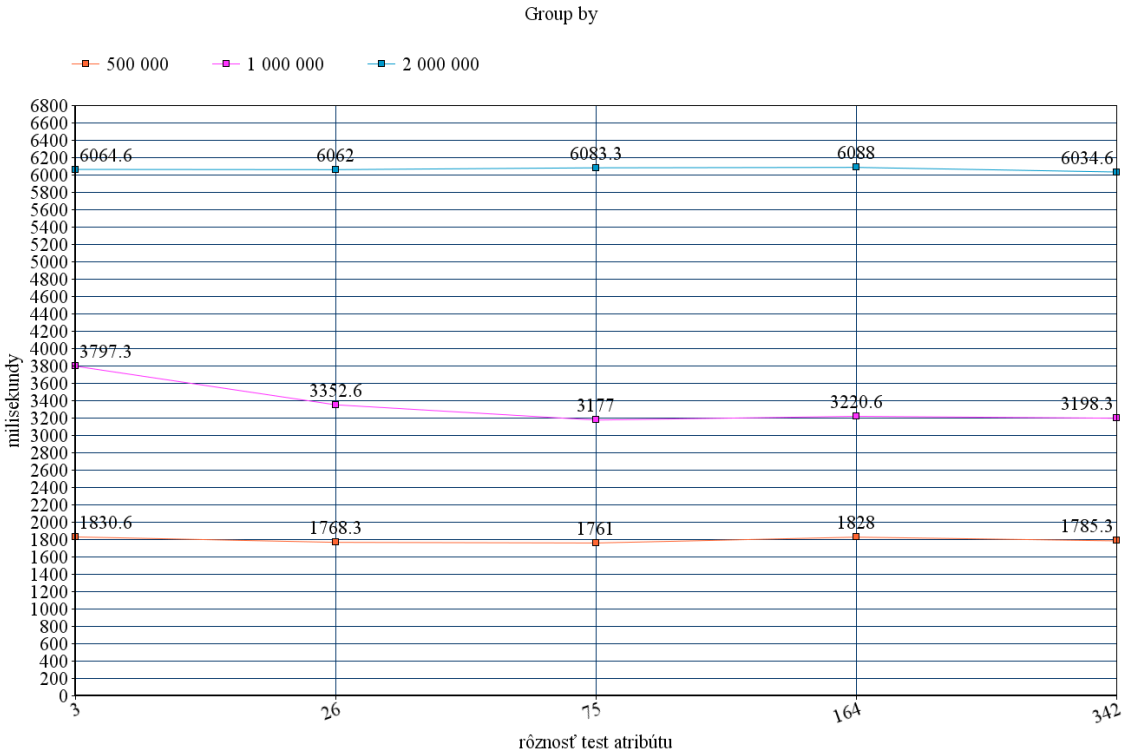
Z tabuľky je krásne vidieť že Esper pri novej udalosti posunie okno o jednu udalosť, pozrie sa na hodnotu atribútu podla ktorého grupuje a vytvorí novú udalosť, ktorej atribúty sú v **SELECT** liste. V tomto prípade vytvorí udalosť s atribútmi timestamp(časové razítko udalosti, ktorá vyvolala grupovanie), počet(počet udalostí v grupe) a test(hodnota atribútu podla ktorého sa grupovalo). Vyskytla sa udalosť E_1, Esper ju zaradí do skupiny $test = 2$, spočíta všetky udalosti v tejto skupine a vytvorí udalosť. Príde udalosť E_2, Esper ju zaradí do tej istej skupiny ako predchádzajúcu a vytvorí udalosť že už sú tam 2. Posun okna je viditeľný pri udalostiach E_6 až E_9, kde počet udalostí v grupe ostáva rovný 4.

Ďalší príklad je podobný ako predchádzajúci, akurát sa zmení kontext. Vstup ostáva nezmenený a výraz, ktorý sa vyhodnocuje vyzerá následovne: **SELECT** timestamp, **COUNT**(*), test **FROM** Test.win:time_batch(1 sec) **GROUP BY** test;

prišla udalosť	časové razítko	počet	test
E_1	0	2	2
E_2	250	2	2
E_3	500	2	1
E_4	750	2	1
E_5	1000	4	1
E_6	1250	4	1
E_7	1500	4	1
E_8	1750	4	1

Tabuľka vyzerá podobne ako v predchádzajúcom príklade, akurát rozdiel je v tom, že prvé 4 udalosti sa vygenerujú až keď príde E_5. Je vidieť že E_9 sa nenachádza vo výslednej tabuľke, ale keby nebola vo vstupe, nevygenerujú sa spodné 4 udalosti, pretože by okno nebolo „uzavreté“. Esper nemá odkiaľ vedieť či už má okno zatvoriť a vygenerovať výsledok, pokiaľ nepríde udalosť, ktorá patrí do ďalšieho okna, ktorej príchodom sa uzavrie okno a otvorí ďalšie.

