FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Návrh a implementace konektoru cloudového nástroje do systému manta
Student:	Michal Podrábský
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Cílem práce je provést analýzu cloudových nástrojů Looker a FiveTran, porovnat jejich možnosti z pohledu transformace dat a poté pro jeden z nich navrhnout a implementovat funkční prototyp modulu, který provede syntaktickou a sémantickou analýzu transformací a následně její výsledek využije pro analýzu datových toků. Modul bude zařazen do projektu Manta.

Dílčí cíle bakalářské práce:
1. Seznamte se s projektem Manta a s nástroji Looker a FiveTran.
2. Navrhněte modul v projektu Manta pro analýzu datových toků v jednom z těchto nástrojů. Využijte existující infrastrukturu projektu.
3. Implementujte prototyp, řádně ho zdokumentujte a otestujte.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 15. února 2022 v Praze.

Bakalářská práce

NÁVRH A IMPLEMENTACE KONEKTORU CLOUDOVÉHO NÁSTROJE DO SYSTÉMU MANTA

Michal Podrábský

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Michal Valenta, Ph.D.
11. května 2022

Obsah

Seznam obrázků

Seznam výpisů kódu

Rád bych využil této příležitosti k poděkování mému vedoucímu Ing. Michalu Valentovi Ph.D. za cenné rady a Ing. Petru Košvancovi za neocenitelnou pomoc a vedení při práci v Mantě. Také bych chtěl poděkovat mým rodičům a sestře za podporu, a to nejen v dobách, kdy jsem pracoval na této práci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů (dále jen „autorský zákon"), především § 35 a § 60 autorského zákona upravující školní dílo.

V případě počítačových programů, jež jsou součástí mojí práce či její přílohou, a veškeré související dokumentace k počítačovým programům (dále jen „software"), uděluji v souladu s ust. § 2373 zákona 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, nevýhradní a neodvolatelné oprávnění (licenci) k užití software, a to všem osobám, které si přejí software užít. Tyto osoby jsou oprávněny software užít jakýmkoli způsobem a za jakýmkoli účelem v neomezeném rozsahu (včetně užití k výdělečným účelům), vč. možnosti software upravit či měnit, spojit jej s jiným dílem a/nebo zařadit jej do díla souborného. Toto oprávnění je časově, teritoriálně i množstevně neomezené a uděluji jej bezúplatně.

V Praze dne 11. května 2022 .....

Abstrakt

Bakalářská práce se zaměřuje na implementaci prototypu skeneru, který je začleněn do softwaru Manta. Prototyp skeneru se specializuje na analýzu transformací dat uložených v databázích, které s nimi nástroj Fivetran vykonává. K tomu prototyp využívá metadata poskytnutá nástrojem Fivetran. Skener metadata zpracovává a načítá do interního datového modelu, který následně využije pro sestavení grafu reprezentujícího tok dat skrze nástroj Fivetran. Graf je využit softwarem Manta, který z něj vytvoří vizualizaci datového toku. Součástí práce je i analýza nástrojů Fivetran a Looker, ve které jsou popsány veškeré transformace, které s daty nástroje vykonávají.

Klíčová slova datová linie, datové toky, metadata, BI nástroje, ETL nástroje, Looker, Fivetran, DBT, Manta

Abstract

The work focuses on the implementation of a prototype scanner for the Manta tool. The prototype specializes in the analysis of database transformations performed by the Fivetran tool, in order to visualize them. The scanner achieves this by analyzing the metadata which loads into an internal data model. It then processes the data model and uses it to build a graph describing the data transformations. The Manta tool uses this graph for their visualization. The work also includes an analysis of the Fivetran and Looker tools describing all the transformations they perform with the data.

Keywords data lineage, data flow, metadata, BI tools, ETL tools, Looker, Fivetran, DBT, Manta

Seznam zkratek

BI	Business Intelligence
dbt	data built tool
ETL	Extract, Transform, Load
SQL	Structured Query Language
DRY	Don't repeat yourself
JSON	JavaScript Object Notation

Kapitola 1
Úvod

V úvodní kapitole vás seznámím s obsahem, užitkem a cíli bakalářské práce.

V dnešní době společnosti sbírají a zpracovávají velké množství dat. Například data o svých ziscích, zaměstnancích, zákaznících nebo o produktech, které prodávají. Takové informace jsou velmi důležité pro správný rozvoj těchto společností. Během procesu analyzování sesbíraných informací s daty pracuje velké množství specializovaných nástrojů, jako jsou integrační nástroje, které slouží pro formátování a přesun dat do centralizovaného úložiště nebo analytické nástroje specializující se na vytváření informačních zpráv. Tyto nástroje jsou často používány sériově a mohou tvořit velmi komplikovaný systém. Pokud by se společnost rozhodla odebrat z tohoto systému jeden z nástrojů, musela by zjistit, jaké nástroje v systému používají výstupy odebíraného nástroje. To zahrnuje kontrolu všech nástrojů v systému. Což je velmi obtížný proces jak časově, tak i personálně. Značně ho však mohou ulehčit nástroje pro analýzu datových toků.

Jedním takovým nástrojem je Manta Flow. Manta se umí připojit k nástroji pro správu dat, zpracovat jeho metadata a vytvořit vizualizaci pohybu dat skrze tento nástroj. Toho Manta docílí pomocí takzvaných skenerů. Každý skener se specializuje na jeden nástroj, pro který zpracuje a načte metadata do vhodné datové struktury, z níž následně vytvoří graf datového toku. V mé práci se zaměřím na analýzu nástrojů Fivetran a Looker, na návrh a implementaci prototypu pro jeden z nich. Fivetran se specializuje na integraci dat z různých datových zdrojů do jednoho centralizovaného úložiště a na jejich přípravu pro budoucí zpracování. Looker je nástroj pro vytváření komplexních zpráv složených z grafů a tabulek. Před implementací skeneru je třeba analyzovat možnosti transformací obou nástrojů. Výsledek analýz využiji nejen při návrhu skeneru, ale i při výběru nástroje. Návrh bude rozšiřitelný a implementace řádně otestována.

