Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky
Katedra informačních technologií

Studijní program: Aplikovaná informatika
Obor: Informatika

Knihovna umožňující práci s libovolnými zdroji dat prostřednictvím SQL dotazů

BAKALÁŘSKÁ PRÁCE

Student : Jan Sýkora
Vedoucí : Ing. Rudolf Pecinovský, CSc.
Oponent : Ing. Vladimír Oraný

2015

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze které jsem čerpal.

V Praze dne 6. května 2015 . . . . . .

Jan Sýkora

Poděkování

Rád bych na tomto místě poděkoval panu Ing. Rudolfu Pecinovskému, CSc., za odborné vedení mé bakalářské práce, cenné rady a připomínky, za odborné konzultace v dané oblasti a čas, který mi při psaní této práce věnoval.

Abstrakt

Osmá verze programovacího jazyka Java přinesla řadu novinek, které silně inklinují k deklarativnímu programování. Ve své práci jsem zaměřil na možnosti využití těchto novinek v oblasti deklarativního zpracovávání dat na platformě Java. Pro co nejvyšší míru deklarativnosti jsem zvolil Structured Query Language jazyk.

Cílem mé práce bylo vytvořit knihovnu, jež by interpretovala SQL dotazy. Knihovna k interpretaci využívá datovody a celou řadu prvků z Javy, jež umožňují větší míru funkcionálního programování v Javě.

Hlavním přínosem této práce bylo dát vývojářům alternativu v oblasti zpracování a filtrování dat, jež je z důvodu optimalizace většinou prováděno v perzistentní vrstvě aplikací.

Klíčová slova

datovody, SQL, Java 8, deklarativní programování, interpret

Abstract

Eighth version of programming language Java brought broad variety of changes, which strongly incline to declarative style of programming. In my thesis I have focused on using these changes in declarative ways of data processing on Java platform. I have decided on using Structured Query Language due to its declarativity

Topic of my thesis was to create a library, which could interpret SQL queries. For interpretation, my library uses Streams and many other concepts which allow functional style of programming in Java.

The most significant benefit of this library was giving alternative to programmers, who are looking for ways, how process data in application layer rather than on persistent layer.

Keywords

Streams, SQL, Java 8, declarative programming, interpret

Obsah

1 Úvod

Předmětem mé bakalářské práce je zastupitelnost a převoditelnost SQL dotazů za datovody v jazyku Java 8. Osmá řada Javy přišla s řadou vylepšení a nabízí širší využití funkcionálního programování v Javě. Tím se otvírají dveře řadě deklarativních implementací, vývojář se tedy vyhne imperativní implementaci. Obecně téma mé práce se zabývá deklarativním zpracováním dat, což v prostředí Javy není jednoduchý úkol. Proto jsem zvolil jazyk SQL, jež je velmi deklarativní a rozšířený jak mezi vývojáři, tak i mezi ostatními profesemi. Touto knihovnou chci nabídnout uživatelům využít nových funkcionalit Javy 8 bez potřeby její hluboké znalosti skrze interpretaci SQL dotazů.

Jelikož v současné době je většina dat uložena v prostředí relačních databází, tak se vývojáři nevyhnou jazyku SQL. Díky tomu jsou vývojáři skoro v denním kontaktu s SQL jazykem nebo s jeho možnou derivací. Při řešení zadání nebo úkolu vývojáři často stojí před rozhodnutím, zda mají řešit problém na úrovni aplikace nebo v databázi. Touto knihovnou chci dát vývojářům možnost tuto situaci řešit na úrovni aplikace a i přesto zůstat v deklarativní rovině. Nepřímým záměrem bylo vytvořit knihovnu, jež by se stala možným konkurentem knihovny LINQ v .NETu.

Knihovna je určena pro veřejnost a jakéhokoliv uživatele. Z pohledu businessu jsou uživatelé knihovny mými klienty a mou úlohou je uspokojit jejich potřeby a požadavky. Což vede k dalšímu vývoji knihovny a její evoluci, a i přesto by knihovna měla být zpětně kompatibilní s kódem klienta [1]. Návrh a implementace knihovny si sebou nese řadu specifik a problémů. V řešení těchto věcí vidím největší přínos pro mne, kdy dosud jsem měl zkušenosti jen s psaním menších modulů a komponent, které jsou využívané menším počtem uživatelů v uzavřeném prostředí, kde znám své „klienty“ a požadavky, pokud se mění, tak jenom částečně.

Druhým přínosem je prohloubení znalostí a zkušeností s funkcionálním programováním, jelikož dosud jsem se zabýval imperativní programováním a s deklarativním programováním jsem se pouze setkával v SQL a v některých webových technologií.

Knihovna očekává od svého uživatele řadu znalostí, prvním z nich je znalost samotného SQL jazyka. V případě SQL knihovna vychází ze standardu SQL-93 [2]. Nicméně pro svoje interní potřeby je syntax a gramatika zjednodušena. Nedá se říci, že by ji zásadně měnila, ale spíše ji ořezává do stavu, který je schopna interpretovat. Dalším obecným předpokladem je, aby uživatel byl schopen rozeznat rozdíly mezi sémantickými modely využívanými v relačních databázích a modelem reprezentovaný třídami v Javě a následně podle toho upravit zadávaný SQL dotaz.

2 Rešerše

Před napsáním práce jsem provedl rešerši literatury, jež se týká cíle mé práce. Do rešerše jsem zařadil jak knihy týkající se Javy 8, funkcionálního programování a SQL, ale také zároveň zdroje, které mi pomohli navrhnout a implementovat výslednou knihovnu.

V oblasti Javy 8 a funkcionálního programování jsem se využil dvě knihy, první z nich je Java 8 in Action [3], tato kniha plně pokrývá novinky osmé řady Javy a na příkladech je ukazuje. Druhou knihou je Java 8 Lambdas od Richarda Warburtona [4]. Pan Warburton se blíže věnuje implementacím novinek a jejich konkrétním aplikacím.

Při využívání zdrojů o jazyku SQL jsem opíral publikaci SQL: kompletní kapesní průvodce od Milana Šimůnka [5], jež se zabývá jazykem SQL a relačními databázemi. Z pohledu vývojáře tato publikace vystačí pro běžné užívání jazyka SQL,

Při návrhu knihovny jsem využil knihu Jaroslava Tulacha, Practical API Design [6]. I přes řadu změn, které nastaly v Javě, je tato kniha cenným zdrojem. Jelikož řada těchto postřehu a doporučení vychází ze zkušeností autora.

Výše zmíněné publikace jsem doplnil knihou JUnit in Action od Petara Tahchieva a Massola Vincenta [7], jež mi umožnila napsat JUnit testy určené pro kontinuální testování a vývoj knihovny.

3 Analýza

V rámci analýzy chci nastínit současnou situaci a možnosti deklarativního programování v Javě, upozornit na řadu problémů a ukázat, jak je některé knihovny a frameworky řeší, případně jakým způsobem rozšiřují možnosti funkcionálního programování v Javě.

3.1 Deklarativní programování v Javě 8

V případě, že by mé téma bylo jen čistě o interpretaci SQL dotazů, tak bych asi retrospektivně zvolil jiný jazyk. Mojí volbou by byla Scala nebo Groovy a to z důvodu, že jejich syntax je postavena na Javě a jejich deklarativnost daleko přesahuje Javu. Oba tyto jazyky běží na platformě Java. Mnoho knihoven a technologií v současné době jsou již implementované ve Scale.

Při prvním styku s Javou 8 mě okamžitě zaujaly datovody, jako alternativa v oblasti zpracování dat, ale při bližší zkoumání a pokusech jsem zjistil, že to jsou ostatní novinky, které Javě dávají funkcionální punc a otvírají ji nové možnosti. Velkým překvapením pro mne bylo funkční rozhraní, které vývojáři umožňuje s funkcemi nativně pracovat jako s objekty a předávat si je. Příkladem je interfejs `Predicate`, který je také vstupním parametrem metody `filter(Predicate<? super T> predicate)` [8]. Již zde se nabízejí možnosti tyto predikáty vytvářet dynamicky až za běhu programu.

Samozřejmě pro spoustu vývojářů jsou Lambdy větším přínosem a to z důvodu, že odpadl nepotřebný syntaktický cukr a lze se vyjádřit nyní kratším způsobem, namísto anonymní třídy.

3.2 Uspořádaná n-tice

V mé práci používám termín uspořádaná n-tice prvků, proto ji chci v této části pevně definovat, aby nemohlo dojít k jiným výkladům. V anglickém jazyce se tento pojem nazývá *tuple* a může se na něj narazit v oblasti relačních databází nebo v algebře. Z hlediska definice se jedná o množinu prvků, která má pevně danou velikost (velikost je větší nebo rovna 0) a tyto prvky jsou uchovávány ve stanoveném pořadí. Na rozdíl od setu uchovává pořadí a může obsahovat duplicity

U relačních databází jsou termíny řádek, záznam a *tuple* za sebe zaměňovány a jejich význam a definice se považují za stejné. Všechny technologie v Javě, které zpracovávají data, dřív nebo později narazí na problém, že jejich výstupem je uspořádaná n-tice. Tento

fakt je většinou řešen výběrem jedné z implementací n-tice, často je volen seznam nebo pole.

Toto řešení má problém v typové kontrole, protože typ toho seznamu nebo pole je třída `Object` a to z důvodu, že autoři knihoven dopředu neví, jakého typu budou výsledná data. Jediné, co s jistotou ví, je že to budou potomci třídy `Object`. Druhým možným přístup je vytvořit vlastní implementaci uspořádané n-tice. Tato možnost otvírá dveře i typové kontrole, příkladem může být framework jOOQ a jeho interfejs `Record` [9].

3.3 Přehled současného stavu

V současné době má vývojář k dispozici řadu frameworků a knihoven jež jsou specializované na jednotlivé domény. Záleží tedy pouze na vývojáři, po kterých nástrojích sáhne a jak šikovně je dokáže integrovat do své aplikace. Proto v této části jsem uvedl řadu knihoven, které se zabývají oblastmi, jež jsem také řešil při návrhu a implementaci mé knihovny.

3.3.1 Knihovny určené pro ORM

V rámci knihoven pro ORM jsem hledal možná řešení, jak spojit dva rozličné sémantické model a zkoumal jsem jejich přístup k SQL jazyku.

3.3.1.1 Hibernate/JPA

Tyto dvě technologie nadále zůstávají průmyslovými standardy a oblíbenými nástroji mezi vývojáři. I přestože jejich hlavním cílem je ORM a dotazování do databáze, tak ve velké míře je možné je využít pro zpracování dat a jejich filtraci skrze SQL dotazy. Hibernate/JPA se opírají o výhody i omezení relačních databází jako takových. Tyto technologie kromě jazyka SQL také podporují jazyk HQL a JPQL, které pracují s objektovým kontextem. I přesto jsou založeny na SQL a jsou deklarativní.

Tyto dvě technologie se také opírají o další standardy JavaEE, jako příklad lze uvést přístup k atributům serverový komponentám (Enterprise JavaBean) skrze přístupové metody (obě knihovny neberou ohled, zda jsou privátní nebo ne). Samotné EJB jsou mapované třídy skrze anotace nebo konfigurační XML soubory. Jelikož Hibernate podporuje kompletní SQL jazyk, včetně velké řady dialektů, tak musí řešit problém, jak přistupovat ke konstrukci JOIN a výběru pouze sloupců v SELECT dotazu (Výpis 1).

Výpis 1: Ukázka HQL dotazu

```
String hql = "select name
    from list1 as t
    inner join list2.name as name
    where t.number = 3";
Query query = session.createQuery(hql);
List results = query.list();
```

Jedním z možných řešení je mít připravenou třídu pro konkrétní případ, ale to vyžaduje implementaci navíc. Přistup, který byl zvolen, jsou uspořádané n-tice objektů jakožto pole objektů. Toto řešení separuje práci s databází od mapování.

Tyto technologie mají řadu chyb a nedostatků, jedním z nich, že plně nepodporuje typovou kontrolu prováděnou kompilátorem i přestože SQL jazyk jako takový je velmi silně typový jazyk. Tento fakt je zapříčiněn tím, že SQL/HQL/JPQL dotazy jsou psány v textové podobě (`String`) a tím, že při výběru sloupců vrací n-tici objektů jako pole. Druhým problémem jsou poměrně velké náklady na udržitelnost a správu při použití na větším projektu. Hibernate/JPA vyžaduje, aby každá entita měla 3 modely. Prvním z nich je databázový, který vyjadřuje uložení entity v relační databázi. Druhým je třída v Javě a třetím je mapovací model (anotace nebo XML soubor). Takže při větším počtu entity počet modelů velmi rychle narůstá.

3.3.1.2 jOOQ

Z mého pohledu framework jOOQ je ORM framework na steroidech. Krátce pro příchodu Javy 5 vznikl tento framework, který se snaží řešit nedostatky JDBC a ostatních ORM knihoven. Knihovna jOOQ preferuje *database-first* postup, kdy se zaměřuje na aplikace, jejichž databáze je obtížné měnit a upravovat dle potřeby. Takže se třídy v Javě vytváří na základě již hotové a připravené databáze. V knihovně jOOQ je připravený generátor Java tříd [10] z databáze pomocí zpětného inženýrství, který je spouštěn skrze nástroje pro sestavování aplikací jako je Maven.

JOOQ je velmi vhodný pro aplikace, které využívají komplexní dotazy do databáze spíše než CRUD operace [11]. Cílem tvůrců bylo, aby vývojář mohl psát Java kód, který při kompilaci vytvoří SQL dotaz (Výpis 2). Toho dosáhli využitím návrhového vzoru stavitel, kdy vytvářejí řetězce volání metod z knihovny. Parametry těchto metod jsou třídy a atributy vygenerované třídy skrze zpětné inženýrství. Tyto dvě věci umožňují automatické doplňování, kontrolu syntaxe a typovou kontrolu již v době kompilace. Takže jOOQ lze považovat za typově bezpečný. To neplatí pro některé jeho implementace, například v JavaScriptu.