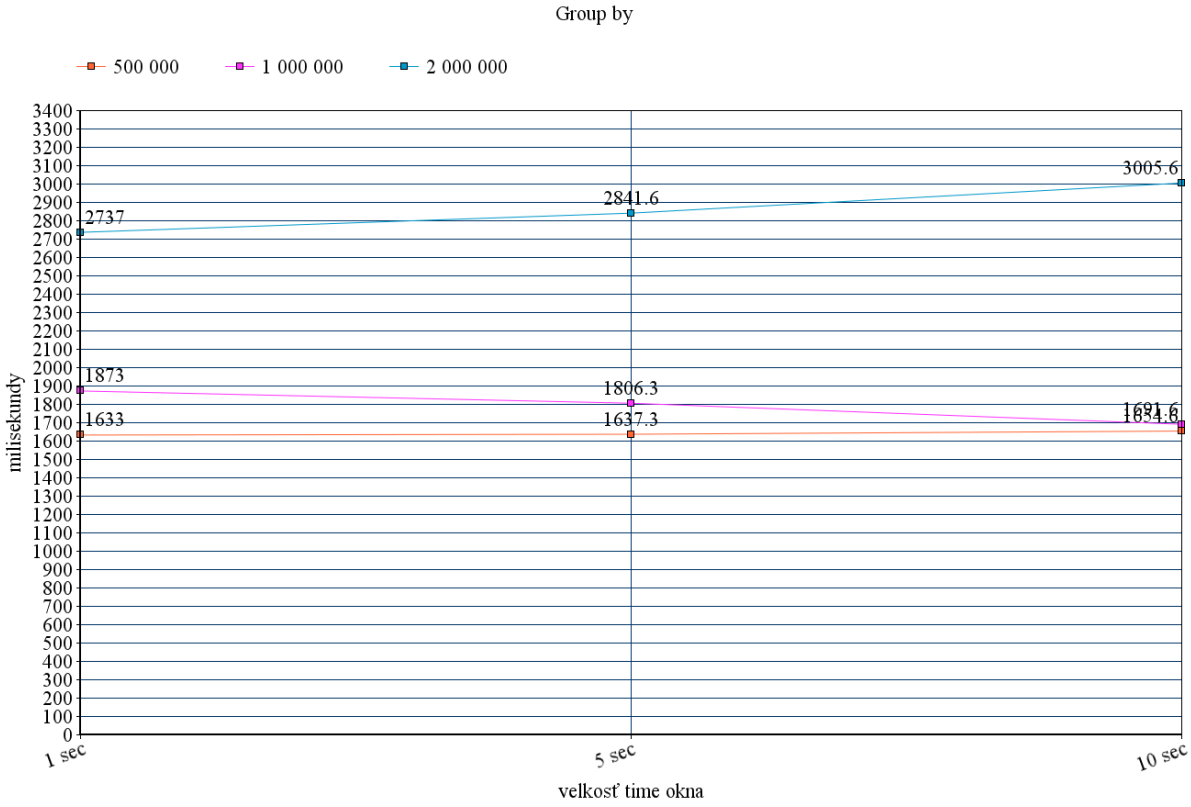
Čo sa týka rýchlosti tohoto operátoru, premeral som rozdiely rýchlostí pri rôznom počte vytvorených grúp a pri rôznych veľkostiach okna.



Obr. 4.6: Tento graf reprezentuje chovanie operátora GROUP BY pri rozdielnych počtoch grúp.

Graf na Obrázku 4.6 ukazuje rýchlosť spracovania udalostí pomocou operátoru *GROUP BY*. Je vidieť že moc nezáleží na rôznorodosti atribútu podľa ktorého sa vytvárajú grupy. Počet grúp ovplyvní pamäťovú náročnosť, nie časovú.

Na grafe na Obrázku 4.7 je vidieť že ani zmena veľkosti okna moc neovplyvňuje rýchlosť výpočtu algoritmu pre tento operátor. Ale každopádne je operátor pomalší pri väčšej veľkosti okna.



Obr. 4.7: Tento graf reprezentuje chovanie operátora GROUP BY pri rozdielnych veľkostiach veľkostí kĺzajúceho okna.

4.3 JOIN

Môže nastať situácia, kedy je potrebné spojiť viac prúdov do jedného. Napríklad keď máme veľa producentov alebo podobné prúdy od rôznych zdrojov, tak chceme tieto informácie uchovávať v jednej udalosti a tým zjednodušiť jej ďalšie spracúvanie.

Implicitne sa používa inner join, ktorý produkuje výslednú udalosť iba ak obidva prúdy obsahujú udalosť. To znamená že ak máme takýto výraz

```
select *
from TestLeft.win:time(30 sec),
TestRight.win:time_batch(1 sec)
```

tak výsledná udalosť je vygenerovaná iba ak v intervale 30 sekúnd sa vytvorí udalosť v prúde *TestLeft* a zároveň sa vyskytne udalosť v prúde *TestRight* v intervale 1 sec.

Join ako operátor je formálne definovaný v relačnej algebre následovne

$$R \bowtie S = \{t \cup s | t \in R \wedge s \in S \wedge Fun(t \cup s)\}$$

kde Fun je predikát, ktorý zvyčajne znamená že R a S majú aspoň jeden spoločný atribút. Tento predikát môže byť vynechaný, potom sa to bude správať rovnako ako karteziánsky súčin. [9]

Esper nepožaduje spoločný atribút, takže sa chová ako karteziánsky súčin. Vygeneruje udalosť, ktorá sa skladá so všetkých atribútov každej vstupnej udalosti. Formálne zapísané je to funkcia ktorá berie $EL_1 \dots EL_x$ a vracia výsledný EL_v, ktorý sa skladá z udalostí, ktoré obsahujú atribúty $EL_1(E(A_1)) \dots EL_1(E(A_y)) \dots EL_x(E(A_1)) \dots EL_x(E(A_z))$, kde veľkosť EL_v je $MIN(|EL_1| \dots |EL_x|)$.

Esper obsahuje aj iné druhy joinov ako napríklad (full | left | right) outer join, ktoré nebudem rozoberať pretože fungujú rovnako ako v databáze, viac informácii o týchto joinoch sa dá nájsť na stránkach Esperu [4].

Teraz sa presuniem ku praktickejšej časti a rozoberiem zopár príkladov so vstupom a ukážkami výsledkov, ktoré Esper vygeneroval. Budem používať 2 vstupné prúdy v ktorých budú testovacie udalosti, ktoré som definoval v úvode kapitoly akurát atribút *test* bude typu reťazec a nie long. Dotaz bude vždy vyzerať podobne a to: **SELECT** * **FROM** TestLeft.win:«typ ľavého okna», TestRight.win:«typ pravého okna»; Vstup budem zapisovať ako $\{a|b\}^{<<poradie>>}$ kde horný riadok tabuľky sú udalosti, ktoré sú púšťané do ľavého prúdu a spodný riadok do pravého prúdu.