1.1 Cíle práce

Cílem práce je implementace prototypu skeneru pro jeden z cloudových nástrojů Looker nebo Fivetran. Implementaci bude předcházet analýza jednotlivých nástrojů zahrnující popis jejich možností pro transformace dat. Jeden z nástrojů si vyberu a popíšu obsah a strukturu metadat, které poskytuje. Následně navrhnu a implementuji prototyp pro tento nástroj.

Jedním z úkolů prototypu bude zpracování a nahrání struktury s metadaty do vhodného datového modelu. Návrh struktury adresáře s metadaty a datového modelu je jedním z cílů mé práce. Extrakce metadat z nástroje není součástí této práce, skener bude pouze analyzovat manuálně extrahovaná data z nástroje. Prototyp se zaměří na vytvoření vhodného datového modelu popisujícího datový tok pro databáze. Ostatní typy zdrojů nebudou do implementace zahrnuty. Druhým úkolem skeneru bude použít datový model k vytvoření speciální datové struktury, která bude využita k vizualizaci datových toků. Prototyp musí být řádně otestován a integrován do nástroje Manta. Prototyp rozšíří skupinu podporovaných nástrojů, pro které je Manta schopná vytvářet vizualizace datových toků.

Kapitola 2

Základní pojmy a technologie

V této kapitole představím pojmy používané v mé bakalářské práci a technologie použité při implementaci prototypu.

2.1 Datové toky a Rodokmen dat

Datový tok můžeme chápat jako „*pohyb dat systémem softwaru, hardwaru nebo kombinaci obou*“[1]. Datový rodokmen (anglicky „Data lineage“) je proces identifikování, analyzování a vizualizování datových toků od zdroje až ke koncovému uživateli. Datový rodokmen si můžeme představit jako GPS pro data, jelikož nám poskytuje přehled o tom, jak se data zpracovávají, transformují a přenášejí od zdroje ke koncovému uživateli. [2]

2.2 ETL nástroje

Jako ETL jsou označovány nástroje, jejichž úkolem je transformovat a transportovat data z různých datových zdrojů do centralizovaných datových úložišť. Toho je docíleno třífázovým integračním procesem extrakce, transformace a načtení dat, (anglicky „Extract, Transform, Load“), ze kterého je zkratka ETL odvozena. [3]

První část procesu extrakce je zodpovědná za načtení dat z datových úložišť, jako jsou databáze, soubory nebo také aplikace, do takzvané pracovní oblasti. Po nahrání dat do pracovní oblasti přichází na řadu fáze transformací. V této fázi procházejí data úpravami, které mají za úkol připravit data k budoucímu použití a na nahrání do cílového úložiště. Úpravami je myšleno filtrování a odstraňování duplicitních dat nebo formátování dat do struktury schémat, tabulek a sloupců. V poslední fázi jsou data načtena do cílového úložiště. Celý tento proces může být periodicky opakován, aby byla data v cílovém úložišti co nejaktuálnější. [3]

2.3 Business intelligence nástroje

Business intelligence nástroje, zkráceně BI nástroje, slouží k zpracování velkého množství dat, jejich analyzování, transformování a vytváření vizualizací, jako jsou grafy, tabulky nebo přehledové nástěnky. Vizualizace mohou být založeny jak na aktuálních, tak na historických datech. Narozdíl od ETL, BI nástroje ve většině případů čerpají data pouze z jednoho centralizovaného úložiště. Nástroje pomáhají s monitorováním výnosů, ztrát, efektivity interních procesů nebo správnosti obchodních hodnocení. Na základě těchto informací mohou společnosti vykonávat optimálnější a rychlejší obchodní rozhodnutí [4], [5].

2.4 Metadata

Metadata jsou často označována jako data, která uchovávají informace o jiných datech [6]. Pojem metadata je často definován a užíván v různých profesích jiným způsobem. V mé bakalářské práci jsou metadata chápána jako data, která popisují strukturu a operace, které jsou s daty prováděny [7]. Existuje mnoho případů, kdy jsou metadata velmi užitečná. Využijeme je například, pokud potřebujeme zjistit, jakým způsobem a v jaké struktuře jsou data uložena v datových úložištích. Datovými úložišti jsou myšleny technologie na ukládání dat, jako jsou například databáze. Velmi důležitým zdrojem informací se metadata stávají také v případě, kdy potřebujeme zjistit, jak s daty pracují technologie jako jsou ETL a BI nástroje. Tyto nástroje často poskytují vlastní sady metadat popisující modifikace, kopírování nebo přesuny, které s daty nástroje uskutečňují [8].

2.5 Fivetran

Fivetran je americká společnost sídlící v Ouklandu v Kalifornii. Byla založená v roce 2012. Společnost vyvíjí stejnojmenný cloudový nástroj patřící do rodiny ETL nástrojů [9]. Nejedná se ale o typický ETL nástroj, liší se pořadím fází. Fáze transformace a načtení, (anglicky „Transform, Load"), jsou prohozeny. Fáze jsou tedy prováděny v pořadí extrakce, načtení a transformace. Z tohoto důvodu je Fivetran často označován jako ELT nástroj. Zajímavostí je také, že společnost Fivetran se rozhodla integrovat jádro nástroje dbt, které využívá ve fázi transformací dat [10].

2.6 dbt

Dbt je zkratka pro anglické „data built tool". Tento otevřený software je vyvíjen společností Fishtown Analytics, která se v roce 2020 přejmenovala na dbt Lab [11]. Dbt je používán k transformacím dat v datových úložištích. Některé společnosti, jako je například Fivetran, integrují dbt do svých nástrojů. Z tohoto důvodu je dbt často označován jako T v ETL nástrojích. V jádru slouží dbt ke kompilování a spouštění kódu složeného z databázového jazyka SQL a šablonovacího jazyka Jinja [12]. Další informace o nástroji dbt lze nalézt v kapitole 3.