Výpis 2: Ukázka SQL dotazu vyjádřeného Javou pomocí knihovny jOOQ

```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, count())
    .from(AUTHOR)
    .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTHOR_ID))
    .where(BOOK.LANGUAGE.eq("DE"))
    .and(BOOK.PUBLISHED.gt(date("2008-01-01")))
    .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
    .having(count().gt(5))
    .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
    .limit(2)
    .offset(1)
```

Typová kontrola je i zde ale omezená a to na 22 typů (interfejs `Record`) [9] a to z důvodu omezení v jazyce Scala na počet typových parametrů. Obecně toto řešení není elegantní, ale v současné verzi Javy není lepší řešení. Projekt Valhalla [12], který je součástí plánované Javy 10, by toto mohl změnit.

JOOQ i přesto lze považovat, za nejkomplexnější ORM Framework, jelikož podporuje SQL jako celek, včetně novějších standardů. Mezi běžnými ORM frameworky nenajdeme jiný, který by také podporoval Window Functions [13]. Tím se jOOQ stává velmi unikátním, další výhodou je podpora specifických funkcí jednotlivých dialektů jazyka SQL. Tímto se stal velmi vhodným nástroje například v bankovnictví, kdy aplikace operují na historických a robustních databázích.

Výpis 3: Ukázka Window Function vyjádřené pomocí knihovny jOOQ

```
create.select(t.BOOKED_AT, t.AMOUNT,
        sum(t.AMOUNT).over().partitionByOne().orderBy(t.BOOKED_AT)
    .rowsBetweenUnboundedPreceding()
    .andCurrentRow().as("total")
    .from(TRANSACTIONS.as("t"));
```

3.3.2 LINQ

V roce 2007 Microsoft spolu s C# 3.0 přišel s integrovaným jazykem pro dotazování. Tento jazyk přinesl větší abstrakci. Kromě dotazování nad databázemi, bylo možné se dotazovat nad objekty, XML a konstrukcí DataSet [14]. LINQ také umožňoval kromě dotazovací syntaxe využít lambda syntax

Lambda syntax

Výpis 4: Ukázka knihovny LINQ využívající lambda syntax

```
string[] people = new [] { "Tom", "Dick", "Harry" };
var filteredPeople = people.Where (p => p.Length > 3);
```

Query syntax

Výpis 5:Ukázka knihovny LINQ využívající query syntax

```
string[] people = new [] { "Tom", "Dick", "Harry" };
var filteredPeople = from p in people where p.Length > 3 select p;
```

LINQ tudíž přinesl řadu prvků z funkcionálního programování do .NETu. LINQ jako inspirace dal vzniknout řadě jiných knihoven v ostatních jazycích, které se snažily přinést podobnou funkcionalitu. Mezi tyto jazyky patří JAVA, JavaScript, PHP a Python.

3.3.1.1 Linq4j

Linq4j je pravděpodobně nejznámější implementací knihovny LINQ v jazyku Java. Jeho základy jsou postaveny na projektu Saffron [15] (od stejného autora). Autoři knihovny umožňují manipulaci a práci s daty nad třemi interfejsy (`Iterable`, `Enumerable` a `Queryable`). Druhé a třetí z těchto rozhraní jsou dodány knihovnou. Knihovna si vytváří pro svoje interní účely vlastní implementaci iterátoru v podobě `Enumerator` [16], který lze získat jako návratovou hodnotu implementované metody z interfejsu `Enumerable`. Tento `Enumerator` se dá analogicky srovnat s `System.Collections.Enumerator` z knihovvny LINQ, s tím rozdílem, že zde je pouze dobrovolné vyhazovat výjimku, která nastává v případě konkurenčního přístupu ke zdroji.

3.3.3 Knihovny podporující funkcionální paradigma

3.3.1.1 Functional Java

Functional Java je open-source knihovna zaměřující se na podporu funkcionálního programování. Vydané verze této knihovny cílí i na plně produkční aplikace. Oproti ostatním knihovnám necílí pouze na Javu 8, ale i na Javu 7. Při využití knihovny Retro Lambda je možné i částečného využití až do Javy 5 [17]. Knihovna je rozdělena na dva balíčky, kdy hlavním balíčkem je `functionaljava.core`. Tento balíček poskytuje hlavní funkcionalitu knihovny. Druhým balíčkem je `functionaljava.java8`, který dodává specifickou podporu pro Javu 8 a má závislosti do hlavního balíčku.

Jedná se o velmi povedenou knihovnu, která rozšiřuje možnosti manipulace s kolekcemi.

Syntax pro Javu 7:

Výpis 6: Ukázka knihovny Functional Java využívající syntax Javy 7

```
final Array<String> a = array("hello", "There", "what", "day", "is", "it");
    final boolean b = a.forall(new F<String, Boolean>()0
     {
    public Boolean f(final String s) {
        return fromString(s).forall(isLowerCase);
    }
});
```

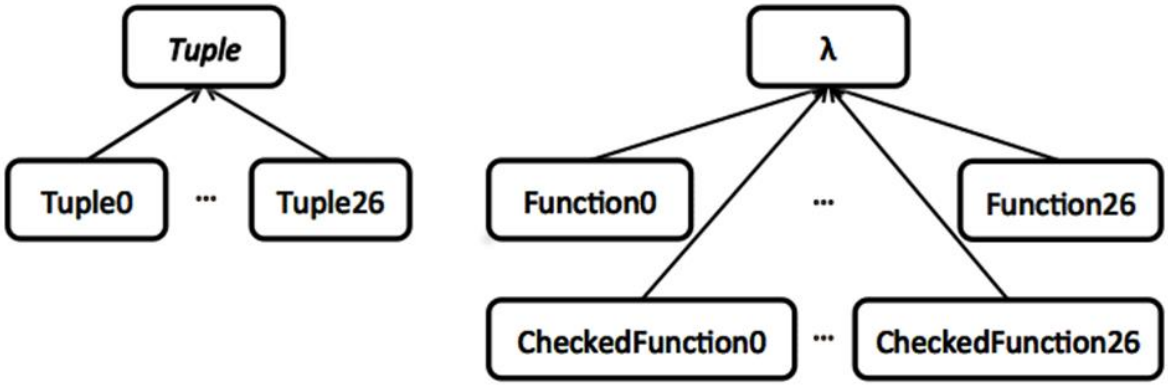
Syntax pro Javu 8

Výpis 7: Ukázka knihovny Functional Java využívající syntax Javy 8

```
final Array<String> a = array("hello", "There", "what", "day", "is", "it");
final boolean b = a.forall(s -> fromString(s).forall(isLowerCase))
```

3.3.1.2 Javaslang

Javaslang je komponenta pro funkcionální programování, která velmi dobře doplňuje základní knihovny Javy 8. Knihovna představuje implementaci uspořádané n-tice (n až 26) a poskytuje celou řadu implementací `Iterable`, které jsou neměnitelné. Dále tato knihovna poskytuje vlastní implementaci datovodů. Implementace je založena na spojových seznamech jež poskytují další prvek na požádání. Velmi zajímavá je taky implementace funkcí, které dovolují zachytávat chyby a výjimky za běhu.



Obrázek 1: Schéma Javaslangu Zdroj: [18] (upraveno inverzí barev)

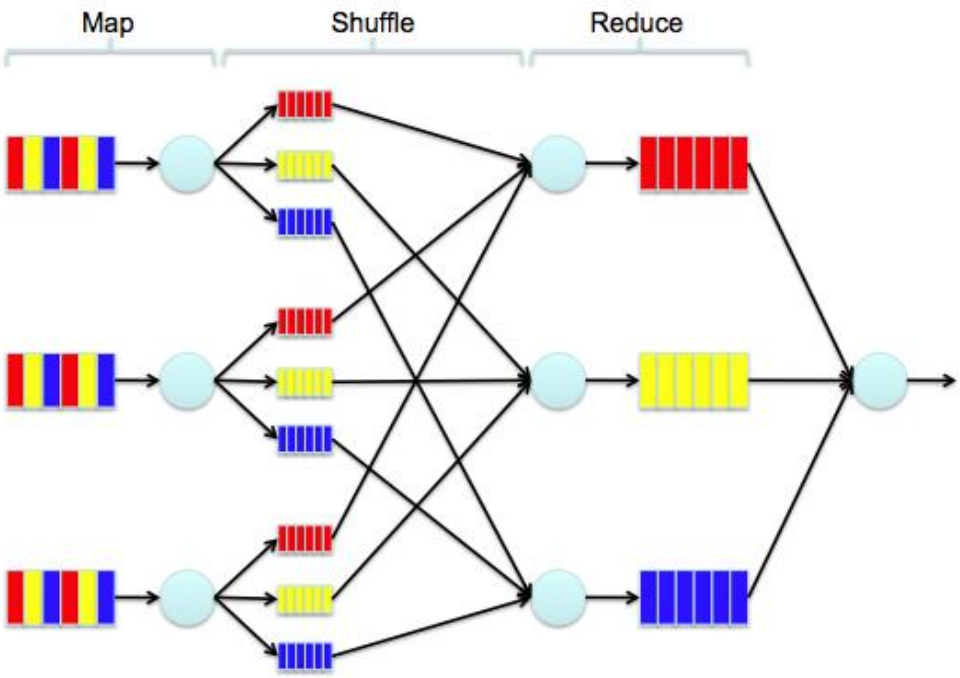
Poslední s čím knihovna přichází, jsou implementace základních algebraických datových struktur, jakou jsou monoid a grupoid. Grupoid [19] je uzavřená množina prvků, která má definovanou binární operaci, jež je také asociativní. Příkladem jsou celá čísla a operace sčítání, kdy výsledkem každého sčítání prvků této množiny je také celé číslo. Monoid [20]

je speciálním druhem grupoidu, který navíc obsahuje neutrální prvky. Příkladem jsou též celá čísla s operací sčítání a nula reprezentuje neutrální prvek, jelikož výsledkem sčítání nuly a jiného celého čísla je stále to samé celé číslo.

3.3.4 Technologie MapReduce

Obecně technologie MapReduce se zabývá paralelním zpracováním velkých data setů v distribuovaných systémech. Tento způsob zpracování je resilientní vůči chybám a navrhnutý tak, aby byl velice snadno škálovatelný, nezávisle na velikosti data setu. Prostředí Hadoop [21] se dá rozdělit na dvě hlavní části, *Storage* část a *Processing* část. *Storage* část definuje perzistentní vrstvu této technologie a má na starost distribuci data mezi jednotlivými systémy. *Processing* část se zabývá paralelním zpracováním dat a jeho řízením. Hadoop má obě části implementované v Javě

Proces MapRedue se skládá ze dvou sub-procesů (Obrázek 1), které mají pevně dané pořadí, jak naznačuje název. Prvním je *Map* proces, který z hlavního data setu, vytváří několik dalších data setů, které se skládají z párů (klíč, hodnota). Pak následuje proces *Reduce*, který s novými data sety na vstupu, redukuje vstupy do menších data setů. V praxi procesů *M*ap i *Reduce* může být více a také se využívá proces *Shuffle*, který spravuje a přesouvá výsledky mezi jednotlivými prvky distribuovaných systémů.



Obrázek 2: Schéma procesu MapReduce Zdroj: [22]

Tyto jednotlivé procesy lze definovat pomocí funkcí v Javě, ale to vyžaduje extensivní znalost Javy a paradigmatu funkcionálního programování. Z toho důvodu, autoři přidali řadu DSL jazyků, které pokrývají všechny domény, které technologie MapReduce

využívají. Většina těchto jazyků je postavena na derivaci z SQL, jelikož se jedná o velmi známý a deklarativní jazyk. Příkladem těchto jazyků jsou Pig a Hive.

Velkou inovací byl Apache Spark [23], který dovoluje psát nativně SQL dotazy, které jsou dále generovány do Hive dotazů nebo přímo funkcí MapReduce. Další z novinek je zabudovaná konzole, která umožňuje testování dotazů v reálném čase a odladňování procesů tudíž trvá podstatně kratší dobu.

3.4 Proč SQL?

Pokud bych abstrahoval téma mé práce, tak by cílem bylo deklarativní zpracovávání dat. Java 8 přišla celou řadou novinek, které přinášejí funkcionální programování v Javě. Datovody nyní umožňují jednoduší paralelní zpracování dat a chrání vývojáře od chybě náchylného více vláknového programování. Datovody za pomoci lambda výrazů a funkčních rozhraní umožňují programovat deklarativněji a kratší zápis. Pro začátek uvádím příklad implementace v Javě před verzí 1.8.

Výpis 8: *Ukázka zpracování data v Javě využívající syntax před Javou 8*

```
for (String text : list)
    {    int local = new Integer(text);
    if (local > 2)
    {
        result.add(local);
    }
}
```

Tato implementace využívá externího procházení kolekce, kdy potřebuje mít všechna data v jeden moment. Tento zápis může poměrně snadno a rychle bobtnat a je dosti košatý, a vývojář je nucen sám implementovat případné paralelní zpracování.

Výpis 9: *Ukázka zpracování data v Javě využívající syntax Javy 8*

```
list.stream()
    .map(t -> {return new Integer(t);})
    .filter(i -> {return i > 2;})
    .forEach(result::add);
```

Na druhou stranu při využití Javy 8 lze tento příklad vyjádřit deklarativněji. Vychází zde najevo, poměrně jasný filozofický rozdíl mezi kolekcemi a datovody, kdy prvky datovodu jsou tzv. „na požádání“ zpracovávány (*on-demand*) [24]. Tento termín lze také srovnat s „*lazy*“ objektově-relačním mapováním [25], kdy prvky kolekce jsou získány z databáze až v momentě, kdy jsou potřeba. Před jejich získáním jejich místo zaujímají zástupci (proxy objekty).