Vstup pre každý príklad je rovnaký a to:

a^1 b^3 a^5 b^7
b^2 b^4 a^6 a^8

Prvý príklad je jednoduchý a typy obidvoch okien sú rovnaké a to *keepall()*. To znamená že sa udržujú všetky udalosti takže prakticky sa okno otvorí na začiatku spracúvania a zavrie na jeho konci. Vygenerovaný výsledok je:

1. $\{\}$
2. (a^1, b^2)
3. (b^3, b^2)
4. (a^1, b^4), (b^3, b^4)
5. (a^5, b^2), (a^5, b^4)
6. (a^1, a^6), (a^3, a^6), (a^5, a^6)
7. (b^7, b^2), (b^7, b^4), (b^7, a^6)
8. (a^1, a^8), (b^3, a^8), (a^5, a^8), (b^7, a^8)

Druhý príklad ukazuje funkčnosť keď je *last_event()* typom obidvoch okien . To znamená že sa udržuje len posledná udalosť. Vygenerovaný výsledok je:

1. $\{\}$
2. (a^1, b^2)
3. (b^3, b^2)
4. (b^3, b^4)
5. (a^5, b^4)
6. (a^5, a^6)
7. (b^7, a^6)
8. (b^7, a^8)

Tretí príklad ukazuje funkčnosť keď sú typy obidvoch okien *length(2)*. To znamená že sa udržujú posledné 2 udalosti. Vygenerovaný výsledok je:

1. {}
2. (a^1, b^2)
3. (b^3, b^2)
4. (a^1, b^4), (b^3, b^4)
5. (a^5, b^2), (a^5, b^4)
6. (b^3, a^6), (a^5, a^6)
7. (b^7, b^4), (b^7, a^6)
8. (a^5, a^8), (b^7, a^8)

A posledný štvrtý príklad ukazuje funkčnosť keď sú typy obidvoch okien *length_batch(2)*. To znamená že sa udržujú posledné 2 udalosti ale je to kľzajúce okno s rovnakou periódou a trvaním intervalu. Vygenerovaný výsledok je:

4. (a^1, b^2), (b^3, b^2), (a^1, b^4), (b^3, b^4)
7. (a^5, b^2), (a^5, b^4), (b^7, b^2), (b^7, b^4)
8. (a^5, a^6), (b^7, a^6), (a^5, a^8), (b^7, a^8)

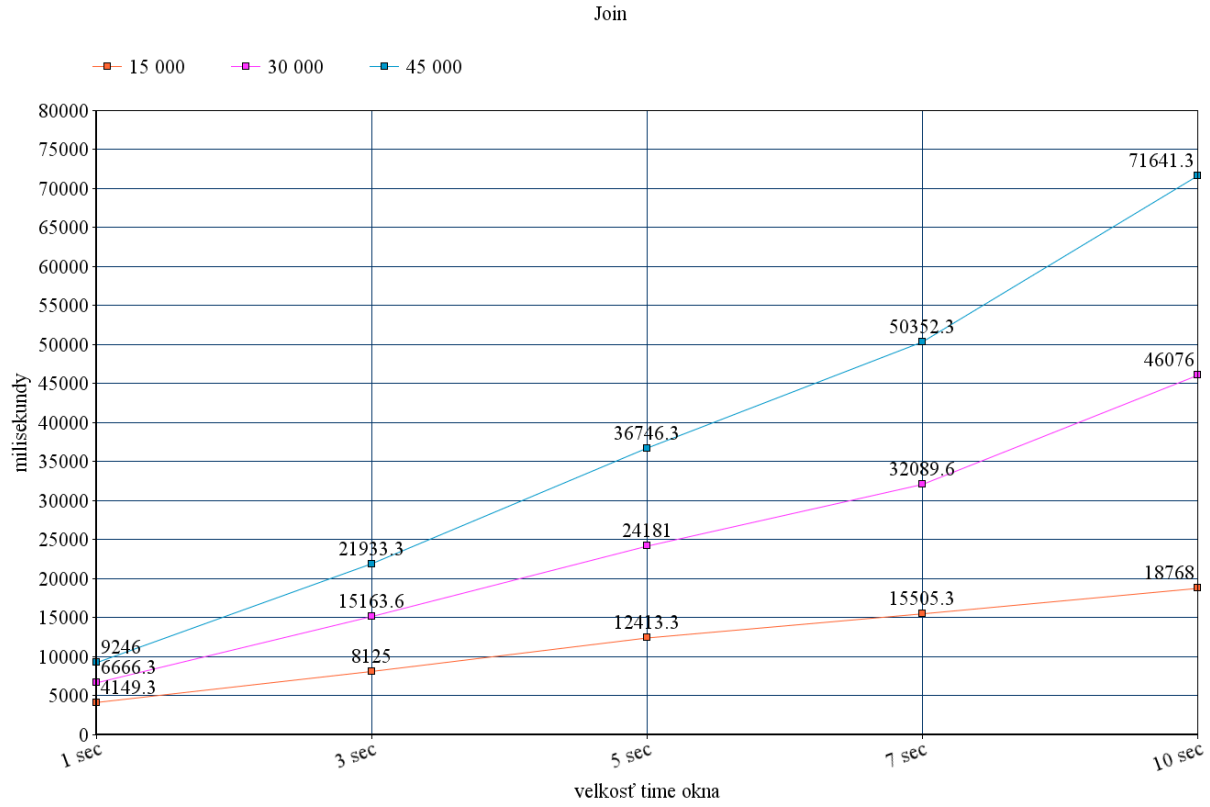
Z výsledkov vyplýva že operátor join v Esperi naozaj funguje ako karteziánsky súčin. Pri prvých troch príkladoch je vidieť že s príchodom každej novej udalosti sa najskôr pozrie či druhý prúd obsahuje nejaké udalosti. Ak nie, tak nič nespraví. Ak áno, vytvorí toľko nových udalostí, koľko je v druhom prúde udalostí. Trochu komplikovanejšie to je v štvrtom príklade, kde máme *length_batch* okno. Tam sa operátor vykoná po zaplnení okna(v našom prípade 2 udalosti) a pozrie sa na posledné plné okno v druhom prúde a vytvorí $x * y$ udalostí, kde x je počet udalostí v prvom prúde a y počet v druhom(v tomto prípade sú rovnaké, $2 * 2 = 4$). Toto správanie je najlepšie vidieť príchodom 7. udalosti. Aj keď druhý prúd už zaplnil okno udalosťami 2 a 4, prišla aj udalosť 6(čiže sa otvorilo nové okno) tak výsledné spojené udalosti sú (5,2), (7,2), ... Keď majú prúdy rôzne druhy okien, funguje každé samo za seba, rovnako ako v týchto príkladoch. Úplne rovnako to funguje aj pri časových oknách, akurát je rozdiel v *time_batch*, pretože musí čakať na udalosť, ktorá nepatrí do daného okna aby ho mohla uzavrieť.