2.7 Looker

Looker Data Sciences, Inc. je americká společnost sídlící v Santa Cruz v Kalifornii. Byla založena v roce 2012. Společnost vyvíjí cloudový business intelligence nástroj s názvem Looker. Společnost byla v roce 2019 zakoupena společností Google a nyní je nástroj Looker součástí platformy Google Cloud Platform [13]. Looker používá jazyk LookML k definování datových modelů, dle kterých vygeneruje SQL dotazy. Tyto dotazy jsou použity k extrakci dat z datového úložiště. Data jsou následně uložena do stejného datového modelu, podle kterého byly vygenerovány SQL dotazy. Tyto modely jsou využívány při vytváření vizualizací [14].

2.8 Manta

Společnost Manta, celým názvem Manta Tools, s.r.o., „*je česko-americká společnost, vyvíjející nástroj pro analýzu datových toků.*" [15]. Manta začala jako projekt konzultační společnosti Profinit mezi roky 2008 a 2009. V roce 2013 byla založena společnost Manta Tools, s.r.o, která projekt převzala [15].

2.8.1 Manta Flow

Hlavním produktem je nástroj sloužící k analýze a vizualizaci datových toků zvaný Manta Flow. Jeho součástí jsou takzvané skenery. Každý skener se specializuje na zpracování jedné technologie účastnící se datového toku. Skener zpracuje metadata a rekonstruuje část datového toku pro danou technologii, za kterou je zodpovědný. Nástroj Manta Flow využívá těchto skenerů k vytvoření grafů datových toků, jenž jsou pak jiným modulem platformy vizualizovány [16]. Manta podporuje mnoho technologií, jako jsou například: [17]

- Databáze: PostgreSQL, IBM DB2, Google Big Query
- Programovací jazyky: C#, Cobol, Java
- BI nástroje: OBIEE, Power BI
- ETL nástroje: Talend, IBM InfoSphere DataStage
- Nástroje na modelování dat: erwin Data Modeler, ER/Studio

2.8.2 Graf datových toků

Výsledkem analýzy skenerů je orientovaný graf popisující daný datový tok. Manta obohacuje vrcholy a hrany grafu o jejich typ. Typy vrcholů a hran ovlivňují, jestli jsou vrcholy zobrazeny ve finální vizualizaci a jakým způsobem. Graf má velké množství těchto typů, avšak v mé práci využiji pouze jejich zlomek [18].

Typy vrcholů: [18], [19]

- **Uzel** (v anglickém originále „Node“) je vrchol reprezentující objekt participující ve finální vizualizaci. Tento uzel má parametr „nodeType“, který určuje jeho podtyp, jako je například schéma, tabulka nebo také SQL skript, soubor či adresář.
- **Zdroj** (v anglickém originále „Resource“) reprezentuje zdroj metadat. Zdrojem je myšlena konkrétní technologie, jako je například OBIEE nebo PostgreSQL.
- **Atribut** (v anglickém originále „Attribute“) je vrchol obsahující přídavné informace o vrcholu typu Uzel.

Typy hran: [18], [19]

- **HasResource** je hrana přiřazující vrcholům typu Uzel vrchol Resource. To znamená, že tato hrana přiřazuje každému Uzlu technologii, ke které patří.
- **HasParent** je hrana, která spojuje syna a rodiče. Tuto vazbu bychom využili například mezi dvěma vrcholy typu Uzel, kdy jeden představuje databázové schéma (rodič) a druhý tabulku (syn) uloženou ve schématu (rodič).
- **HasAttribute** je hrana spojující vrcholy Uzel a Atribut.
- **DirectFlow** je hrana představující přímý datový tok ze zdrojového do cílového uzlu.

2.9 Java

Java je široce používaný objektově orientovaný programovací jazyk. Jednou z hlavních výhod jazyku Java je jeho přenositelnost. Java je technologie skládající se z programovacího jazyka a softwarové platformy. Zdrojový kód napsaný v jazyce Java je kompilátorem převeden na takzvaný bytecode. Bytecode je instrukční sada pro Java virtuální stroj, zkráceně JVM, který je součástí běhového prostředí Java, zkráceně JRE. Bytecode běží bez úprav na jakémkoli systému, který podporuje JVM [20]. Všechny zdrojové kódy v mé bakalářské práci jsou psány v Javě 11.

2.10 SQL a Relační databáze

Relační databáze organizují data do tabulek, které jsou propojeny (odkazují jedna na druhou) pomocí společných dat [21]. Například kdybychom měli tabulky s daty o automobilech a jejich majitelích, tabulka majitelů by obsahovala identifikátory automobilů a naopak.

Strukturovaný dotazovací jazyk (anglicky „Structured Query Language“), zkráceně SQL, je doménově specifický jazyk používaný pro práci s daty nejčastěji v relačních databázích. SQL se skládá z mnoha typů příkazů, které si můžeme neformálně představit jako podjazyky. Jako příklad uvedu: jazyk dotazů na data (DQL), jazyk pro definici dat (DDL), jazyk pro kontrolu dat (DCL) a jazyk pro manipulaci s daty (DML) [22]. V nástroji dbt se setkáme převážně s DDL a DQL kategoriemi.

2.11 JUnit a Mockito

JUnit je otevřený software sloužící k vytváření a spouštění jednotkových testů pro jazyk Java. Jednotkové testy jsou určeny k testování malých částí kódu. V mé bakalářské práci použiji JUnit verze 4.13 pro vytváření tradičních a parametrizovaných jednotkových testů. Parametrizované testy umožňují spouštět stejný test opakovaně s použitím různých testovacích dat [23].

Mockito je otevřený software pro „mockování“ objektů[1] v testech. Mock objekt nebo také falešný objekt, má stejné rozhraní jako třída, ze které byl vytvořen, avšak umožňuje definovat výstup volání jednotlivých metod a testovat, jak bylo s objekty zacházeno. Například jestli a kolikrát byla metoda patřící objektu volána [24].

2.12 Apache Maven

Maven je nástroj, jehož hlavním úkolem je usnadnění řízení procesu vytváření spustitelných programů ze zdrojových kódů. Toho docílí pomocí takzvaného Project Object Model, zkráceně POM. Tento model popisuje projekt z pohledu závislostí na externích a interních knihovnách [25].