Oba dva tyto přístupy vyžadují i přes rozdílnou míru deklarativnosti imperativní přístup. Proto jsem se rozhodl hledat ještě více deklarativní zápis. Jako první možnost se nabízel vlastní doménově specifický jazyk, který by byl určen pro deklarativní zpracování dat v Javě. Tento přístup má sice možnost navrhnout syntax a gramatiku jazyka přímo na míru Javě, ale zase vyžaduje velké technické zázemí a vědomosti. Proto jsem se rozhodl využít z mého pohledu nejrozšířenějšího deklarativního jazyka a to SQL. Výše zmíněný příklad v zápisu v SQL by mohl vypadat takto:

Výpis 10: *Ukázka deklarativního zápis pomocí SQL jazyka*

```
SELECT number FROM list WHERE number>2;
```

Sám tento zápis už ukazuje úskalí, tohoto přístupu. Jazyk SQL je doménově specifický jazyk pro přístup do relačních databází. Jedná se zde o střet dvou sémantických modelů, kdy v Javě se pracuje s objektovým modelem a v databázích s relačním. Tento fakt řeší řada ORM frameworků , většinou skrze JDBC připojení. Tento důvod by mohl vést k vlastnímu DSL jazyku postaveného na derivaci z SQL. I přestože tato metoda je osvědčená (HQL, Pig, Hive), tak jsem se rozhodl zůstat u SQL a to ze dvou důvodů.

Prvním z nich je, že uživatel knihovny není nucen se učit nový nebo odlišný syntax jazyka a postačí si se standardy SQL-92, jelikož pro potřeby a omezení byla syntax „ořezána“. Ale i přesto si ponechává svoji deklarativnost. Druhým důvodem je rozšířenost SQL, pokud se odhlídne od zaměstnání spojených s vývojem softwaru, tak lze jazyk SQL považovat, za jeden z nejrozšířenějších. Tento jazyk najde uplatnění i v oborech jak je ekonomie nebo marketing, najde se řada lidí, kteří využívají jednoduché SQL dotazy do databáze na denní bázi.

Deklarativnost SQL lze ukázat na jednoduchém příkladu z oblasti bankovnictví, který jsem přebral z přednášky Lukase Edera na GeecCON 2014 konferenci v polském Krakově [10]. Velmi často se pracuje na daty týkajících se vkladů a výběrů z účtů. Banka si při každé operaci ukládá ID operace, datum operace, identifikační číslo účtu a kladnou nebo zápornou částku. Takže případný výpis z relační databáze, by mohl vypadat následovně (Tabulka 1).

Tabulka 1: Ukázka výpisu z relační databáze

ID operace	Datum	ID účtu	Částka
1236	12. 12. 2014	123	-135,40
1237	12. 12. 2014	159	+2010
1238	12. 12. 2014	123	+450,50
1239	12. 12. 2014	123	-470
1240	13. 12. 2014	169	+60,80

Požadavek zobrazit například operace účtu 123 za den 12. 12. 2014 je jednoduchá operace, s kterou se vypořádá JDBC, Hibernate nebo jiný ORM Framework poměrně snadno. Ale problém nastává v momentě, kdybychom chtěli dodat další sloupec se stavem účtu. Vývojář, který by dostal tento úkol má dvě možnosti, řešit to na úrovni aplikace (v Javě) nebo na úrovni databáze (SQL).

Pokud bych já dostal tento úkol a rozhodl se ho řešit v aplikaci pomocí Javy (pravděpodobně Java 6 či nižší díky bankovnictví), tak získat data z databáze není problém, ale pak následuje složitější procházení dat a počítání zůstatku. To bych pravděpodobně řešil mapou, kdy klíčem by byl hash vytvořený z čísla účtu nebo čísla účtu a data a hodnotou by byl zůstatek. Na druhou stranu lze tento problém vyřešit pomocí Window Function v SQL na 15 řádcích kódu.

Výpis 11: *Ukázka možného řešení z výše uvedeného problému pomocí jazyka SQL*

```
SELECT
    t.*,
    t.stav_uctu - NVL (
        SUM(t.castka) OVER (
            PARTITION BY t.id_uctu
            ORDER BY t.datum      DESC,
                        t.id_operace DESC
                ROWS BETWEEN UNBOUNDED PRECEDING
                AND   1         PRECEDING
        ),
    0) as stav
FROM operace t
WHERE t.id_uctu = 123
ORDER BY t.datum      DESC,
    t.id_operace DESC;
```

Zde lze vidět sílu deklarativnosti jazyka SQL. Kdyby přišel další požadavek, aby byly nejdříve uváděny operace vložení během dne. Tak to lze řešit upravením klauzulí ORDER BY. Naopak v Javě by bylo potřeba pravděpodobně refaktorovat implementaci.

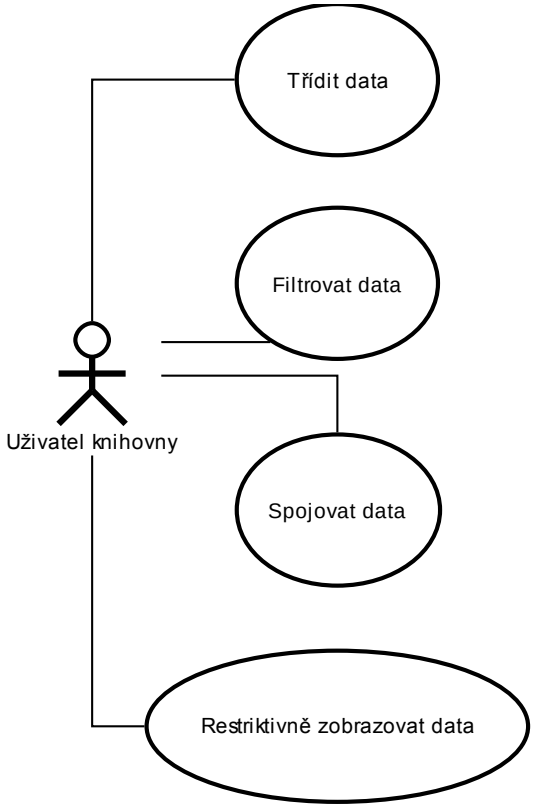
I přestože moje knihovna nepodporuje Window Functions, tak na tom případu lze demonstrovat sílu SQL a to se projevuje i ve faktu, že SQL je ovládáno i řadou lidí, kteří nepracují jako vývojáři. Ale využívají SQL pro jednoduché dotazování.

4 Specifikace zadání knihovny

4.1 Funkcionalita

Jak již bylo uvedeno v úvodu, tak v případě vývoje knihovny klienty jsou jednotliví uživatelé knihovny. V této specifické situace vývojář nebo vývojový tým své klienty nezná přímo a mnohdy je složité identifikovat jejich požadavky. Vzhledem k tomu, že knihovna interpretuje SQL dotazy, tak lze analogicky převzít požadavky na SQL jazyk. Tyto požadavky jsou upraveny vzhledem ke kontextu a vymezí tématu bakalářské práce (Obrázek 3).

Požadavky na SQL jazyk jsou nejdříve omezeny jen na požadavky na SELECT dotazy. Pro konkretizaci jsem v knihovně a v práci pracoval se standardem SQL-92. Požadavky byly omezeny tak, aby byl možno použít objektový sémantický model v Javě. Ze sedmi hlavních klauzulí [26] byl dotaz SELECT omezen na 4 (SELECT, FROM, WHERE a ORDER BY) a dále ještě částečně upraven. Tudíž funkce, agregace a vnořené dotazy nejsou podporované knihovnou.



Obrázek 3: Use Case diagram Zdroj: autor

4.1.1 Třídění

Uživatel knihovny ji může využít pro třídění dat podle jednotlivých atributů. Defaultně bude třídit vzestupně, pokud není specifikováno jinak v SQL dotazu (vzestupně nebo sestupně). Oproti standardnímu SQL jazyku není možné třídit podle relativních pozic.

4.1.2 Filtrování

Knihovna umožňuje filtrovat data podle jednotlivých kritérií uvedených v klauzuli WHERE [27], tyto kritéria můžou být spojovány logickými spojkami AND a OR, případně negací NOT. Počet kritérií není omezený, ale gramatika je omezena. Knihovna podporuje tyto operátory (Tabulka 2).

Tabulka 2: Tabulka podporovaných operátorů

Operátor	Popis	SQL	Knihovna
=	Rovná se	Ano	Ano
<>	Nerovná se	Ano	Ano
>	Větší než	Ano	Ano
<	Menší než	Ano	Ano
>=	Větší nebo rovno	Ano	Ano
<=	Menší nebo rovno	Ano	Ano
BETWEEN	Mezi	Ano	Ne
LIKE	Jako (dle vzoru)	Ano	Ne
IN	V (z množiny)	Ano	Ne
IS / IS NOT	Porovná s hodnotou NULL	Ano	Ne
IS NOT DISTINC FROM	Rovná se nebo obě hodnoty jsou NULL	Ano	Ne
AS	Přiřazení aliasu	Ano	Ne

Syntax jednotlivých kritérií je omezená, pravým operandem je vždy reference na atribut včetně aliasu [28] zdroje a pravým operandem je hodnota (touto hodnotou může být sekvence znaků, logická hodnota nebo číslo).

4.1.3 Spojování

Knihovna podporuje spojování dat z různých zdrojů na základě společného pole. Toto společné pole je vyjádřeno podmínkou, kdy na obou stranách je referencí. V rámci knihovny je možné použít jen jednoduchý JOIN(*simple join – inner join*) [29]. Případné konstrukce FULL JOIN nebo OUTER JOIN knihovna nepodporuje.

4.1.4 Restrikce

Uživatel také může vybrat konkrétní části zpracovaného data setu, může vybrat jen konkrétní atributy, v případě konstrukce JOIN může vybrat jak atributy, tak konkrétní entity. V případě konkrétního výběru musí před atributem nebo entitou být alias jejího zdroje. Druhou možností je hvězdička, která nikterak neomezuje daný data set.

4.2 Vstup

Hlavním procesem knihovny je zpracování dat, které naplňuje požadavky specifikované v předchozí kapitole. Tento proces se rozpadá na řadu sub-procesů, které interpretují jednotlivé části SQL dotazu. V každém případě prvním vstupem do procesu je SQL dotaz v podobě sekvence znaků. Pokud je tento řetězec prázdný nebo hodnotu NULL, tak knihovna jako výstup vrací hodnotu NULL. V případě, že se jedná o nevalidní nebo knihovnou nepodporovaný SQL dotaz, tak knihovna vyhazuje výjimku (`SQL2StreamException` - Výpis 12).

Druhým vstupem jsou data. Tato data jsou reprezentována instancemi tříd uživatele knihovny. Podobu libovolných dat jsem definoval, tak že se jedná podobu, jež pokryje, co největší množství scénářů uživatelů a nepřinese větší problémy při implementaci. Knihovna má na třídy zpracovaných objektů jeden nárok. Touto podmínkou je, aby všechny datové typy, které jsou atributy daného objektu (případně vlastní třídy jako atributy) implementovali interfejs `Comparable`. Knihovna využívá tohoto rozhraní při interpretaci podmínek v klauzulích WHERE a FROM a také pro interpretaci klauzule ORDER BY.

Při využití SQL pro dotazování do databáze jsou jednotlivá data uchovávána v podobě jednotlivých záznamů v tabulce. Proto je potřeba zvolit vhodnou datovou strukturu, ve které budou uchovávána data.

Výpis 12: Implementace třídy SQL2StreamException

```
public final class SQL2StreamException extends StandardException
    {
    private static final long serialVersionUID = 1117459919302661139L;
    protected SQL2StreamException(Exception e)
    {
        super(e);
    }
    protected SQL2StreamException(String message)
    {
        super(message);
    }
    protected SQL2StreamException(String message, Throwable cause)
    {
        super(message, cause);
    }
}
```

4.2.1 Collections

Jednou z možností je interfejs `Collection`, který nabízí široké možnosti pro manipulaci s daty a také od Javy 8, také nabízí metodu `stream()`, která vrací datovod daného typu kolekce. Tato metoda také využívá jednu z novinek, výchozí metody rozhraní, kdy interfejs `Collection` má defaultní implementaci metody `stream()`. Běžný uživatel knihovny by si mohl s kolekcí vystačit, jelikož pokrývá široké spektrum scénářů a poskytuje mnoho implementací.

4.2.2 Iterable

Jelikož data jsou v knihovně zpracovávána pomocí datovodů, tak velká část metod z kolekcí by nebyla využita, tak je zde možnost postoupit o úroveň abstrakce výš a zvolit interfejs `Iterable`. Tím se také pokryje větší množství scénářů, ve kterých lze knihovnu využít.

Interfejs `Iterable` nabízí přímo externí iteraci v podobě metody `iterator()` nebo metodu `spliterator()`, jehož návratová hodnota lze použít pro vytvoření datovodu pomocí statické metody `stream(Spliterator<T> spliterator, boolean parallel)` z třídy `StreamSupport` [30]. Druhým parametrem je logická hodnota, která určuje, zda výstupní datovod má být paralelní nebo ne.

4.2.3 Stream

Vstupem do knihovny nemusí být jen implementace kolekce, jako vstup lze použít již připravený datovod. Tento přístup by umožnil zpracovávat například třída `InputStream` ze souboru nebo z připojení přes síť. Navíc stále pokrývá celé využití pro interfejs `Iterable`. Zde ale bohužel lze narazit na nedostatky Javy a v podobě implementace generických typů.

Při implementaci jednotlivých interpretů v knihovně je potřeba znát typy objektů, z důvodu využívání reflexe. Tento problém se dal řešit řadou způsobů, ale ani jeden z nich není elegantní a jednoduchý.

Nejjednodušší řešení je převést datovod na seznam, následně zjistit typ jeho prvků a následně převést zpět na datovod. Nabízí se otázka, proč není vstupem rovnou seznam a ten pak převeden na datovod? Druhým možným řešením je dodávat typ objektu jako další parametr, toto řešení není již tak nevhodné, ale na druhou stranu by to uškodilo aplikačnímu rozhraní knihovny, kdy nutí uživatele dávat další parametr navíc.

Další možností je využít reflexe, kdy se využívá třída generického předka [31]. V případě knihovny by se jednalo o `BaseStream<T>.`

Výpis 13: *Ukázka části řešení pro zjištění typu generika v Javě*

```
public Class<?> returnedClass(Class<?> clazz)
{
ParameterizedType parameterizedType = (ParameterizedType)
            clazz.getGenericSuperclass();
    return (Class<?>) parameterizedType.getActualTypeArguments()[0];
}
```

Jedná se o elegantní a mnohem čistší přístup. Na druhou stranu silně využívá reflexe, což může mít negativní vliv na výkon a rychlost celého procesu.