Teraz prejdem ku výkonu tohoto operátora. Tento operátor je viac časovo náročný ako ostatné, plus je to blokujúci operátor, takže musí čakať na celý vstup aby mohol produkovať výsledok, a preto som musel zredukovať počet udalostí vytvorených za sekundu na 1000/sec, pretože potom by buď došla pamäť alebo by výpočet trval veľmi dlho. Z toho istého dôvodu mu na vstupe dávam iba 5 ciferné počty udalostí.

Ako je vidieť na grafe na Obrázku 4.8 doba výpočtu rastie exponenciálne ku veľkosti okna. Je to logické, keďže JOIN spojuje každú udalosť z jedného prúdu s každou udalosťou druhého prúdu.

4.4 SEQUENCE

Niekedy je potreba hľadať nejaké vzory v prúde dát, v tomto prípade sekvenciu. Sekvencia je postupnosť definovaných udalostí s definovaným poradím. Môžu sa líšiť typom alebo v hodnotách atribútov. Napríklad keď máme hocijaký systém s tlačítkom, ktoré vyvolá udalosť, môžeme



Obr. 4.8: Tento graf reprezentuje chovanie operátoru JOIN pri rozdielnych veľkostiach veľkostí kĺzajúceho okna.

vytvoriť dotaz, ktorý deteguje či nebolo tlačítko stlačené 2 alebo viackrát za 3 sekundy. Ak sa tak stalo, akceptuje sa iba prvá udalosť(predpoklad že ostatné sú nechcené kliknutia). V Esperi nie je sekvencia implementovaná zvlášť ako operátor, ale dá sa modelovať pomocou všeobecnejšieho modelu detekovania vzorov. Keď detegujeme sekvenciu, zvyčajne nechceme aby nám systém odpovedal že ju našiel a ďalej nehľadal, preto je v Esperi kľúčové slovo *every*, ktoré zaisťuje opakované hľadanie.

Formálne, sekvencia je operátor, ktorý zoberie na vstup EL a na výstup posiela odpovede o nájdení požadovanej sekvencie. Posiela však iba kladné odpovede.

Funkčnosť spomínaného kľúčového slova *every* spočíva v tom, že pri každom výskyte danej udalosti, vytvorí pre ňu sub-výraz, ktorý ďalej hľadá ďalší prvok sekvencie. Napríklad pri dotaze *every(A -> B)* pri výskyte druhej udalosti *A* sa vytvorí spomínaný sub-výraz a pri výskyte udalosti B sa zoberie najdlhšie aktívny sub-výraz a ukončí sa odpoveďou *sekvencia sa našla* [5]. Lepšie to je vidieť na príkladoch.

V Esperi som sa zameral hlavne na funkčnosť kľúčového slova *every*, takže nasledujúce príklady sú vstup+dotaz/výsledok. Výsledná tabuľka bude vyzerať následovne, prvý stĺpec obsahuje dotaz na sekvenciu, ktorá sa hľadá a ostatné obsahujú číslo reprezentujúce počet sekvencií nájdených v danom vstupe. Vstup je postupnosť písmen(pre zjednodušenie budem vypisovať iba atribút *test*, ktorý je dátového typu reťazec).

Prvý príklad je najjednoduchší, testuje či môže byť sekvencia prerušená inou udalosťou bez ovplyvnenia. Výsledok je jednoznačný a kladný. Vstup je $[a, f, b, f, c]$. a pre každý dotaz bol výsledok práve jedna sekvencia. Druhý príklad je vstup $[a, b, c, a, b, c, a, b, c, a, b, c]$. Tretí príklad má vstup $[a, a, a, a, b, b, b, b, c, c, c, c]$.

Dotaz	1. príklad	2. príklad	3. príklad
a -> b	1	1	1
every(a -> b)	1	4	1
every a -> b	1	4	4
a -> every b	1	4	4
every a -> every b	1	10	16
a -> b -> c	1	1	1
every(a -> b -> c)	1	4	1
every a -> b -> c	1	4	4
a -> every b -> c	1	4	4
a -> b -> every c	1	4	4
every a -> every b -> c	1	10	16
a -> every b -> every c	1	10	16
every a -> b -> every c	1	10	16
every a -> every b -> every c	1	20	64

Z týchto výsledkov sa dá pekne vidieť ako to funguje na zadaných vstupoch. No v realite nie sú vstupy vždy takéto „pekné“. Ďalší príklad, ktorý spomeniem berie na vstup mnou náhodne vymyslenú postupnosť a ukážem, ktoré udalosti sú zahrnuté v každej sekvencii.

Vstup je následovný: [A^1, B^2, C^3, B^4, A^5, D^6, C^7, A^8, F^9, A^{10}, A^{11}, B^{12}]. Číslo v exponente je len informačné, podľa neho budem rozlišovať udalosti. Formát následovného listu je «dotaz» nový riadok a očíslovaný list vo formáte «udalosť, pri ktorej sa našla sekvencia» | [«list sekvencií»].