2.13 Spring a vkládání závislostí

Spring je otevřený software s velkým množstvím funkcí a využití. V mé bakalářské práci využiji funkce ze sekce „Core technologies“, především vkládání závislostí (anglicky „Dependency injection“). Vkládání závislostí je programovací technika, při které objekt přijímá závislosti objektů, na kterých je závislý, místo toho, aby je sám vytvářel. Toho je docíleno oddělením vytváření objektu od jeho použití. V nástroji Spring je k tomu použit IoC kontejner, který se stará o vytváření instancí objektu, známých jako Beany [26].

[1] Proces vytváření mock objektů.

2.14 Git

Git je systém pro správu verzí nejen zdrojových kódů. Uchovává historii všech provedených změn, umožňuje sledovat, kdo a kdy dané změny provedl. Informace o změnách jsou uloženy ve speciální databázi zvané Repozitář. Git značně zjednodušuje práci na projektech, kterých se účastní velké množství osob [27].

2.15 ItelliJ IDEA

IntelliJ je vývojové prostředí přizpůsobené pro vývoj aplikací v jazyce Java. Toto prostředí integruje nástroje jako jsou Maven, Spring nebo Git. Z tohoto důvodu je ItelliJ pro mou práci ideální volbou [28].

Kapitola 3
Analýza nástroje Fivetran

V této kapitole si představíme nástroj Fivetran, jeho transformace, obsah a strukturu metadat. Veškeré informace v této kapitole byly čerpány z oficiální dokumentace nástroje Fivetran [29] a nástroje dbt [30].

Fivetran není typickým ETL nástrojem, liší se pořadím fází ETL procesu. První prováděnou fází je extrakce, při které nástroj načte data z datových zdrojů. Poté co jsou data extrahována, přichází na řadu proces načítání (anglicky „Load“) dat do cílového úložiště. Když jsou data uložena v cílové destinaci, přichází na řadu fáze transformací. V cílovém úložišti jsou ponechána jak originální, tak transformovaná data.

3.1 Uživatelské rozhraní a řídící objekty

Hlavním řídícím objektem nástroje Fivetran je takzvaná destinace. Destinace obsahuje veškeré informace potřebné k provedení ETL procesu. Nejdůležitějšími informacemi, které obsahuje, jsou údaje o připojení[1] pro cílové úložiště a objekty zvané konektory. Objekt destinace je vytvářen v uživatelském rozhraní nástroje. V jedné instanci nástroje Fivetran můžeme vytvořit libovolné množství objektů destinací.

Konektor je objekt, obsahující informace o jednom datovém zdroji. Mezi ty nejdůležitější patří údaje o připojení ke zdrojovému úložišti, typ konektoru a informace o tom, jaká data budou přesunuta ze zdrojového do cílového úložiště. Konektory se vytvářejí skrze uživatelské rozhraní, kde si můžeme upravovat veškeré informace, které konektor obsahuje. Fivetran podporuje několik typů konektorů:

Databázové konektory jsou určeny pro databáze s B-stromovou a haldovou strukturou, jako jsou například MySQL a Postgres databáze. U každého konektoru si můžeme vybrat, jaká schémata, tabulky a sloupce budou kopírovány do cílového úložiště. Fivetran podporuje konektory pro technologie jako jsou například PostgreSQL, MySQL nebo MongoDB.

Souborové konektory jsou určeny pro soubory uložené v cloudových úložištích, privátních serverech nebo pro soubory z přijatých emailů. U každého konektoru si můžeme vybrat, jaké soubory, adresáře nebo z jakých emailů mají být soubory integrovány do cílového úložiště. Podporované jsou například technologie Google Sheets, Dropbox nebo servery podporující protokoly FTP, FTPS nebo SFTP.

[1] Pod pojmem údaje o připojení si můžeme představit veškeré informace potřebné k připojení k datovému úložišti. Například pro databáze by to byla adresa serveru, port a jméno databáze.

Konektory událostí jsou určeny pro nástroje, které slouží ke sledování chování spotřebitelů na webových stránkách, jako je například nástroj Webhooks.

Konektory aplikací jsou určeny pro jejich interní data, jako jsou například data o dosahu inzerovaných reklam. Fivetran podporuje aplikace, které umožňují extrakci dat skrze API, jako jsou například Facebook Ads, Google Ads nebo Google play.

Funkční konektor tvoří základ pro vytváření vlastních typů konektorů. Vlastní konektory mohou být použity například pro soukromé API nebo technologie nepodporované Fivetranem.

3.2 Fáze extrakce a načtení dat

Prvním krokem v těchto dvou fázích je nakopírování dat ze zdrojového úložiště do takzvané pracovní oblasti[2]. Jelikož Fivetran podporuje různé typy konektorů s různým formátováním dat, tak první operace, která je s daty v pracovní oblasti provedena, je namapování[3] dat do struktury schémat, tabulek a sloupců. Způsob, jakým jsou data namapována, závisí na typu konektoru.

3.2.1 Mapování dat do struktury cílového úložiště

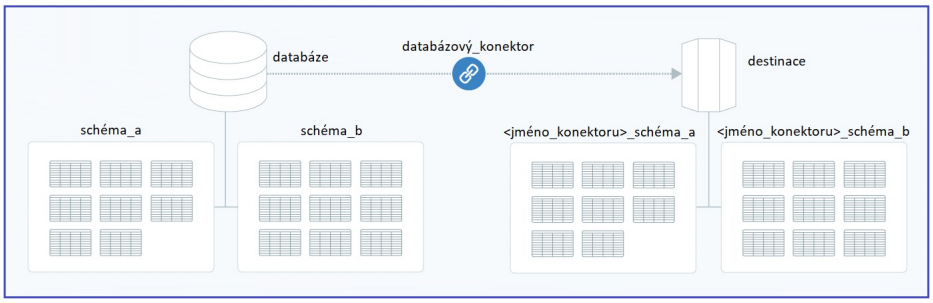
U databázových konektorů jsou data mapována do stejné hierarchie tabulek a sloupců, v jaké byly uloženy i ve zdrojové databázi. Změněna jsou pouze schémata, jejichž názvům jsou předřazeny jména konektorů. Ilustraci toku dat skrze databázový konektor lze pozorovat na obrázku 3.1.