4.2.4 Streamable

Interfejs, který bohužel Java 8 nepřinesla je `Streamable` (Výpis 14) nebo jemu podobné. Proto je jednou z navrhovaných možností definování vlastního rozhraní. Tento interfejs by měl vytvořit obecné rozhraní pro třídy, jež jsou schopné vytvářet nebo poskytovat datovod.

Výpis 14: *Moje implementace rozhraní Streamable*

```
public interface Streamable<T>
{
    public Stream<T> stream();
    public Class<T> getStreamedClass();
}
```

Interfejs by měl dvě metody, které vyhovují potřebám knihovny. Nicméně toto řešení vyžaduje poměrně kostrbatou implementaci na straně uživatele knihovny, jelikož většina scénářů vychází s kolekcí na vstupu, které neimplementují toto rozhraní. Tento fakt by šel řešit pomocí připravené obálky.

Výpis 15: Ukázka možného řešení obálky pro kolekce

```
public class WrapperIterable2Streamable<T> implements Streamable<T>
{
    private Iterable<T> iterable;
    private Class<T> clazz;
    public WrapperIterable2Streamable(Iterable<T> iterable, Class<T> clazz)
    {
        this.iterable = iterable;
        this.clazz = clazz;
    }
    @Override
    public Stream<T> stream()
    {
        return StreamSupport.stream(iterable.spliterator(), false);
    }
    @Override
    public Class<T> getStreamedClass()
    {
        return clazz;
    }
}
```

I přes připravenou obálku, se jedná o nevhodné řešení z důvodu vytváření košatého „boilerplate“ kódu.

4.2.5 Streamer

Další možností je vlastní třída `Streamer` (Výpis 16), která pomocí reflexe prohledává třídu zadanou v konstruktoru a při volání metody a dodání instance třídy `Object`, tak vrací datovod typu zadané třídy.

Třída využívá funkcionálního paradigmatu a jednotlivé části jsou napsány jako funkce. Nejdříve si zjistí všechny metody, které má daná třída. Pak vybere jen ty, které vrací datovod, následně vyfiltruje jen ty, které nemají parametr. Posledním krokem je kontrola typu návratové hodnoty přes generického předka. Pokud těmto kritériím vyhovuje více metod, tak se vybere první, jelikož v tento moment, třída by měla jedinou možnost, jak vybírat dále a to přes název metody, ale to je zavádějící kritérium. Z důvodu bezpečnosti, po té co se metodou `invoke(Object obj, Object… args)` se vstupním objektem jako parametrem získá datovod, tak se ještě volá metoda `sequential()`, aby zabezpečila to, že

vracený datovod nebude paralelní a tento pomocný nástroj bude mít konkrétní chování, bez nahodilého chování.

Výpis 16: *Moje implementace třídy Streamer*

```
@SuppressWarnings("unchecked")
public Stream<T> toStream(Object object) throws Exception
    {
    Optional<Method> first = null;
    try
    {
        first = new StreamReturnTypeFunction()
                    .apply(object.getClass())
                    .filter(new AgregateFunction(clazz))
                    .findFirst();
    } catch (Exception e)
    {
        throw new Exception("Error while finding stream method for object: " +
            object + " |" + e);
    }
    if (!first.isPresent())
    {
       throw new Exception("No method without parametres found for object: " +
            object + ", thats returns Stream<" + clazz.getSimpleName() + ">");
    }
    return ((Stream<T>) first.get().invoke(object, null)).sequential();
}
```

4.3 Výstup

Z pohledu užívání knihovny musí být výstup hlavního procesu uživatelsky přívětivý. I pokud knihovna funguje správně a zpracovává data, tak pokud má nevhodný výstup, tak ji uživatelé nebudou využívat.

Obecně většina knihoven zpracovávajících data nebo vracející data z databáze vrací nějakou formu uspořádané n-tice prvků. V případě, že vrací více záznamů, tak se z toho stává dvou rozměrná uspořádaná n-tice prvků. V případě Javy nejčastější implementace této n-tice bývá seznam nebo pole.

V případě frameworku Hibernate a výběru více záznamů z databáze se vrací seznam objektů nebo seznam polí objektů. Druhý případ není úplně ergonomický pro uživatele, ale jelikož autoři knihovny předem neví jak velké je toto pole nebo jaké typy si bude uživatel vracet a současné možnosti Javy nenabízejí nativnější přístup. V tento moment musí programátor implementovat metodu nebo funkci, která n-tici mapuje na daný objekt.

Proto knihovna vrací objekt, jenž implementuje interfejs `Iterator`, jehož prvky jehož prvky jsou buď typu `Element` (interfejs `Element` uvedeno níže v práci) nebo již namapovaný objekt v případě, pokud uživatel poskytne mapovací funkci v parametru.

4.4 Podpora SQL

Z SQL jazyka moje knihovna využívá příkaz SELECT, který se obecně využívá k dotazování do relačních databází. Syntaxi příkazu SELECT lze rozdělit do 7 částí, z čehož jsou dvě části povinné SELECT a FROM. Příkaz SELECT vyjádřit v textové podobě jako řetězec znaků.

Výpis 17: *Ukázka SQL dotazu*

```
SELECT name FROM students WHERE age > 25;
```

Tento způsob zápisu je jasný a čitelný, ale pro strojové zpracování je stromová datová struktura vhodnější, jelikož lépe vyjadřuje vztahy a hierarchii. Lze najít mnoho způsobů, jak daný strom může vypadat. Díky tomu, že strom je acyklický neorientovaný graf, tak lze jeho uzle rozdělit na další uzle (respektive stromy). Tohoto faktu knihovna využívá a jednotlivé části může interpretovat nezávisle na sobě, ale pevně daném pořadí. Výše uvedený dotaz lze grafem reprezentovat následovně (Výpis 18).

Ve své knihovně jsem se rozhodl použít parser z knihovny FoundationDB [32], implementace vlastního parseru by mohla být případným pokračováním a rozšířením této knihovny. Parser z knihovny FoundationDB poskytuje dvě důležité funkcionality. Prvním z nich je validace SQL a druhá je převedením do AST. V následující části proberu všech 7 části SELECT příkazu a možnost převoditelnosti jejich částí na datovody.

Výpis 18: *Výpis AST stromu převedeného dotazu z* ***Výpis 17****pomocí parseru z knihovny FoundationDB*

```
com.foundationdb.sql.parser.CursorNode@504bae78
name: null
updateMode: UNSPECIFIED
statementType: SELECT
resultSet:
    com.foundationdb.sql.parser.SelectNode@3b764bce
    isDistinct: false
    resultColumns:
        com.foundationdb.sql.parser.ResultColumnList@759ebb3d

         [0]:
        com.foundationdb.sql.parser.ResultColumn@484b61fc
        exposedName: name
        name: name
        tableName: null
```

```
        isDefaultColumn: false
        type: null
        expression:
            com.foundationdb.sql.parser.ColumnReference@45fe3ee3
            columnName: name
            tableName: null
            type: null
    fromList:
        com.foundationdb.sql.parser.FromList@4cdf35a9

         [0]:
        com.foundationdb.sql.parser.FromBaseTable@4c98385c
        tableName: students
        updateOrDelete: null
        null
        correlation Name: null
        null
    whereClause:
        com.foundationdb.sql.parser.BinaryRelationalOperatorNode@5fcfe4b2
        operator: >
        methodName: greaterThan
        type: null
        leftOperand:
            com.foundationdb.sql.parser.ColumnReference@6bf2d08e
            columnName: age
            tableName: null
            type: null
        rightOperand:
            com.foundationdb.sql.parser.NumericConstantNode@5eb5c224
            value: 25
            type: INTEGER NOT NULL
```

4.4.1 Klauzule SELECT

Jedná se o jednu ze dvou povinných části SQL dotazu [33]. Syntaxe této klauzule je následující (Výpis 19):

Výpis 19: *Syntax SELECT klauzule SQL dotazu*

```
SELECT [ALL|DISTINCT|DISTINCTROW]
{* | <specific_row1> [,
    <specific_row2>[, ...] }
```

V první fázi knihovna podporovala pouze „*“, kdy vracela pouze celé objekty. Později byla rozšířena o možnost vrátit specifický sloupec nebo objekt, čehož je dosaženo pomocí metody map. Takže syntaxe, která lze použít vypadá následovně (Výpis 20).

Výpis 20: *Syntax SELECT klauzule podporované knihovnou*

```
SELECT [* | <alias>.<specific_row>]*?
```

Knihovna nepodporuje modifikovaný text v záhlaví a ani výpis modifikovaných údajů.

4.4.2 Klauzule FROM

Tato klauzule je druhá povinná část, obecně v SQL se používá pro výběr tabulek, ze kterých chceme vrátit data. Jelikož v Javě nepracujeme s relačním modelem, ale již s jednotlivými entitami mapovanými na jednotlivé třídy, kde vztahy mezi tabulkami jsou vyjadřovány dědičností a kompozicí.

Implementace interpretace konstrukce JOIN sebou nese řadu aspektů, které ovlivňují celou knihovnu. Ať už se jedná o typy vstupů do knihovny nebo konkrétní implementaci konstrukce JOIN. Tato část je popsána v samostatné kapitole. V této verzi knihovny jsem se rozhodl podporovat pouze obyčejnou konstrukci JOIN, bez její modifikací. Jednotlivé kolekce, které figurují jako vstupy do knihovny a mají aliasy „t“, „u“, „r“ a „v“ v tomto pořadí. I přestože knihovna díky své implementaci je schopna interpretovat více jak 4 kolekce, které se spojují, tak již při třetí konstrukci JOIN se výkon knihovny rapidně zhoršuje.

Možná syntax pro využití v knihovně vypadá následovně:

Výpis 21: *Syntax FROM klauzule podporované knihovnou*

```
FROM [<název_kolekce>|<nazáv_třídy> <alias>] [JOIN <název_kolekce>|<nazáv_třídy>
<alias> ON <alias>.<atritbut><operátor><alias><atribut>] *? (max 3)
```

4.4.3 Klauzule WHERE

Klauzule WHERE (Výpis 22**)** je klauzule podmínek, které jsou spojovány logickými spojkami AND, OR a negací NOT. Zde je potřeba si uvědomit, že mezi jednotlivými podmínkami existují vztahy a hierarchie vyjádřena logickými spojkami a závorkami. Z toho důvodu je potřeba využít stromovou strukturu při strojovém zpracování.

Výpis 22: *Syntax WHERE klauzule podporované knihovnou*

```
WHERE <podmínka> [(AND|OR) <podmínka>]*?
```

4.4.4 Klauzule ORDER BY

Třídění kolekcí je jedna z častých činností nad kolekcemi. Použití této klauzule omezuje použití paralelního zpracování a lze pokládat za úzké hrdlo výsledného datovodu. Knihovna podporuje třídění podle více kritérií, v obou směrech setřídění. Syntaxe je následující

Výpis 23: *Syntax ORDER BY klauzule podporované knihovnou*

```
ORDER BY [<specific_row> [ASC|DESC]?]*
```

5 Použité technologie

Ve své práci jsem využil technologií třetích stran, kromě standardních nástrojů a knihoven, které se využívají při vývoji knihoven a aplikací, jsem využil jedné knihovny. Tou knihovnou je parser od FoundationDB. Jedná se o knihovnu, které je na úrovni produkčního využití.

5.1 FoundationDB

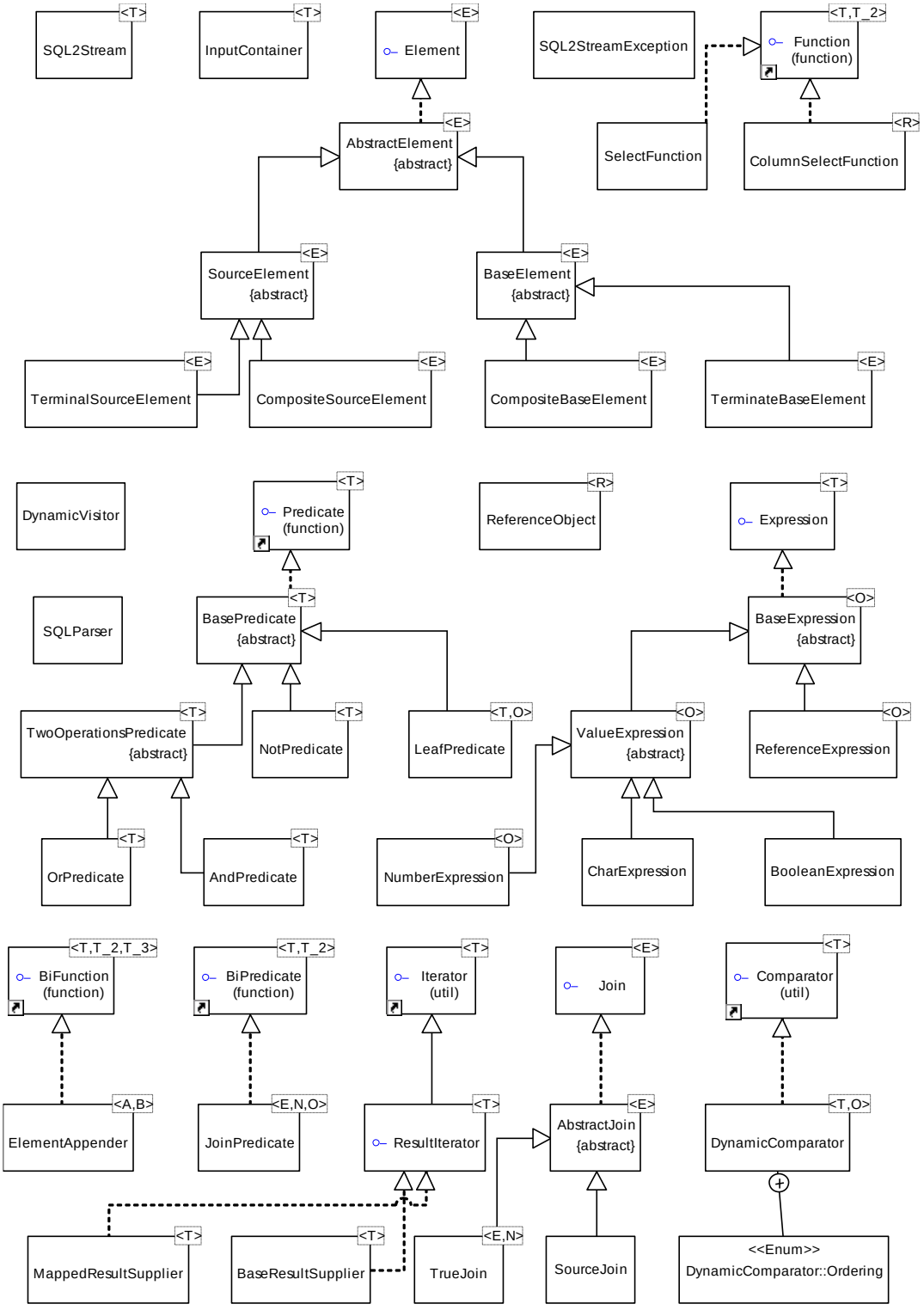
FoundationDB je obecně více vrstvá NoSQL databáze, která umožňuje práci s nerelačními daty pomocí SQL jazyka. Tato databáze využívá klíč-hodnota dvojic, nad kterými částčně implementuje ACID pravidla a umožňuje CRUD operace.