- $A \rightarrow B$
  1. $B^2 | \left[(A^1, B^2)\right]$
- $every\,(A \rightarrow B)$
  1. $B^2 | \left[(A^1, B^2)\right]$
  2. $B^{12} | \left[(A^5, B^{12})\right]$
- $everyA \rightarrow B$
  1. $B^2 | \left[(A^1, B^2)\right]$
  2. $B^{12} | \left[(A^5, B^{12}), (A^8, B^{12}), (A^{10}, B^{12}), (A^{11}, B^{12})\right]$

- $A \rightarrow everyB$
  1. $B^2|\,[(A^1, B^2)]$
  2. $B^4|\,[(A^1, B^4)]$
  3. $B^{12}|\,[(A^1, B^{12})]$
- $everyA \rightarrow everyB$
  1. $B^2|\,[(A^1, B^2)]$
  2. $B^4|\,[(A^1, B^4)]$
  3. $B^{12}|\,[(A^1, B^{12}), (A^5, B^{12}), (A^8, B^{12}), (A^{10}, B^{12}), (A^{11}, B^{12})]$

Viac príkladov je na [5]. Teraz prejdem ku výkonu tohoto operátora. Premeral som hľadanie vzoru *every(1->2->3)* a *every 1 -> every 2 -> every 3*. Keďže sekvencia vytvára pre každú novú sekvenciu podvýraz, musel som taktiež zredukovať vstup pretože dochádzala pri testovaní pamäť.

Na grafe na Obrázku 4.9 sú vidieť časové hodnoty pre vzor *every(1->2->3)*. Tieto hodnoty výrazne klesajú, čo je logické keďže zväčšujem rôznosť atribútu test, čo znamená že je menšia pravdepodobnosť že sa vyskytne vo vstupe sekvencia(udalosti, ktoré neobsahujú v atribúte test čísla 1,2 alebo 3 sú ignorované). V grafe je aj konštantná hodnota pre nenáhodný vstup a to postupne 1,2,3,1,2,3,... Samozrejme že pre tento vstup je výpočet pomalší, pretože je tam tých sekvencií najviac.

To isté platí aj pre vzor *every 1->every 2->every 3* zobrazený na grafe na Obrázku 4.10. Pri tomto meraní som musel znížiť počet vstupných udalostí až na 4 ciferné čísla, pretože tento vzor vytvára veľké množstvo podvýrazov, ktoré rýchlo zaplnili pamäť.

4.5 ORDER BY

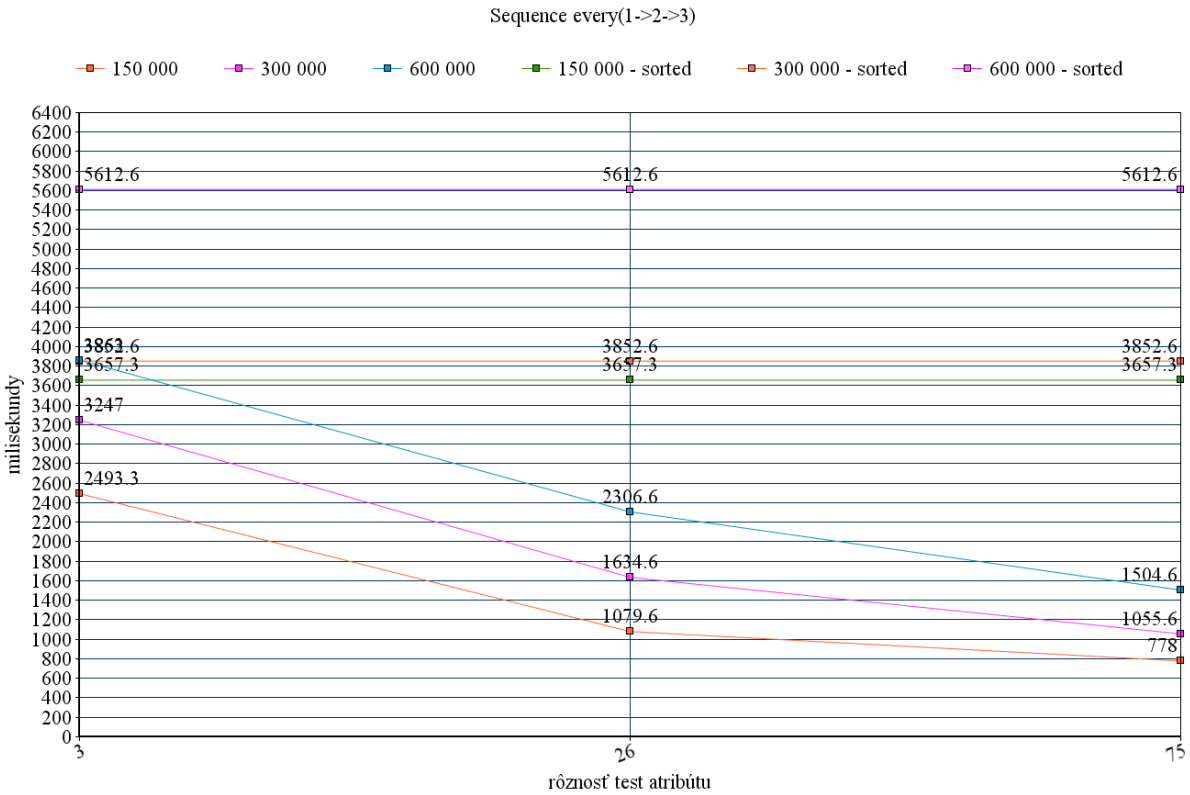
Udalosti máme zoradené podľa času príchodu, časového razítka. Lenže môže nastať situácia, kedy je potreba ich zoradiť pomocou nejakého iného atribútu, napríklad nejakej ceny a podobne. Aby to nebolo tak jednoduché, musíme ešte definovať počet udalostí, po ktorých to chceme zoraďovať. Je to celkom logické, keďže bez tejto informácie by sme sa snažili zoradiť celý potenciónálne nekonečný prúd a to by nemuselo dopadnúť dobre.