Souborové konektory mapují všechna data do jednoho schématu, jehož jméno odpovídá jménu konektoru. Každý soubor je mapován do právě jedné tabulky, jejíž jméno odpovídá jménu souboru. Způsob mapování dat do sloupců je závislý na typu souboru. U CSV souborů budou vygenerovány názvy sloupců podle vstupů prvního řádku. U JSON souborů bude vytvořen jeden sloupec pro každé pole nejvyšší úrovně.

U konektorů událostí jsou data jednoho konektoru uložena do jednoho schématu a jedné tabulky. Jména jak pro schéma, tak tabulku lze nastavit v uživatelském rozhraní nástroje. Konektory nabízejí dva způsoby mapování dat do sloupců. Konektor všechna data uloží do jednoho sloupce nebo odvodí jména sloupců ze struktury příchozích dat.

Funkční konektor mapuje data do jednoho schématu, jehož jméno odpovídá jménu konektoru. Hierarchie tabulek a sloupců je plně závislá na individuální implementaci konektoru.

Obrázek 3.1 Ilustrace toku dat skrze databázový konektor. [29]



[2]Pracovní oblast je dočasné úložiště, ve kterém Fivetran uchovává a zpracovává data před nahráním do cílového úložiště.

[3]Pod pojmem *namapovat* si můžeme představit upravení dat tak, aby mohla být uložena v určité formě.

3.2.2 Jmenné konverze

Dalším krokem je aplikování jmenných konverzí na název každého schématu, tabulek a sloupců v pracovní oblasti. Fivetran využívá dvou sad konverzí. Pravidla pro názvy tabulek a sloupců:

- Jiné než ASCII znaky (například čínské znaky) jsou nahrazeny jejich transliterací[4].
- Povolena jsou pouze písmena, čísla a podtržítka. Jakékoli jiné znaky jsou nahrazeny podtržítkem.
- Názvy jsou převedeny z velbloudí notace (CammelCase) na notaci podtržítkovou (snake_case).
- Sekvence čísel jsou od ostatních znaků odděleny podtržítkem.
- Sekvence podtržítek jsou nahrazeny jedním podtržítkem.
- Velká písmena jsou převedena na malá.
- Názvům, které začínají číslem, je předřazeno podtržítko.
- Podtržítka nacházející se na začátku názvu tabulek jsou odstraněna.

Pravidla pro názvy schémat:

- Jiné než ASCII znaky (například čínské znaky) jsou nahrazeny jejich transliterací.
- Povolena jsou pouze písmena, čísla a podtržítka. Jakékoli jiné znaky jsou nahrazeny podtržítkem.
- Velká písmena jsou převedena na malá.

Jednotlivé konverze jsou prováděny v pořadí, ve kterých byly uvedeny. Pro databáze typu MySQL, Oracle. PostgreSQL a SQL Server je na tabulky a sloupce aplikovaná pouze konverze pro názvy schémat. Po aplikaci jmenných konverzí jsou data přesunuta z pracovní oblasti do cílového úložiště.

3.3 Fáze transformace dat

Fivetran podporuje dva typy transformačních projektů. Oba lze spouštět automaticky jako součást ETL procesu (fáze transformace je provedena po fázích extrakce a nahrání) v určitý čas bez ohledu na ETL proces nebo manuálně. Prvním typem projektu je základní SQL projekt (v anglickém originále „Basic SQL Project“). Jak již jeho jméno napovídá, projekt se skládá z krátkých SQL skriptů, kdy finální příkaz každého skriptu musí být typu DML, přesněji příkaz *select*. Projekt je možné vytvořit pouze skrze uživatelské rozhraní nástroje Fivetran.

Druhým typem je takzvaný dbt projekt. Tento druh projektu je kompilován a spouštěn stejnojmenným nástrojem. Zkratka dbt znamená nástroj pro výstavbu dat (v anglickém originále „Data Building Tool“). Nástroj není vyvíjen společností Fivetran, ta pouze integrovala jeho jádro do svého produktu. Tento projekt nelze vytvořit v uživatelském rozhraní Fivetranu, jako tomu je v případě základního sql projektu. Díky integrovanému jádru je Fivetran schopný pouze kompilovat a spouštět dbt projekty. Před samotným spuštěním je potřeba projekt připojit k určité destinaci, ve které má provést transformace. K tomu budeme potřebovat git repozitář, do kterého dbt projekt uložíme. V uživatelském rozhraní Fivetranu následně připojíme tento repozitář a nastavíme způsob spouštění. Fivetran se umí připojit pouze k vybraným git poskytovatelům, kterými jsou GitHub, GitLab, BitBucket, Azure Repos a AWS CodeCommit.

[4] Transliterace je přepis z jednoho písma do jiného. Například z cyrilice do latinky.

3.3.1 Struktura dbt projektu

Projekt je rozdělen do čtyř typů adresářů umístěných v kořenovém adresáři:

Model je adresář obsahující hierarchii konfiguračních souborů, podadresářů a modelů. Model je sql skript, obsahující pouze *select* příkazy rozšířené o příkazy jazyku Jinja.

Makro je adresář obsahuje krátké skripty složené z Jinja příkazů. Skripty jsou uloženy v souborech s příponou sql. Makra jsou krátké znovupoužitelné kusy kódu podobající se funkcím. Ukázku makra můžeme vidět v ukázce kódu 3.3.

Analýza je adresář se stejnou strukturou jako adresář s modely. Jediným rozdílem je, že SQL skripty se pouze zkompilují, ale nijak neovlivní cílovou databázi. Z tohoto důvodu je Fivetran při transformacích ignoruje.