Jakožto NoSQL databáze nabízí větší možnost škálovatelností než klasické relační databáze. Tato databáze nabízí velké množství API (Python, Ruby, node.js, PHP, Java Go, .Net a C). Letos v březnu tato databáze byla koupena společností Apple.

Od FoundationDB jsem se rozhodl využít parser SQL jazyka. Knihovna nabízí `SQLParser` a rozhraní `Visitor`, pomocí kterého lze výsledný AST strom procházet.

6 SQL2Stream knihovna

6.1 Class diagram knihovny



Obrázek 4: Zjednonušený Class diagram celé knihovny Zdroj:autor

6.1.1 Obecná pravidla pro návrh aplikačních rozhraní knihoven

Jak jsem již uvedl v úvodu práce, tak návrh knihovny si sebe nese řadu specifik, na které si musí autor pozor. Jelikož i drobné zaváhání může knihovnu paralyzovat tak, že její využití bude velice omezené. Při vytváření knihovny jsem se držel obecných zásad OOP a dále několika níže uvedených pravidel/doporučení, jak při vytváření API-knihovny, tak uvnitř v implementaci. A to z důvodu, že v budoucnosti, pokud by došlo k rozvoji knihovny, tak nemusím být jediným programátorem, který se bude účastnit vývoje.

Prvním pravidlem, kterého jsem se držel, je definování a užívání silných termínů [34] a to z pohledu, že i pří prvním setkání s knihovnou, by měl uživatel pochopit funkce a role jednotlivých tříd a metod. Tudíž aby jejich byly nativní a schéma názvu odpovídalo její roli nebo funkci. V rámci toho pravidla je dobré si vzít i příklad ze základních knihoven Javy případně z knihoven, které vystupují jako průmyslové standardy.

Druhým pravidlem je symetrie v návrhu a ve smyslu, že programátor by se měl držet zavedeného názvosloví a postupů, které jsou v knihovně již zavedeny [35]. Toho lze dosáhnout například vytvořením společného rozhraní.

Třetím pravidlem, které má až neočekávanou hodnotu, je dodržování pořadí parametrů. Jestliže uživatel již proniknul do API knihovny a má základní přehled, tak bývá častou chybou, že přetěžovaná metoda mění pořadí parametrů než je zvykem v knihovně.

Tato tři pravidla by měla pomoci knihovně vytvořit API, které i při změně požadavku, nemusí být změněno. Tím se dosahuje zpětné kompatibility při vydání nové verze knihovny s novou funkcionalitou nebo opravenými nedostatky. Z pohledu návrhu je vždycky lepší mít API na začátku, co nejužší [36] a zahrnout do něj jen to nejdůležitější, jelikož vždycky je možné do API něco přidat. Bohužel pokud se z API zužuje v nové verzi nebo se mění jeho chování, tak to může mít neblahé důsledky na aplikaci uživatele.

6.1.2 Veřejné rozhraní-knihovny

Veřejné API knihovny má 4 konstruktory, podle jednotlivých vstupů a dvě metody, které vrací výstup z procesu zpracování dat (Výpis 24). Při implementaci knihovny jsem uvažoval do rozhraní knihovny také zahrnout možnost volat jednotlivé interprety, které by vracely predikáty, komparátory nebo proud spojených dat. Jelikož jednotlivé interprety mohou být pro uživatele užitečné i samy o sobě. Ale do první verze knihovny jsem rozhodl je nezařadit.

Do veřejného rozhraní knihovny též byly přidány některé implementace `Elementu`, aby koncový uživatel s nimi mohl nakládat a manipulovat.

Výpis 24: Část aplikačního rozhraní knihovny

```
public SQL2Stream(String sql, InputContainer<T> t)
public <U> SQL2Stream(String sql, InputContainer<T> t, InputContainer<U> u)
public <U, R> SQL2Stream(String sql, InputContainer<T> t, InputContainer<U> u,
            InputContainer<R> r)
public <U, R, V> SQL2Stream(String sql, InputContainer<T> t,
            InputContainer<U> u, InputContainer<R> r,
            InputContainer<V> v)

@SuppressWarnings({ "unchecked", "rawtypes" })
public <F> ResultSupplier<F> interpret() throws SQL2StreamException

@SuppressWarnings({ "unchecked", "rawtypes" })
public <F> ResultSupplier<F> interpret(Function<List<? extends Object>, ?>
            mappingFunction) throws SQL2StreamException
```

6.2 Vstup a výstup

Definovat libovolný vstup do knihovny je obecně těžké, ke konci tvorby práce jsem chtěl zvolit variantu přepravky datovodu spolu třídou typu datovodu, ale jelikož API již bylo definované a jeho změna by vyžadovala změnu implementace, tak jsem se rozhodl zůstat u současné varianty.

Jako vstup do knihovny jsem zvolil přepravku `InputContianer<T>`, důvodu, že jsem se rozhodl jít bezpečnější cestou a uživatele knihovny typ vstupu definovat třídou na vstupu v konstruktoru této přepravky (Výpis 25: Konstruktor přepravky InputContainer Druhým parametrem konstruktoru je interfejse `Iterable`, pro něj jsem se rozhodl z důvodu interpretace klauzule JOIN.

Výpis 25: Konstruktor přepravky InputContainer

```
public InputContainer(Iterable<T> iterable, Class<T> clazz) {
    this.clazz = clazz;
    this.iterable = iterable;
}
```

Pokud by byl na vstupu jen jeden datovod a ostatní interfejs `Iterable`, tak již je to možné, ale nese to sebou řadu omezení. Prvním je, že by datovod musel mít pevné místo v SQL dotazu, aby bylo možné ho interpretovat. Druhým problémem je, že interfejs `Iterable` nebo kolekce dovolují spojování interpretovat pouze jednou a následně vytvořit datovod. Pokud bychom měli datovod na vstupu, tak se musí interpretovat víckrát, pro každý jeho prvek.

V případě mé knihovny jsem se rozhodl nabídnout uživateli dvě možnosti na výstupu. V obou případech vracím iterátor, který implementuje interfejs `Iterator` (Výpis 26, Obrázek 5), takže má implementovanou metodu `next()`, která vrací další prvek v pořadí, a metodu hasNext. Třetí metoda `remove()` z rozhraní `Iterator` má defaultní implementaci a vyhazuje `UnsupportedOperationException`.

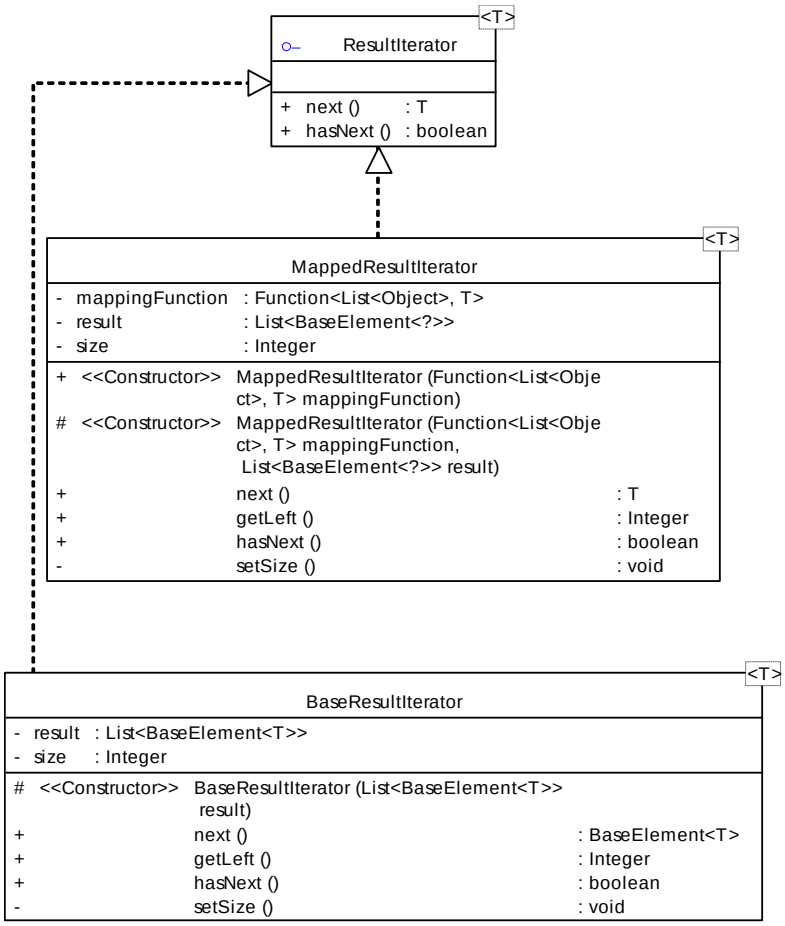
Výpis 26: *Rozhraní ResultIterator, které vystupuje jako výstup z knihovny*

```
public interface ResultIterator<T> extends Iterator<T>
{
    @Override
    public T next();

    @Override
    public boolean hasNext();

}
```

Výpis 27*: Deklarace mapovací funkce*

```
Function<List<Object>, T> mappingFunction
```

Veřejné API-knihovny má konstruktory třídy SQL2Stream, která má dvě veřejné metody `interpret()` a `interpret(Function<List<? extends Object>, ?> mappingFunction)` s mapovací funkcí jako parametrem (Výpis 27).

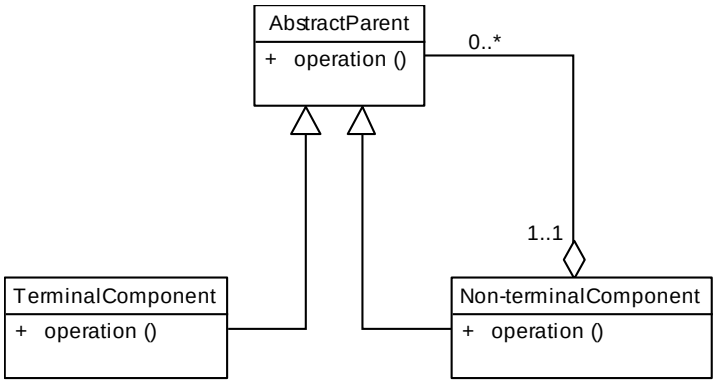


Obrázek 5: Detailní Class diagram výstupu z knihovny Zdroj: autor

Uživatel knihovny je v tuhle chvíli zodpovědný za implementaci této mapovací funkce. Může spoléhat na to, že na vstupu funkce má list, který reprezentuje uspořádanou n-tice prvků, která byla vytvořena na základě klauzule SELECT SQL dotazu. Pořadí těchto prvků, které je pevně dáno dotazem.

6.3 Návrhový vzor složenina

Návrhový vzor interpret sám o sobě využívá vzor složenina (Obrázek 6) [37]. Hlavním cílem tohoto vzoru je, aby klient (knihovna) mohla zacházet stejně jak s jednoduchými dotazy, tak se složitějšími. Vzor umožňuje rekurzivní volání a vyjadřovat vztahy 1:N a 1:1. Stejně jako u návrhového vzoru interpret lze rozdělit prvky na koncové a nekoncové se společným rozhraním. V jednodušší implementaci, nemusí být ani rozděleny na koncové a nekoncové. Příkladem může být instance mající list dalších instancí své třídy, které vystupují jako potomci. Nejznámější implementací toho vzoru je třída File v základní knihovně Javy. Tato třída může představovat jak soubor, tak složku.



Obrázek 6: Obecný návrhový vzor složenina Zdroj: autor

Ve své knihovně jsem se snažil využít generické implementace, která umožňuje uchovávat jak data na vstupu, tak využít ho při zpracovávání a případně i na výstupu. Na vrcholu stojí rozhraní `Element<E>` (Výpis 28,Obrázek 7).