Syntax tohoto operátora je následovná:

```
order by expression [asc | desc] [, expression [asc | desc]] [, ...]
```

Ako je vidieť zo syntaxe, môžeme zoraďovať pomocou viacerých atribútov naraz. Ďalej si taktiež môžeme zadefinovať či chceme zoraďovať vzostupne(asc) alebo zostupne(desc), kde vzostupne je preddefinovaná hodnota.

Formálne je to funkcia, ktorá zoberie EL a číslo n a vráti n-tice zoradených udalostí podľa definovaných atribútov. Môžeme zoraďovať aj pomocou viacerých atribútov. Potom to funguje ako pri alfabetickom zoraďovaní. Najskôr sa vstup zoradí pomocou prvého atribútu a kde sú rovnaké, rozhodne sa o ich poradí pomocou druhého atribútu, atď.

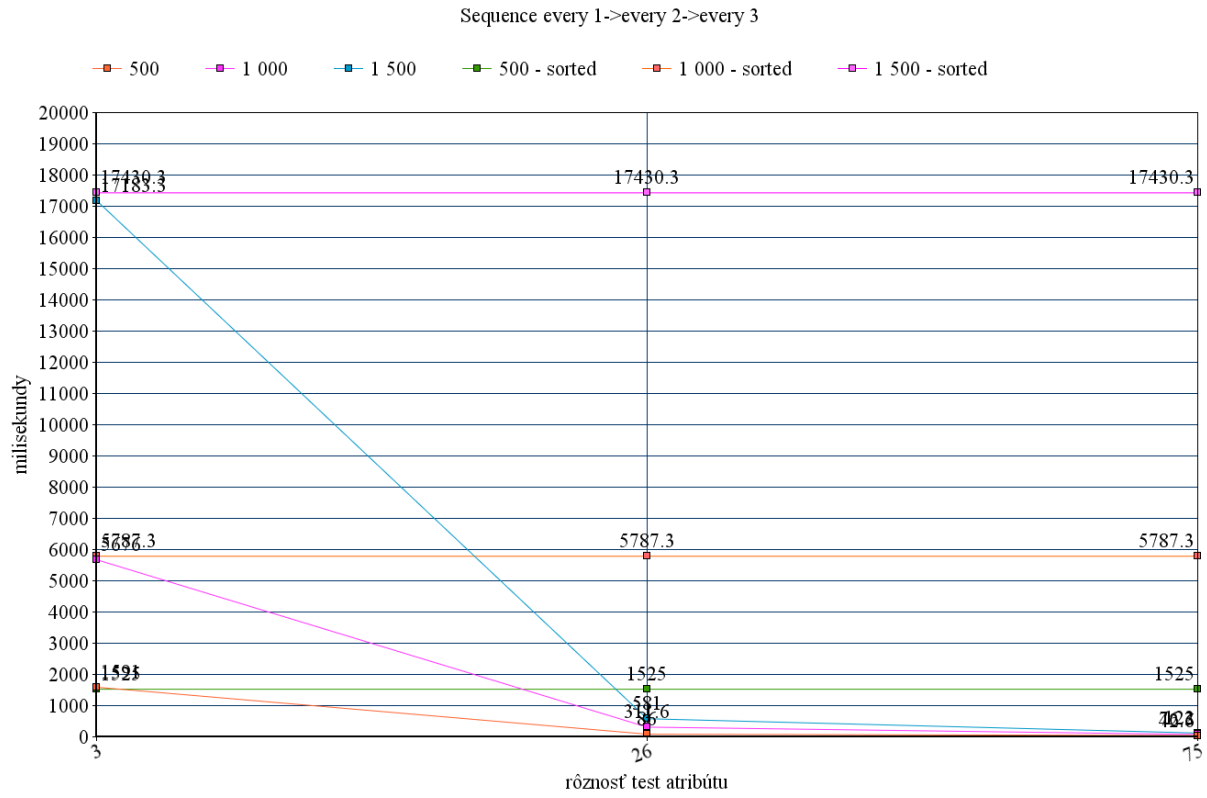


Obr. 4.9: Tento graf reprezentuje chovanie operátora SEQUENCE pri rozdielnych rôznych hodnotách atribútu test. Sorted(zoradené) znamená že atribút udalostí test má hodnoty postupne 1,2,3,1,2,3,....

Esper na operátor *ORDER BY* kladie nasledujúce obmedzenie [4]:

- Všetky agregačné funkcie, ktoré sa objavia v ORDER BY klauzule sa musia objaviť aj v SELECT výraze.

Poďme sa teraz pozrieť na funkčnosť tohoto operátora v Esperi. Formát vstupu sa nezmení oproti minulým príkladom, to znamená že budem písať iba atribút test(ktorý je typu reťazec) a ako jeho horný index budem písať poradie(na rozlíšenie udalostí). Vstup pre prvý príklad je [a^1, b^2, c^3, d^4, e^5, f^6, g^7, h^8, i^9, j^10] a vstup pre druhý príklad je presne opačný, t.j. [j^1, i^2, h^3, g^4, f^5, e^6, d^7, c^8, b^9, a^10]. Pre obidva príklady je veľkosť buffru 3.



Obr. 4.10: Tento graf reprezentuje chovanie operátora SEQUENCE pri rozdielnych rôznych hodnotách atribútu test. Sorted(zoradené) znamená že atribút udalostí test má hodnoty postupne 1,2,3,1,2,3,....