Test je adresář obsahuje soubory s testy. Jako téměř vše v dbt, jsou testy SQL skripty složené ze *select* příkazů. Tyto příkazy se snaží najít záznamy, které nesplňují určité podmínky. Jako příklad uvedu tvrzení, že sloupec není nikdy null. Test se pokusí najít záznamy s hodnotou null a vrátí jejich počet. Pokud test nevrátí žádný neúspěšný řádek, tak je považován za úspěšný. Takovýto test můžeme pozorovat ve výpisu kódu 3.1.

```
{% test not_null(model, sloupec) %}
    select *
    from {{ model }}
    where {{ sloupec }} is null
{% endtest %}
```

Výpis kódu 3.1 Skript nástroje dbt, testující fakt, že sloupeček neobsahuje žádné záznamy s hodnotou null.

Projekt obsahuje dva typy souborů. Prvním z nich jsou YAML soubory, které slouží ke konfiguraci projektu. V projektu můžeme narazit na dva druhy těchto souborů.

Vedlejší řídící soubory se jménem ve tvaru *<jméno_souboru>.yml*, které slouží k nastavení zdrojů dat a jsou umístěny v adresáři s modely. Fivetran podporuje spouštění transformačních projektů pouze nad cílovým úložištěm[5], což znamená, že cílové úložiště je jak zdroj, tak cíl transformovaných dat. Z tohoto důvodu Fivetran ignoruje jméno zdroje i databáze a použije pouze strukturu schémat a tabulek. Ukázku konfigurace zdroje můžeme vidět v ukázce kódu 3.2.

```
sources:
- name: postgres_zdroj #jméno připojení
    database: fivetran_destinace_db #jméno zdrojové databáze
    schema: data_o_cestách
    tables:
    - name: adresa
    - name: klient
    - name: zájezd
    - name: rezervace
```

Výpis kódu 3.2 Ukázka konfigurace zdroje dat v YAML souboru.

[5] Cílovým úložištěm je myšleno datové úložiště definované v objektu destinace.

Hlavní řídící soubor má jméno *dbt_project.yml*. Tento soubor se vždy nachází v kořenovém adresáři projektu a slouží k jeho konfiguraci. Soubor je zodpovědný za mapování uživatelem vytvořené struktury adresářů na strukturu adresářů modelů, maker, testů a analýz. Dalším úkolem souboru je konfigurace adresáře modelu. Toho je docíleno pomocí zanořující se hierarchie jmen adresářů. Tuto hierarchii můžeme pozorovat ve výpisu kódu 3.4. V každém hladině hierarchie můžeme definovat konfigurační dvojici, skládající se ze jména a hodnoty konfigurace oddělené dvojtečkou. Konfigurace je využita pro adresář definovaný na dané hladině a pro všechny jeho podadresáře. V hierarchii se mohou opakovat stejné konfigurace, ale pro každý adresář platí pouze konfigurace definovaná na jeho hladině nebo první z konfigurací, na kterou narazíme, posouváme-li se hierarchií směrem ke kořenovému adresáři. Ukázku hierarchie modelů můžeme pozorovat v ukázce kódu 3.4.

Druhým typem souborů jsou SQL skripty. Nejedná se však o čisté SQL skripty, jelikož jsou složeny z SQL a Jinja příkazů. Jinja je šablonovací jazyk, který rozšiřuje SQL o řídící struktury, jako jsou například *for* smyčky. Dbt také podporuje tvorbu maker a urychluje vytváření připojení ke zdrojům dat. Jinja příkazy jsou lehko rozlišitelné od sql příkazů, jelikož se vždy nacházejí uvnitř složených závorek. Dbt rozlišuje tři druhy příkazů:

Výrazy jsou obaleny ve zdvojených složených závorkách **{{ ... }}**. Používají se k odkazování na proměnné a volání maker. Ukázku tohoto využití si lze prohlédnout v těle kódu 3.3.

Řídící příkazy se nacházejí uvnitř dvojitých závorek s procenty **{% ...%}**. Tento typ příkazů použijeme pro řízení toku, například k vytváření smyček, *if* příkazů nebo k definování maker.

Komentáře poznáme dle složených závorek a mřížek {# ...#}. Používají se ke komentování kódu a chrání text před kompilací.

```
{% macro centy_na_dolary(sloupec, presnost=2) %}
    (({{ sloupec }} / 100)::numeric(16, {{ presnost }})
{% endmacro %}
```

Výpis kódu 3.3 Makro v jazyce Jinja pro převod centů na dolary.

```
models:
    projekt_pro_fivetran:           # jméno kořenového adresáře
        schéma_normálních_dat:      # jméno adresáře s modely
            +materialization: table # nastavní konfigurace
            +schema: info_cesty
            etapa:                  # jméno podadresáře v adresáři s modely
                +materialized: table
                +schema: stg
                stg_rezervace:
                    +enabled: true
                stg_cesta:
                    +materialized: incremental
```

Výpis kódu 3.4 Ukázka konfigurace modelu dat v hlavním YAML souboru.

3.3.2 Typy konfigurací

Konfigurace můžeme v projektu nalézt na dvou místech. Jedním je hlavní konfigurační YAML soubor, ve kterém konfigurujeme skupiny souborů podle jejich adresáře. Ukázku tohoto přístupu lze nalézt ve výpisu kódu 3.4. Druhým způsobem je definování Jinja příkazů v jednotlivých SQL souborech. Tyto způsoby můžeme kombinovat s tím, že konfigurační příkazy jazyka Jinja mají větší prioritu. Ukázku konfiguračního příkazu lze nalézt v ukázce kódu 3.5. Nástroj dbt nabízí velké množství konfigurací, ale uvedu pouze konfigurace relevantní pro nástroj Fivetran.