První část rozhraní umožňuje práci s prvkem a jeho obsahem, druhá část nahrazuje návrhový vzor návštěvník a sama má metody na svoje procházení. Pod tímto rozhraním stojí dvě abstraktní třídy, které ho implementují. Prvním je `BaseElement`, který se používá pro uchování jednotlivých prvků dat druhým je `SourceElement`. Zde jsou také doplněny navíc přístupové metody, jež dovolují přístup k typu uvnitř. Pod ním již stojí koncové a nekoncové potomci těchto abstraktních tříd

Výpis 28: *Rozhraní Element*

```
public interface Element<E> extends Cloneable {
    public Class<E> getClazz();
    public void setClazz(Class<E> clazz);
    public String getAlias();
    public void setAlias(String alias);
    public boolean hasNext();
    public Element<?> visit(String alias);
    public Element<?> getLast();
    public void setLast(Element<?> element);
    public Element<?> getPenultimate();
    public String getStrategy();
    public Element<E> clone() throws CloneNotSupportedException;
}
```

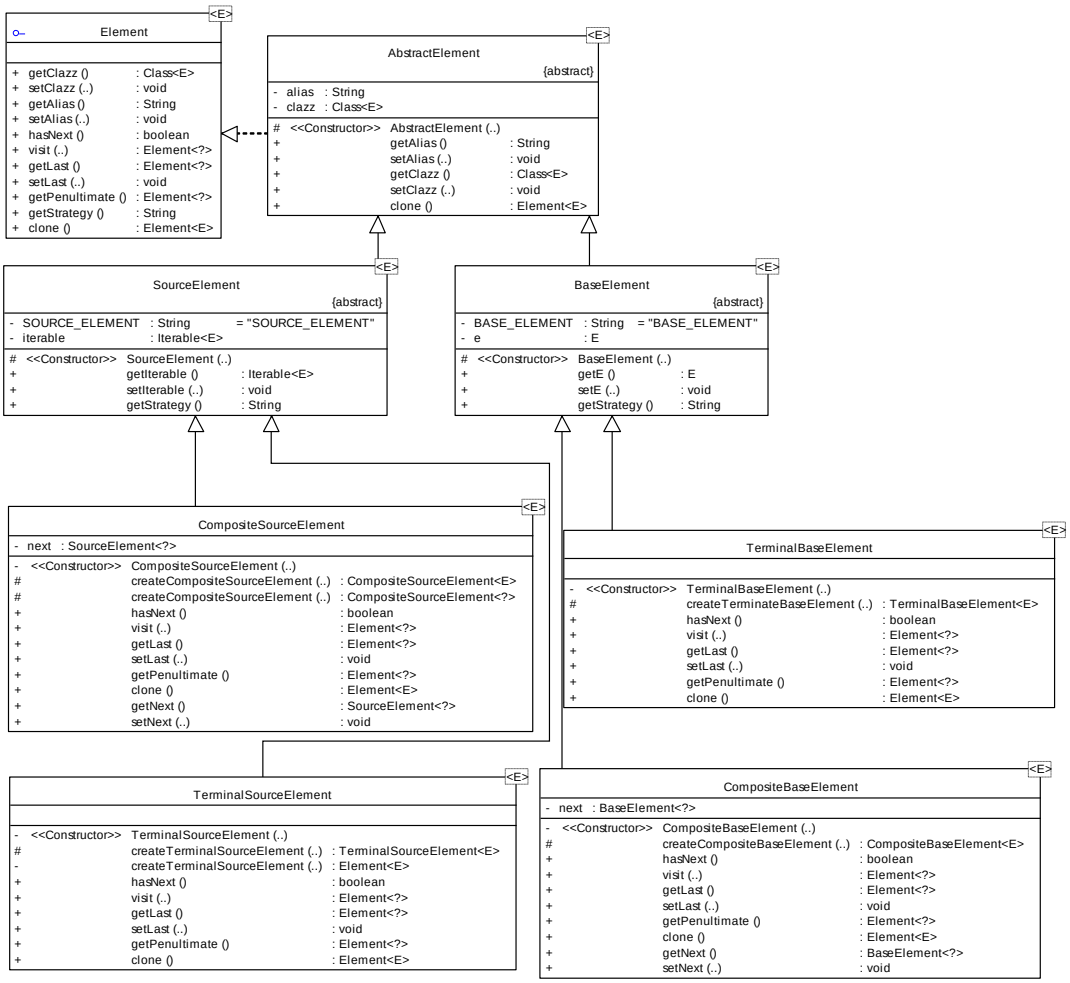
Tato implementace má velkou nevýhodu v tom, že pokud již mám strom a chci, zařadit něco na jeho konec (připojit), tak koncový element musí být nahrazen nekoncovým a až následně může být další element připojen. Proto je jednoduší přidávat element na začátek. Druhou nevýhodou je, že má pouze jedno směrné vztahy, takže je možnost procházet strukturu pouze ze shora dolů.

Pro snadnější práci jsem vytvořil nástroj pro spojování elementů (Výpis 29), jež implementuje funkční interfejs `BiFunction`, na vstupu jsou dva elementy (`Element<A>` a

`Element<B>`) a na výstupu je již `Element<A>`, jelikož druhý element byl připojen na jeho konec. Tento nástroj je využívám hlavně při interpretaci klauzule JOIN.

Výpis 29: *Zkrácená ukázka implementace nástroje pro spojování prvků*

```
final class ElementAppender<A, B> implements BiFunction<Element<A>, Element<B>,
                Element<A>>
{
    @Override
    public Element<A> apply(Element<A> e, Element<B> n)
    {
        //...
    }
}
```

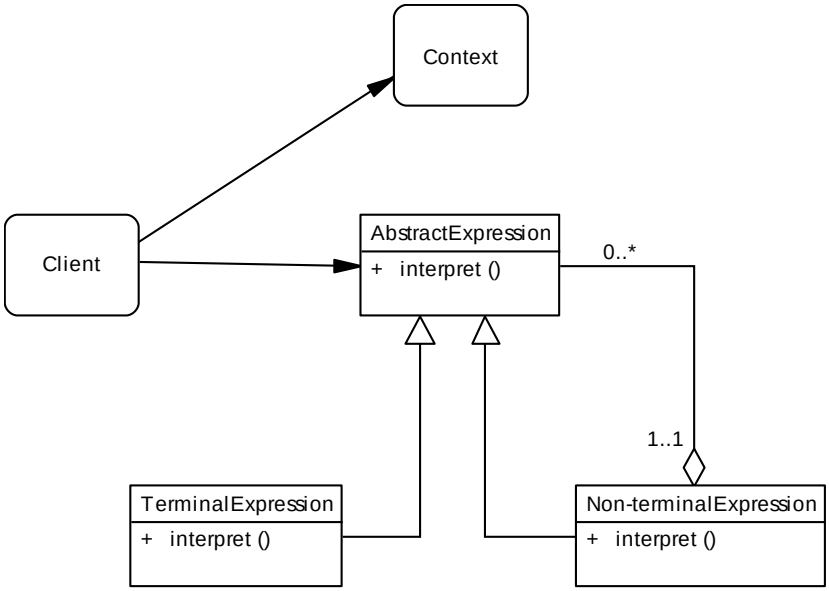


Obrázek 7: Class diagram implementace prvku Element v knihovně Zdroj: autor

6.4 Návrhový vzor interpret

Návrhový vzor interpret (Obrázek 8) byl poprvé představen v roce 1994 [38], i přestože jeho využití je značně omezené, tak možnost efektivně interpretovat ostatní jazyky (DSL

nebo i jazyky jako jsou not či angličtina a čeština). Návrhový vzor se skládá z 5 částí. Prvním z nich je výraz – abstraktní předek (případně interfejs), který definuje jeho rozhraní (metodu `interpret()`). Tato třída má dva potomky, prvním z nich je nekoncový výraz, který implementuje metodu interpret a dále obsahuje referenci na další výrazy. Druhým potomkem je koncový výraz, který implementuje také metodu interpret. Takže pomocí skládání těchto výrazů vzniká strom. Další částí je klient, který konstruuje výrazy a volá spouští interpret. Poslední částí je samotný kontext, předmět interpretace. Tento kontext lze předávat do výrazů skrze parametry metody interpret nebo již v konstruktoru.

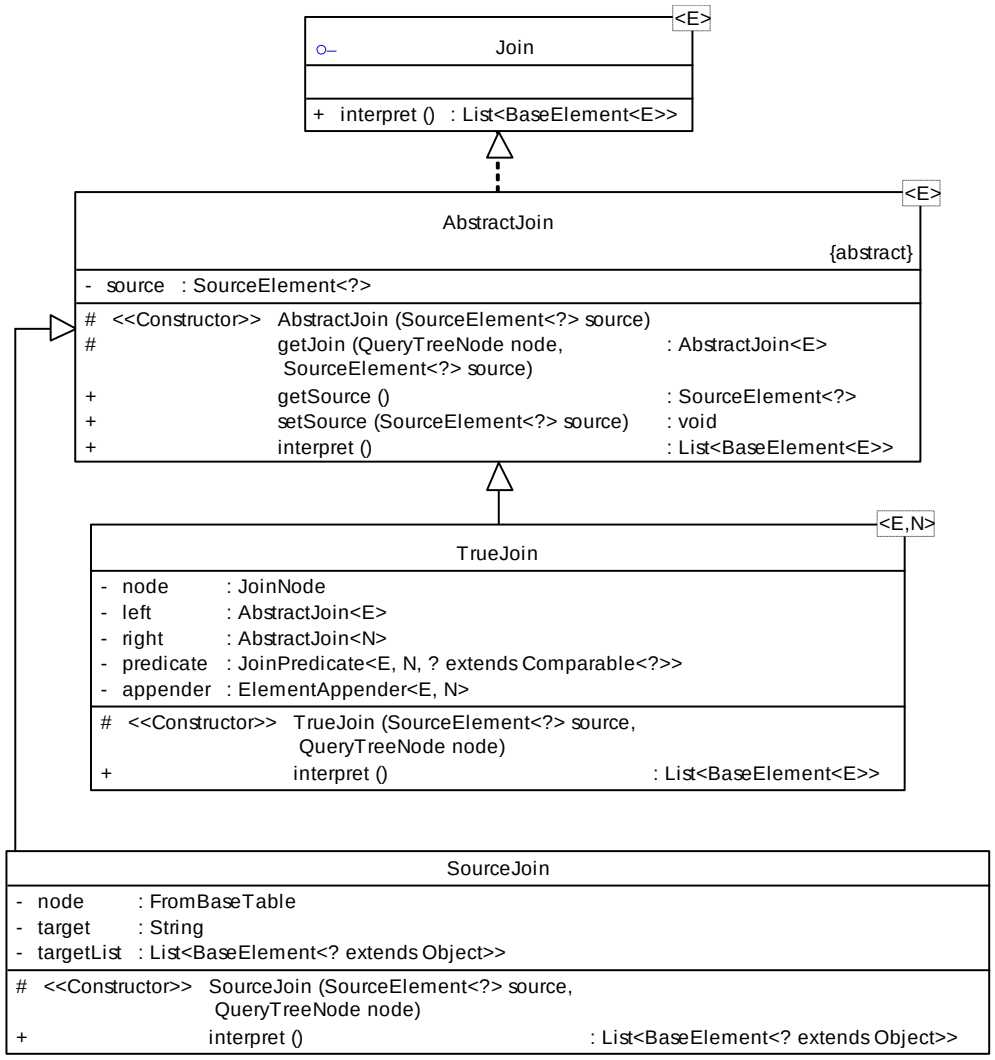


Obrázek 8: Obecný Class diagram návrhového vzoru interpret Zdroj: autor

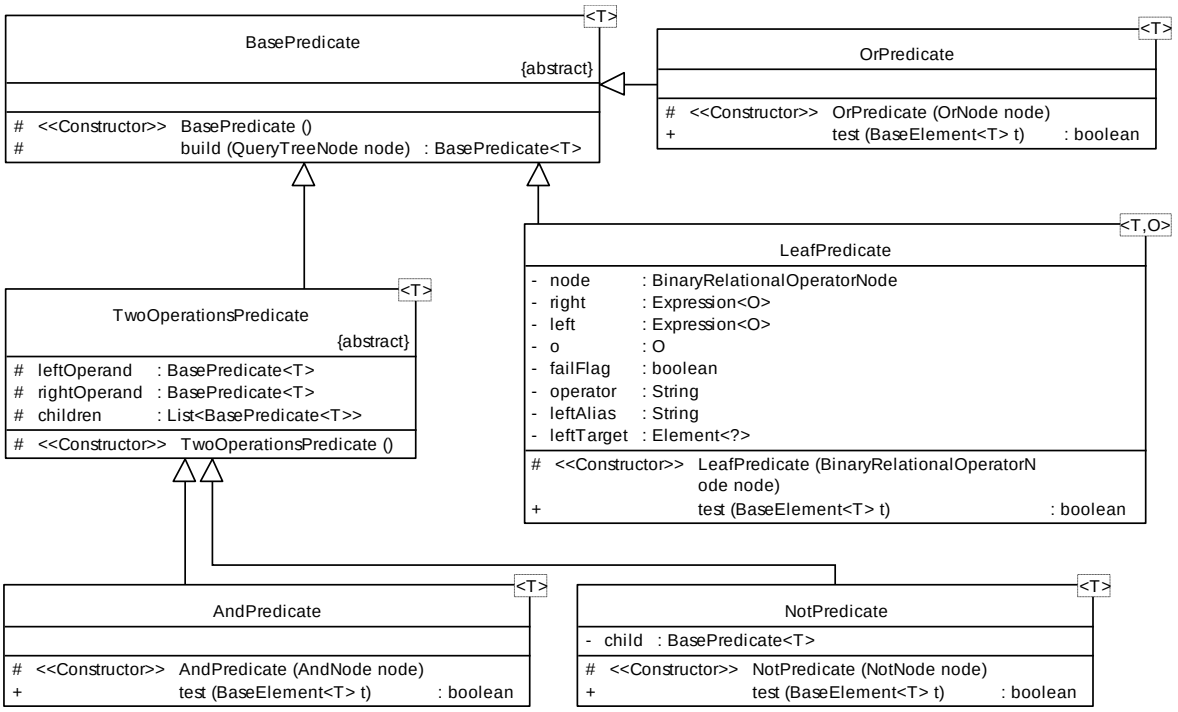
Ve své knihovně jsem se rozhodl využít návrhový vzor interpret několikrát, jelikož SQL dotaz, který je reprezentován AST stromem, lze rozdělit na jednotlivé uzle, které jdou interpretovat samostatně při dodržení správného pořadí interpretace.

Jelikož moje knihovna využívá 4 částí dotazu SELECT, tak využívám 3 interpretů a jedné funkce v tomto pořadí, interpret klauzule FROM, interpret klauzule WHERE, interpret klauzule ORDER BY a funkce SELECT(`SelectFunction`). Prvním interpretem je interpet konstrukcí JOIN (Obrázek 9), jedná se o start celého procesu, kdy spojuji jednotlivé prvky spolu. V tomto případě se kontext předává již v konstruktoru, protože se jedná o vstupy v rozhranní `Iterable`, které zůstávají stejné během celého procesu.

Dále se v konstruktoru předává SQL dotaz v AST, využívá se toho faktu, že jeho stromová struktura odpovídá struktuře, která se má vytvořit z tříd našeho interpretu. Druhým interpretem je interpret podmínek z klauzule WHERE (Obrázek 10). Tento interpret využívá dvou interpretů zároveň, první interpretuje hierarchii dotazu, konkrétně logické spojky AND a OR a také negaci NOT. Všechny tyto tři výrazy jsou neterminální výrazy a implementují funkční interfejs `Predicate`, jehož metoda test vrací logickou hodnotu.