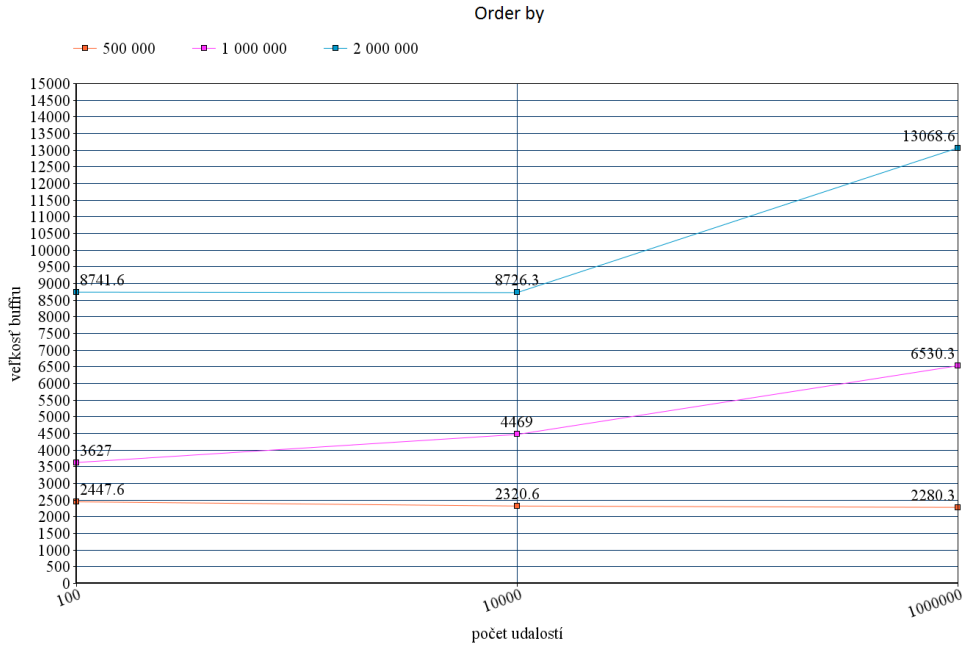
Výsledok prvého príkladu je následovný:

prišla udalosť	časové razítko	test
	1	a
E^3	2	b
	3	c
	4	d
E^6	5	e
	6	f
	7	g
E^9	8	h
	9	i

Výsledok druhého príkladu je veľmi podobný:

prišla udalosť	časové razítko	test
	3	h
E^3	2	i
	1	j
	6	e
E^6	5	f
	4	g
	9	b
E^9	8	c
	7	d

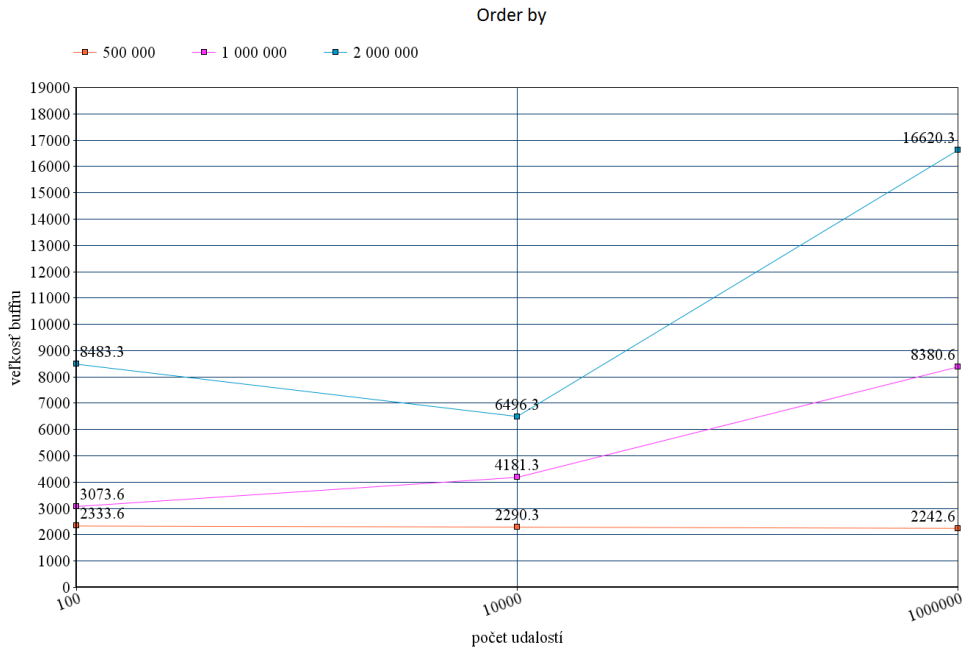
Z výsledkov je pekne vidieť že funkčnosť operátora odpovedá jeho definícii. Teraz sa presuniem na porovnanie výkonu operátora. Zmeral som ho na základe zmeny veľkosť buffru, ktorý ostáva zoradený. Dotaz vyzerá následovne: **SELECT** test, timestamp **FROM** Test.win:keepall() output every X events **ORDER BY** test;



Obr. 4.11: Tento graf reprezentuje chovanie operátora ORDER BY pri rôznych hodnotách veľkosti buffru.

Na grafe na Obrázku 4.11 je vidieť že pri väčšej veľkosti buffru je výkon operátora pomalší. To je jasné, keďže mu na vstupe príde viac udalostí, ktoré musí zoradiť.

Z grafu na Obrázku 4.12 môžeme vydedukovať že Esper používa interne stromové uloženie, keďže až pri veľkosti okna 1 000 000 je výpočet náročnejší ako pri náhodnom vstupe. Očividne je vkladanie nových listov do stromu rýchlejšie ako vytvorenie nového stromu.



Obr. 4.12: Tento graf reprezentuje chovanie operátora ORDER BY pri rôznych hodnotách veľkosti buffru a vstupe zoradenom zostupne.

4.6 First N

Tento operátor je v Esperi implementovaný ako okno. Správa sa to ako kontext, do ktorého patrí prvých n udalostí a ďalšie do neho nepatria. Tento operátor som už spomenul v 1.2.3 ako vzor pre detekčného agenta.