3.3.2.1 Materializace

Materializace určuje, jakým způsobem bude se skripty zacházeno v cílovém úložišti. Jak bylo zmíněno dříve, výsledkem každého skriptu je příkaz *select*, který by sám o sobě nebyl příliš užitečný. Materializace obalí SQL skript příkazem typu DDL nebo DML. Jaký příkaz bude použit, záleží na typu materializace:

table: česky tabulka, obalí skript příkazem *create table*. Ukázku konfigurace si můžeme prohlédnout ve výpisu kódu 3.5.

view: česky pohled, obalí skript příkazem *create view*. Tento typ konfigurace je výchozím nastavením a využívá se vždy, když není přítomna žádná materializace.

incremental: česky přírůstková. Tato materializace při prvním spuštění v destinaci obalí SQL skript příkazem *create table* a při následujících spuštěních obalí skript příkazem *insert into*.

ephemeral: česky efemérní, jež lze pochopit jako dočasné. Při použití této materializace se vytvoří tabulka, která je později smazána. dbt podporuje práci s více procesorovými jádry najednou a každé pracuje ve své relaci (anglicky „session“). Kdyby dbt vytvořil pouze dočasnou tabulku, tak k ní má přístup pouze to jádro, které jí vytvořilo. Z toho důvodu je skript při spuštění obalen příkazem *create table* a po provedení všech ostatních skriptů z projektu je na tabulku zavolán příkaz *drop table*.

3.3.2.2 Konfigurace lokace vytvářených tabulek a pohledů

V předchozí podkapitole o materializaci jsem představil, jakým způsobem se SQL skripty ukládají v databázi, ale nezmínil jsem, pod jakými jmény a do jakých schémat se ukládají. O to se starají dvě konfigurace:

schema: je konfigurace, ovlivňující jméno schématu, ve kterém bude skript zhmotněn. V oficiální dokumentaci se toto nastavení označuje jako vlastní schéma (v anglickém originále „custom schema“). Finální jméno schématu se skládá z takzvaného cílové schématu (v anglickém originále „target schema“) a z vlastního schématu, která jsou oddělena podtržítkem. Jméno cílového schématu se nastavuje v uživatelském rozhraní, při vytváření spojení s git repozitářem obsahujícím dbt projekt. Finální název schématu má tvar *<jméno cílového schématu>_<jméno vlastního schématu>*.

alias: nastavuje jméno výsledné tabulky nebo pohledu. Pokud alias neuvedeme, jméno bude převzato z názvu souboru se skriptem.

3.3.2.3 Ostatní konfigurace

Jedním z posledních relevantních nastavení je uvozovkování (v anglickém originále „quoting“), které nastaví, jestli se má jméno databáze, schématu a tabulky či pohledu uzavřít do uvozovek. Uvozovkování umožňuje použití vyhrazených slov a speciálních znaků ve jménech.

Poslední relevantní konfigurací je povolení (v anglickém originále „enabled“). Tato konfigurace rozhoduje, jestli bude daný model nebo adresář využit při transformacích.

```
{{ config(materialized='ephemeral') }}
{{ config(alias='staging_file_for_client') }}
{{ config(schema='this_is_custom_schema') }}
```

Výpis kódu 3.5 Ukázka konfiguračního příkazu jazyku Jinja s různými druhy konfigurací

3.3.3 Ukázka šablony jazyka Jinja

V ukázce kódu 3.6 můžeme pozorovat šablonu, ze které bude vygenerován SQL dotaz, který vytvoří tabulku s názvem *reservations_and_clients* ve schématu *client_info*. Dotaz bude používat data z tabulek *stg_client* a *stg_reservation*. Výsledná tabulka bude obsahovat hodnoty se sloupci identifikátor klienta, jeho celé jméno a identifikátor rezervace.

```
{{ config(materialized='table') }}
{{ config(schema='client_info') }}
{{ config(alias='reservations_and_clients') }}


with clients as (
    select * from {{ ref('stg_client') }}
),
reservations as (
    select * from {{ ref('stg_reservation') }}
),
final_result as (
    select
        clients.id_client,
        full_name,
        id_reservation as client_reservation_count
    from clients
    left join reservations
        on clients.id_client = reservations.id_client
)
select * from final_result
```

Výpis kódu 3.6 Ukázka šablony jazyka Jinja pro SQL dotaz.

Kapitola 4

Analýza nástroje Looker

V této kapitole představím nástroj Looker a jeho transformace. Veškeré informace v této kapitole byly čerpány z oficiální dokumentace nástroje Looker [31].

Looker je bussines intelligence nástroj specializující se na vizualizaci dat. Od ostatních BI nástrojů se odlišuje vlastním jazykem zvaným LookML. Tento jazyk slouží k sestavování datových modelů využívaných ve vizualizacích. Looker nabízí dvě hlavní vizualizace, náhled (anglicky „Look“) a Nástěnku (anglicky „Dashboard“), skládající se z různých druhů grafů a tabulek. V následujících kapitolách představím možnosti jazyka LookML, strukturu a závislosti Pohledů a Nástěnek.

4.1 LookML

LookML je jazyk závislostí (anglicky „dependency language“), sloužící k popisování vztahů dat v SQL databázích. Jazyk vytváří dva druhy souborů, Model a Pohled (anglicky „View“). Tyto soubory jsou sdružovány do projektů, které popisují model dat, jenž je pak využíván při vytváření vizualizací. Projekt však neslouží pouze k popisu datového modelu, ale Looker ho používá i k vygenerování SQL dotazů, pomocí kterých načte data z databáze a následně jimi datový model naplní. Tento proces načítání a ukládání dat je prováděn i při aktualizaci dat v datovém modelu.

Vytváření LookML projektů je založené na DRY principu (anglicky „Don’t repeat yourself“). Soubory projektu mohou odkazovat na již definované struktury z ostatních souborů projektu. Tento přístup je využíván i při vytváření jednotlivých SQL dotazů, jelikož jsou jednotlivé soubory projektu používány při generování různých SQL dotazů opakovaně. To značně zjednodušuje vytváření komplexních datových modelů.