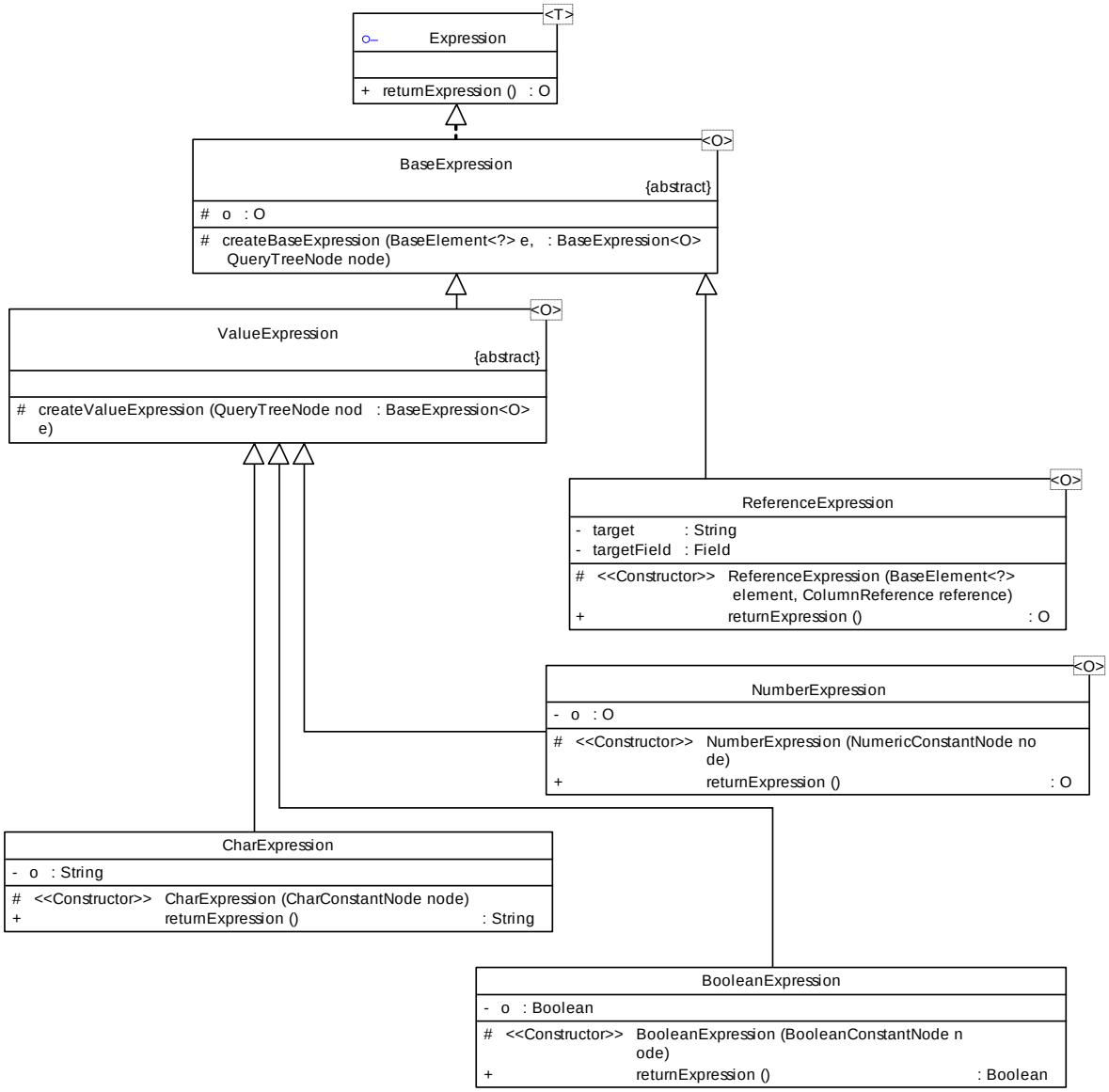


Obrázek 9: Class diagram Join interpreteru Zdroj: autor



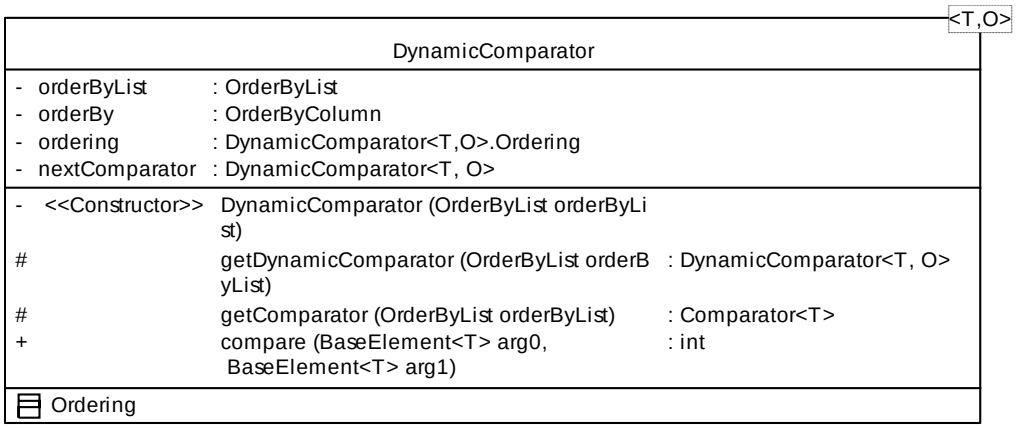
Obrázek 10: Class diagram Predicate interpreteru Zdroj: autor

Druhým interpretem je interpret podmínek, podmínku lze rozdělit levý a pravý operandy a operátor uprostřed. V rámci operátoru knihovna zvládá běžné operátory včetně *nerovná se*. Na vrcholu stojí interfejs `Expression` (Obrázek 11), která má navíc statickou metodu `getResult(String operator, Integer result)`, která na základě operátoru a výsledku metody `compareTo(T o)` vrací logickou hodnotu. Pod ní stojí abstraktní třída `BaseExpression` a ta funguje jako společný předek, buď pro hodnotu získanou jako referenci z dat nebo jako hodnotu získanou z SQL dotazu.



Obrázek 11: Class diagram Expression interpretu Zdroj: autor

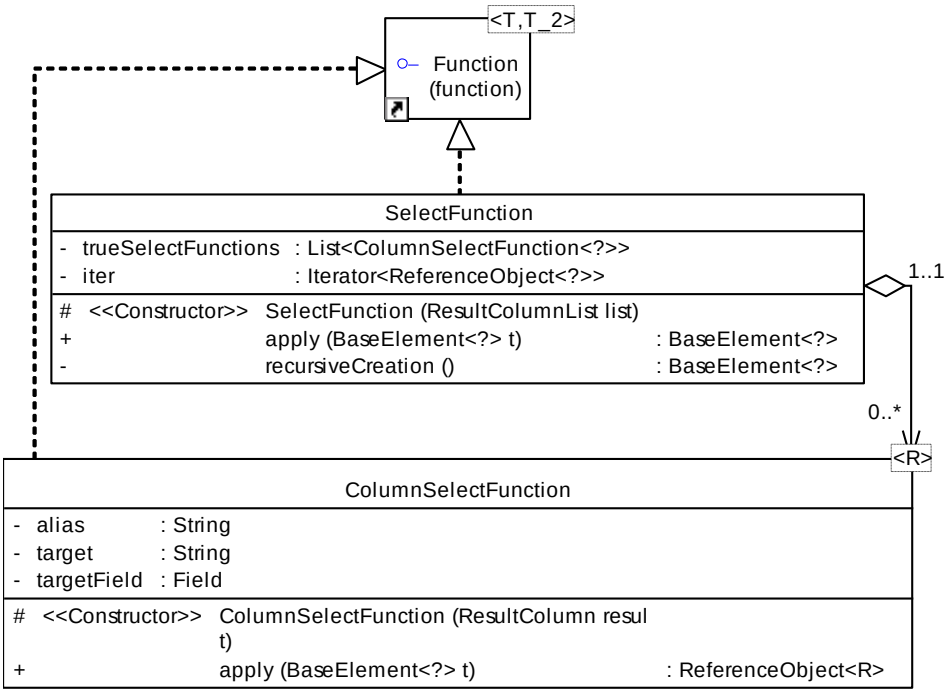
Interpret klauzule ORDER BY (Obrázek 12) implementuje interfejs `Comparator`. Interpret je předán datovodu a podle něj je tříděn. Vnitřní implementace komparátoru je implementována jako spojový seznam, kdy obsahuje referenci na další komparátor a při konstrukci se tato struktura vytváří podle předaného uzle AST ve formě `OrderByList` (seznam kritérií podle čeho třídit). Při interpretaci, pokud je výsledkem nula, tak se komparátor pokusí interpretovat následný komparátor a tento výsledek případně upraví podle směru třídění.



Obrázek 12: Class diagram Order By interpretu Zdroj: autor

6.5 Funkce Select

Požadavek restriktivní výběr atributů nebo entity naplňuji skrze třídu `SelectFunction` (Obrázek 13), jež agreguje řadu menších funkcí a spojuje jejich výsledky. V konstruktoru třídy `SelectFunction` je parametr `ResulList [39]`, jež je uzel AST. Dle jednotlivých prvků toho listu jsou vytvořeny funkce, které pak následně v aplikování vrací vybírané atributy nebo entity.



Obrázek 13: Class diagram Select funkce Zdroj: autor

Posledním krokem metody `apply(BaseElement<?> t)` je vytvoření nového řetězce `BaseElement` prvků, jelikož jsou mezivýsledky uloženy v seznamu, tak je možné pomocí rekurzivního volání vytvořit celý řetězec.

Výpis 30: *Implementace rekurzivního vytváření BaseElementu pro výstup z knihovny*

```
private BaseElement<?> recursiveCreation()
{
    ReferenceObject<?> local = iter.next();
    if (iter.hasNext()) {
       return CompositeBaseElement.createCompositeBaseElement(local.getR(),
                local.getAlias(), recursiveCreation());
    }

    return TerminateBaseElement.createTerminateBaseElement(local.getR(),
                local.getAlias());
    }
}
```

7 Příručka

SQL2Stream je knihovna podporující zpracování dat pomocí SQL dotazů pro Javu. Snaží se na základě deklarativního vyjadřování skrze jazyk SQL odstínit uživatele od implementace třídění a zpracování dat. Knihovna využívá nové prvky z Javy 8, jako jsou datovody nebo funkční rozhraní. Jelikož se SQL jazyk zaměřuje na dotazování do relačních databází, tak pro účely této knihovny byl zjednodušen, aby se dal využít pro práci s objekty a třídami v Javě.

Ve zdrojových kódech aplikace jsou dostupné testy, jež ukazují využití knihovny a způsob, jakým se knihovna testovala.

Podpora jazyka SQL knihovnou je omezená, ale pro většinu činností a požadavků by měla postačovat. Knihovna nepodporuje funkce, vnořené dotazy a seskupování.

V klauzuli SELECT máme dvě možnosti: „*“ nebo konkrétní atributy/objekty. SQL dotaz vypadá následovně:

Výpis 31: *Ukázka použití knihovny při výběru celého data setu*

```
String sql = "SELECT * FROM list1 t";
SQL2Stream<TestingModel> test = new SQL2Stream<>(sql,
            new InputContainer<>(list1, TestingModel.class));
ResultSupplier<BaseElement<TestingModel>> result = test.interpret();
```

Výpis 32: *Ukázka použití knihovny při výběru konkrétního atributu entity z výsledného data setu*

```
String sql = "SELECT t.name FROM list1 t WHERE t.number>2";
SQL2Stream<TestingModel> test = new SQL2Stream<>(sql,
            new InputContainer<>(list1, TestingModel.class));
ResultSupplier<BaseElement<String>> result = test.interpret();
```

Klauzule FROM podporuje jednoduché konstrukce JOIN, kdy jednotlivé kolekce/zdroje jsou povinně označeny aliasy, které odpovídá pořadí jejich vstupů v parametrech konstruktorů. Knihovna omezuje počet konstrukcí JOIN na tři, takže jsou 4 aliasy *t,u,r* a *v*.

Nejdůležitější a nejvíce využívaná je klauzule WHERE, kde je možno spojovat kondicionály logickými spojkami AND a OR a negací NOT. Možné operátory naleznete níže (Tabulka 3).

Tabulka 3: Výčet možných operátorů

Operátory	
=	Rovná se
<>	Nerovná se
>	Větší než
>=	Větší nebo rovno
<	Menši než
<=	Menší nebo rovno

Výpis 33: Ukázka možného SQL dotazu se složitější WHERE klauzulí

```
String sql = "SELECT * FROM list WHERE number='1' AND name='test' OR
        name='test2'";
```

Klauzule WHERE respektuje vztahy a hierarchii mezi kondicionály, takže lze využít závorek. Uvnitř knihovny je tato klauzule reprezentována stromem, ve kterém jednotlivé uzly vyjadřují logické spojky a listy stromu jsou konkrétní kondicionály.

Poslední podporovanou klauzulí je ORDER BY, která slouží pro setřídění množiny dat. Lze třídit dle jednoho atributu dané třídy v obou směrech (sestupně, vzestupně-defaultní hodnota).

Celkový podporovaný syntax jazyka SQL knihovnou vypadá následovně:

Výpis 34: Celkový syntax SQL dotazu podporovaný knihovnou

```
SELECT [*|<atribut>]*?
FROM [<název_kolekce>|<nazáv_třídy> <alias>] [JOIN <název_kolekce>|<nazáv_třídy>
<alias> ON <alias>.<atritbut><operátor><alias><atribut>] *? (max 3)
WHERE <podmínka> [(AND|OR) <podmínka>]*?
ORDER BY <atribut> [ASC|DESC]*?
```

7.1 Práce s výstupem

Metoda interpret vrací instanci třídy, která implementuje interfejs `ResultIterator`, která funguje jako generátor. Tato instance vrací buď instance interfejsu `BaseElement<T>` nebo instance typu T.