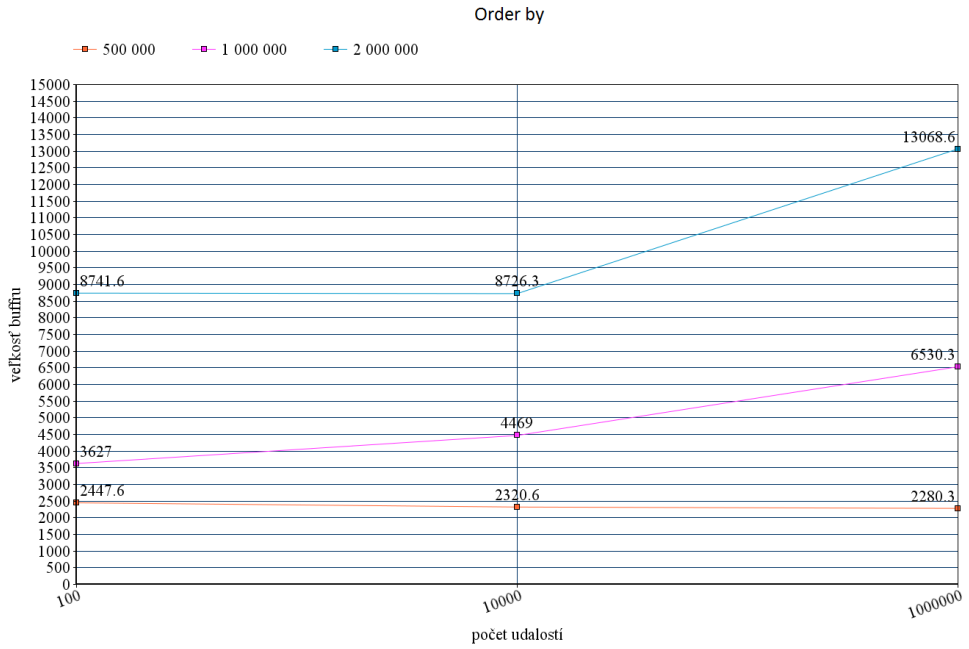
Formálne je to funkcia, ktorá berie na vstup EL a číslo n a vracia EL_v, ktorého dĺžka je n a obsah tvorí prvých n udalostí z EL. Ak veľkosť EL je menšia ako n, potom $EL = EL_v$.

Keďže funkčnosť tohoto operátora je úplne zrejmá. Teraz prejdem na funkčnosť implementácie pomocou Esperu. Uvediem 3 príklady, ktoré budú mať rovnaký vstup ale rôznu veľkosť bufferu. Vstup je [a^1, b^2, c^3, d^4, e^5, f^6, g^7, h^8, i^9, j^10]. Prvý riadok tabuľky budú veľkosti bufferu pre každý stĺpec. Udalosť je ďalej poslaná hneď pri jej príchode, takže budem písať dvojice (test, časové razítko).

Výsledok ku danému vstupu je nasledovný:

5	10	20
(a,1)	(a,1)	(a,1)
(b,2)	(b,2)	(b,2)
(c,3)	(c,3)	(c,3)
(d,4)	(d,4)	(d,4)
(e,5)	(e,5)	(e,5)
	(f,6)	(f,6)
	(g,7)	(g,7)
	(h,8)	(h,8)
	(i,9)	(i,9)
	(j,10)	(j,10)

Teraz prejdem ku rýchlosti tohoto operátora. Premeral som rýchlosť na základe zmeny veľkosť bufferu rovnako ako pri minulom operátore.



Obr. 4.13: Tento graf reprezentuje chovanie operátora FIRST N pri rozdielnych rôznych hodnotách veľkosti buffru.

Na grafe na Obrázku 4.13 je pekne vidieť aj funkčnosť tohoto operátora. Pre 1 000 000 a 2 000 000 udalostí, keď je veľkosť bufferu 1 000 000, tak ich výkon je rovnaký(pre 2 000 000 by mal byť väčší, lebo ešte musí ignorovať ďalších milión udalostí ale to je chvíľka).

Kapitola 5

Záver

Bakalárska práca zahrňuje základné pojmy z teórie spracúvania dát, snaží sa o zovšeobecnenie architektúry, ktorá má veľa rôznych implementácií. Porovnáva tento prístup so systémom pre správu databáz, kde ukazuje na rozdiely a podobnosti.

Ďalej rozoberá dôležité, často používané operátory, vysvetľuje ako pracujú. Funkčnosť je prezentovaná na rôznych príkladoch kde je uvedený vstup a výstup. Je premeraná časová náročnosť každého operátora na rôznych vstupoch(rozdielna veľkosť okna, rôznorodosť testovaného atribútu, počet udalostí).

Možným rozšírením tejto práce by mohlo byť rozpísanie viacerých operátorov alebo porovnanie správania a funkčnosti operátorov s iným programovacím jazykom pre spracúvanie dát ako napríklad Microsoft StreamInsight[1] alebo iným.

1. http://technet.microsoft.com/en-us/library/ee362541.aspx

Literatúra

[1] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *???*, 2003.

[2] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. *???*, 2002.

[3] Opher Etzion and Peter Niblett. *Event processing in action*. Manning, c2011.

[4] EsperTech Inc. Chapter 4. epl reference: Clauses. `http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/epl_clauses.html`. [Online; navštívené 5.5.2014].

[5] EsperTech Inc. Chapter 6. epl reference: Patterns. `http://esper.codehaus.org/esper-4.6.0/doc/reference/en-US/html/event_patterns.html`. [Online; navštívené 8.5.2014].

[6] EsperTech Inc. Esper - license. `http://esper.codehaus.org/about/license/license.html`. [Online; navštívené 26.4.2014].

[7] Wikipedia. Spracovanie dát — Wikipedia, the free encyclopedia. `http://sk.wikipedia.org/wiki/Spracovanie_d%C3%A1t`, 2009. [Online; navštívené 04-November-2009].

[8] Wikipedia. Data stream management system — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Data_stream_management_system`, 2014. [Online; navštívené 11-May-2014].

[9] Wikipedia. Relational algebra — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Relational_algebra#Natural_join`, 2014. [Online; navštívené 07-May-2014].

[10] Zigomatic. Online charts | create and design your own charts and diagrams online. `http://www.onlinecharttool.com/`. [Online; navštívené 29.4.2014].