Výpis 35: Rozhraní ResultIterator

```
public interface ResultIterator<T> extends Iterator<T>
{
    @Override
    public T next();
    @Override
    public boolean hasNext();
}
```

Aby knihovna vracela instance typu T, je potřeba metodě interpret předa mapovací funkci v parametru.

Výpis 36: Deklarace mapovací funkce

```
Function<List<Object>, T> mappingFunction
```

Uživatel ví, že jednotlivé prvky listu odpovídají vybraným atributům nebo prvkům z klauzule SELECT a mají téže pořadí. Seznam může mít i jen jeden prvek. Uživatel může zvolit sám implementaci a je na něm jestli použije jednoduché přetypování nebo využije nějaké komplikovanější postupy jako je reflexe. Jednoduchý příklad implementace:

Výpis 37: Ukázka jednoduché implementace mapovací funkce

```
public class MappingFunction implements Function<List<Object>, TestingModel>
{
    @Override
    public TestingModel apply(List<Object> t)
    {
        TestingModel local = new TestingModel();
        local.setName((String) t.get(0));
        return local;
    }
}
```

8 Závěr

Pokud porovnám specifikace zadání s výslednou knihovnou, tak z pohledu funkcionality jsem naplnil svůj cíl. Knihovna umí třídit, filtrovat a zpracovávat data podle zadaných SQL dotazů. Z pohledu vstupů a výstupů jsem cíl zadání knihovny zcela nenaplnil, jelikož vstupem do knihovny je pouze interfejs `Iterable`. Toto omezení vzniklo na základě technických příčin. Dle mého názoru výstup knihovny odpovídá danému zadání, ale v tomto případě záleží individuálně na uživateli knihovny, zda mu tento výstup vyhovuje, či nikoliv.

Vzhledem k rozsahu jazyka SQL a jeho funkcionalit bylo těžké pevně definovat jeho podobu, ve které se bude používat v této knihovně. Určitě jsem naplnil základní očekávání a požadavky, ale vypustil jsem dvě klauzule SELECT dotazu, jež přinášejí možnost agregace dat. Pro implementaci klauzule GROUP BY by bylo potřeba též implementovat i funkce (minimálně agregační). Poslední část, která je zcela vynechána, jsou vnořené dotazy. Proto možnosti pro vylepšení knihovny a rozšíření funkcionalit velmi rozsáhlé.

Dle mého názoru jsem naplnil cíl z pohledu využití funkcionálního programování a novinek v Javě 8, k němu určených. Moje knihovna nabízí uživatelům alternativu v podobě deklarativního zpracování dat za využití SQL dotazů na platformě Java. Z mého pohledu Java ještě nenarazila na strop, jenž by ukázal na její maxima v oblasti funkcionálního programování a trend implementování funkcionálních prvků do Javy bude dále pokračovat.

Terminologický slovník

Termín	Zkratka	Význam [zdroj]
Structured Query Language	SQL	Standardizovaný dotazovací jazyk používaný pro práci s daty v relačních databázích a jejich ovládání. [40]
Refaktorace		Kontrolovaný proces zvyšování kvality existujícího kódu, za použití menších transformací, které zachovávají chování a funkcionalitu. [41]
Objektově-relační mapování	ORM	Objektově-relační mapování je programovací technika, jež konvertuje data nekompatibilní typy systémů v objektovém programování. [42]
Abstraktní syntaktický strom	AST	Abstraktní syntaktický strom je stromová reprezentace abstraktní syntaktické struktury zdrojového kódu. [43]
Application Programming Interface	API	Množina procedur, funkcí a objektů, které aplikace poskytuje uživatelům k používání. Tato množina je nezávislá na implementaci programu nebo knihovny. [44]
Parser		Též syntaktický analyzátor. Parser je program jež zpravidla vstupní text transformuje na určitou datovou strukturu, většinou syntaktický strom. [45]
Datovod		Sekvence elementů ze zdroje, jež poskytuje možnost zpracování. Datovody nabízejí možnost interní iterace a paralelní zpracování [46]
Data set		Data set je kolekce dat, jež odpovídá obsahu jedné tabulky v relační databázi.
Database-first		Programovací technika, jež nabádá vývojáře, aby navrhoval a implementoval perzistentní vrstvu jako první a v aplikační vrstvě se o ní opíral. [47]
Java Database Connectivity	JDBC	API ze standardní knihovny Javy, jež definuje jednotné rozhraní pro přístup do relačních databází. [48]
Boilerplate code		Opakující část kódu, která má minimální vliv na konkrétní požadavek či implementaci, ale i přesto musí být implementována. Též syntaktický cukr. [49]
Not-only SQL	NoSQL	Databázový koncept, jehož perzistentní vrstva je reprezentována i jinými než relačními databázemi. Tyto databáze jsou zaměřeny na škálovatelnost a jsou resilientní vůči chybám [50]

Seznam literatury

1. **Tulach, Jaroslav.** *Practical API design: confessions of a Java framework architect.* Berkeley, CA : Apress, 2008. str. 43. ISBN 1430209739.

2. **Jim, Melton a Simon, Alan R.** *Understanding the new SQL: a complete guide.* San Francisco, CA : Morgan Kaufmann Publishers Inc, 1993. ISBN 1-55860-245-3 .

3. **Urma, Raoul-Gabriel, Fusco, Mario a Mycroft, Alan.** *Java 8 in action: lambdas, streams, and functional-style programming.* Shelter Islan : Manning, 2015. ISBN 9781617291999.

4. **Warburton, Richard.** *Java 8 Lambdas.* Beijing : O'Reilly, 2014. ISBN 9781449370770.

5. **Šimůnek, Milan.** *SQL: kompletní kapesní průvodce.* Praha : Grada, 1999. ISBN 8071696927 9788071696926.

6. **Tulach, Jaroslav.** *Practical API design: confessions of a Java framework architect.* Berkeley, CA : Apress, 2008. ISBN 9781430209737.

7. **Tahchiev, Petar a Massol, Vincent.** *JUnit in Action.* Greenwich : Manning, 2011. ISBN 9781935182023 1935182021.

8. **Urma, Raoul-Gabriel, Fusco, Mario a Mycroft, Alan.** *Java 8 in action: lambdas, streams, and functional-style programming.* Shelter Island : Manning, 2014. stránky 119-120. ISBN 9781617291999.

9. **Geekery, Data.** Interface Record. *jOOQ 3.2.6 API.* [Online] 2014. [Citace: 20. 3 2015.] http://www.jooq.org/javadoc/3.2.x/org/jooq/Record.html.

10. **Eder, Lukas a conference, GeeCon.** GeeCON 2014: Lukas Eder - Get Back in Control of Your SQL. [Online] 2014. [Citace: 20. 3 2015.] https://vimeo.com/99526433.

11. **Šimůnek, Milan.** *SQL: kompletní kapesní průvodce.* Praha : Grada, 1999. stránky 33-41. ISBN 8071696927 9788071696926.

12. **Corporation, Oracle.** Valhalla. *OpenJDK.* [Online] 2015. [Citace: 20. 4 2015.] http://openjdk.java.net/projects/valhalla/.

13. **Fontaine, Dimitri.** Understanding Window Functions. *Expertise PostgreSQL.* [Online] 2013. [Citace: 3. 5 2015.] http://tapoueh.org/blog/2013/08/20-Window-Functions.

14. **Microsoft.** LINQ to DataSet. *Microsoft Developer Network.* [Online] 2011. [Citace: 27. 4 2015.] https://msdn.microsoft.com/en-us/library/bb386977%28v=vs.110%29.aspx.

15. **Hyde, Julian.** Saffron overview. *Sourceforge.* [Online] 2001. [Citace: 3. 5 2015.] http://saffron.sourceforge.net/overview.html.

16. —. linq4j 0.3 API. *Interface Enumerator<T>.* [Online] 2012. [Citace: 3. 5 2015.] http://www.hydromatic.net/linq4j/apidocs/net/hydromatic/linq4j/Enumerator.html.

17. **Luontola, Esko.** Retro Lambda. *GitHub.* [Online] 2013. [Citace: 5. 5 2015.] https://github.com/orfjackal/retrolambda.

18. **Dietrich, Daniel.** Documentation. *Javaslang.* [Online] 2014. [Citace: 20. 3 2015.] htttp://www.javaslang.com.

19. **Olšák, Petr.** *https://vimeo.com/99526433.* Praha : FEL ČVUT v Praze, 2007. str. 22. ISBN 9788001037751 8001037754.

20. **Weisstein, Eric W.** Monoid. *Wolfram Web Resource.* [Online] 2000. [Citace: 3. 5 2015.] http://mathworld.wolfram.com/Monoid.html.

21. **Javi, Roman.** The Hadoop Ecosystem Table. [Online] 2013. [Citace: 5. 5 2015.] https://hadoopecosystemtable.github.io/.

22. **Langit, Lynn.** Hadoop MapReduce Fundamentals. [Online] 2013. [Citace: 3. 5 2015.] http://www.slideshare.net/lynnlangit/hadoop-mapreduce-fundamentals-21427224.

23. **Foundation, The Apache Software.** *Web Apache Spark.* [Online] 2014. [Citace: 3. 5 2015.] https://spark.apache.org/.

24. **Urma, Raoul-Gabriel, Fusco, Mario a Mycroft, Alan.** *Java 8 in action: lambdas, streams, and functional-style programming.* Shelter Island : Manning, 2014. stránky 100-118. ISBN 9781617291999.

25. **Red Hat Middleware, LLC.** Chapter 19. Improving performance. *HIBERNATE - Relational Persistence for Idiomatic Java.* [Online] 2004. [Citace: 20. 4 2015.] https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/performance.html#performance-fetching.

26. **Šimůnek, Milan.** *SQL: kompletní kapesní průvodce.* Praha : Grada, 1999. stránky 128-129. ISBN 8071696927 9788071696926.

27. —. *SQL: kompletní kapesní průvodce.* Praha : Grada, 1999. stránky 140-141. ISBN 8071696927 9788071696926.

28. —. *SQL: kompletní kapesní průvodce.* Praha : Grada, 1999. stránky 212-217. ISBN 8071696927 9788071696926.

29. —. *SQL: kompletní kapesní průvodce.* Praha : Grada, 1999. stránky 154-155. ISBN 8071696927 9788071696926.

30. **Oracle.** Class StreamSupport. *Java™ Platform, Standard Edition 8, API Specification.* [Online] 2015. [Citace: 3. 5 2015.] https://docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html.

31. **Robertson, Ian.** Code by Any Other Name - Reflecting generics. *Artima Developer.* [Online] 2007. [Citace: 20. 3 2015.] http://www.artima.com/weblogs/viewpost.jsp?thread=208860.

32. **FoundationDB.** GitHub. *sql-parser.* [Online] 2015. [Citace: 5. 5 2015.] https://github.com/rimig/sql-parser.

33. **Šimůnek, Milan.** *SQL: kompletní kapesní průvodce.* Praha : Grada, 1999. str. 128. ISBN 8071696927 9788071696926.

34. **Tulach, Jaroslav.** *Practical API design: confessions of a Java framework architect.* Berkeley, CA : Apress, 2008. str. 37. ISBN 9781430209737.

35. —. *Practical API design: confessions of a Java framework architect.* Berkeley, CA : Apress, 2008. str. 38. ISBN 9781430209737.

36. —. *Practical API design: confessions of a Java framework architect.* Berkeley, CA : Apress, 2008. str. 55. ISBN 9781430209737.

37. **Gamma, Erich.** Reading, Mass : Addison-Wesley, 1995. stránky 183-217. ISBN 0201633612.

38. —. *Design patterns: elements of reusable object-oriented software.* Reading, Mass : Addison-Wesley, 1995. stránky 274-288. ISBN 0201633612.

40. **Šimůnek, Milan.** *SQL: kompletní kapesní průvodce.* Praha : Grada, 1999. stránky 28-29. ISBN 8071696927 9788071696926.

41. **Fowler, Martin.** Refactoring - Improving the Design of Existing Code. *Martin Fowler Web.* [Online] 2012. [Citace: 5. 5 2015.] http://martinfowler.com/books/refactoring.html.

42. **Wikipedia.** Object-relational mapping. *Wikipedia, the free encyclopedia.* [Online] 2015. [Citace: 5. 5 2015.] http://en.wikipedia.org/wiki/Object-relational_mapping.

43. —. Abstract syntax tree. *Wikipedia, the free encyclopedia.* [Online] 2015. [Citace: 5. 5 2015.] Abstract syntax tree.

44. —. Application programming interface. *Wikipedia, the free encyclopedia.* [Online] 2015. [Citace: 5. 5 2015.] http://en.wikipedia.org/wiki/Application_programming_interface.

45. —. Parser. *Wikipedia, the free encyclopedia.* [Online] 2015. [Citace: 5. 5 2015.] http://en.wikipedia.org/wiki/Parsing.

46. **Urma, Raoul-Gabriel, Fusco, Mario a Mycroft, Alan.** *Java 8 in action: lambdas, streams, and functional-style programming.* Shelter Island : Manning, 2015. stránky 100-106. ISBN 9781617291999.

47. **Devart.** Database-First Approach. *Devart.* [Online] 2015. [Citace: 5. 5 2015.] http://www.devart.com/entitydeveloper/database-first.html.

48. **Wikipedia.** Java Database Connectivity. *Wikipedia, the free encyclopedia.* [Online] 2015. [Citace: 5. 5 2015.] http://en.wikipedia.org/wiki/Java_Database_Connectivity.

49. —. Boilerplate code. *Wikipedia, the free encyclopedia.* [Online] 2015. [Citace: 5. 5 2015.] http://en.wikipedia.org/wiki/Boilerplate_code.

50. —. NoSQL. *Wikipedia, the free encyclopedia.* [Online] 2015. [Citace: 5. 5 2015.] http://en.wikipedia.org/wiki/NoSQL.

Seznam obrázků a tabulek

Základní text

Seznam obrázků

Seznam tabulek

Obsah přiloženého CD

- Vlastní text práce
- Zdrojové kódy knihovny spolu s testy
- Class diagramy v EMF formátu
- Zkompilovaná knihovna