4.1.1 Druhy souborů jazyka LookML

Prvním ze souborů je soubor s Pohledy (anglicky „View“). Soubor obsahuje sadu Pohledů, kde každý z nich reprezentuje jednu databázovou nebo derivovanou tabulku. Derivovaná tabulka je SQL dotaz, s jehož výsledkem můžeme pracovat jako s klasickou databázovou tabulkou. Pohled se skládá ze jména databázové tabulky nebo v případě derivované tabulky z SQL skriptu, pomocí kterého je tabulka vytvořena. Pohledu reprezentující derivovanou tabulku lze pozorovat v ukázce kódu 4.1 a Pohled reprezentující databázovou tabulku ve výpisu kódu 4.2. Dále Pohled obsahuje definice Dimenzí a Měření (anglicky „dimension“ a „measure“), které definují a upravují sloupce tabulky, kterou Pohled reprezentuje.

```
view: customer_order_summary {
  derived_table: {
    sql:
      SELECT
        customer_id,
        MIN(DATE(time)) AS first_order,
      FROM orders
      GROUP BY customer_id
  }
  dimension: customer_id {
    type: number
    sql: ${TABLE}.customer_id
  }
```

Výpis kódu 4.1 Ukázka Pohledu reprezentujícího derivovanou tabulku.

Dimenze reprezentuje jeden nebo kombinaci sloupců z tabulky definované Pohledem. Oba případy můžeme pozorovat v ukázce kódu 4.2. Každé Dimenzi může být přiřazen typ, který určuje její formát a obohacuje ji o přídavné vlastnosti. Například Dimenze typu řetězec (anglicky „string“) umožňuje spojovat tabulky pomocí jakýchkoliv SQL výrazů. Typ Dimenze zvaný číslo (anglicky „number“) může spojovat sloupce pomocí matematických znamének. Mimo běžné datové typy, se kterými se můžeme setkat i v databázích, obsahuje Looker i typy jako jsou například vzdálenost, či lokace (anglicky „distance“ a „location“). Lokace je definována pomocí zeměpisné šířky a délky. Vzdálenost je definována pomocí dvou lokací.

Stejně jako Dimenze, tak i Měření pracuje s hodnotami jednoho nebo více sloupečků. Měření se od Dimenzí liší tím, že nespojuje řádky sloupců, ale provádí s nimi výpočty. Měření by se dala přirovnat k agregačním funkcím používaných v jazyce SQL. Jakým způsobem bude se sloupečkem nebo jejich kombinací nakládáno určujeme pomocí typů. Looker podporuje Měření typu *min* a *max*, jejichž výsledkem bude nejmenší a největší hodnota ze sloupce. Dále také podporuje typy *sum* nebo *count*, které sečtou hodnoty sloupců nebo jejich počet. Looker jednotlivé typy Měření obohacuje o formát hodnot. Například Měření typu *sum* lze nastavit přesnost zaokrouhlování.

```
view: letadla {
    sql_table_name: letové_statistiky.letadla
    dimension: kód_motoru {
        type: string
        sql: ${TABLE}.kód_motoru
    }
    dimension: název_a_identifikátor_letadla {
        type: string
        sql: CONCAT(${TABLE}.name, ':', ${$TABLE}.id)
    }
    measure: počet_letadel {
        type: count
        sql: ${TABLE}.id
    }
}
```

Výpis kódu 4.2 Ukázka souboru s jedním Pohledem.

Druhým ze souborů je takzvaný Model, který obsahuje sadu Průzkumů (anglicky „Explore“). Průzkum si lze představit jako jednu z výsledných tabulek vytvořenou LookML projektem, která bude použita ve vizualizacích nástroje Looker. Průzkum se skládá z Pohledů a definic jejich připojení do Průzkumu. V ukázce kódu 4.3 můžeme pozorovat definici Průzkumu s názvem *sklad*, který k sobě připojuje dva Pohledy *produkt* a *distribuční_centra*. Každé připojení pohledu je definováno klíčovým slovem *join*. Připojení obsahuje tři hlavní nastavení, určující jakým způsobem budou řádky Pohledu s řádky Průzkumu napojeny.

Nastavení definované klíčovým slovem *type* určuje jaké řádky pohledu budou do Průzkumu zahrnuty. Může nabývat hodnot:

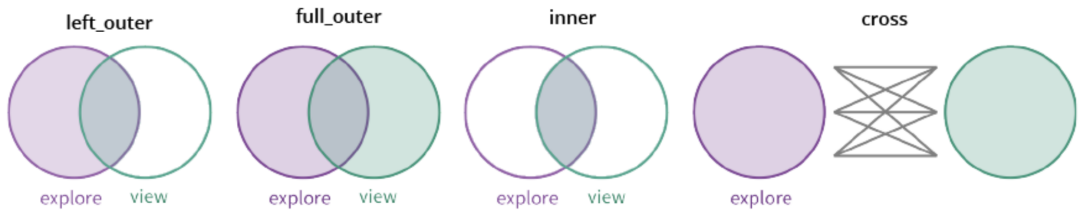
Left_outer je ekvivalentní s SQL připojením s názvem *LEFT JOIN*. Při jeho použití jsou k Průzkumu připojeny pouze řádky z Pohledu, které obsahují společná data.

Full_outer je známý v SQL jako *FULL OUTER JOIN*. Připojuje všechny řádky z Průzkumu a Pohledu i přesto, že neobsahují žádná společná data.

Inner je ekvivalentní s SQL připojením s názvem *JOIN* nebo také *INNER JOIN*, který připojuje pouze řádky z Průzkumu a Pohledu, které mají společná data.

Cross je známý v SQL jako *CROSS JOIN*. Ten vytvoří kartézský součin řádků Průzkumu a Pohledu.

Obrázek 4.1 Ilustrace nastavení pro zahrnutí řádků z Pohledu do Průzkumu. [31]



Nastavení definované pomocí klíčového slova *relationship* určuje kolik řádků z Pohledu může být spojeno s jedním řádkem z Průzkumu a naopak. Nastavení nabývá hodnot:

One_to_one značí, že každý řádek z Průzkumu může být spojen maximálně s jedním řádkem z Pohledu. Stejné platí i v opačném směru od Pohledu k Průzkumu.

Many_to_one nám říká, že jeden řádek z Pohledu může být spojen s libovolným počtem řádků z Průzkumu, ale řádky Průzkumu mohou mít vazbu maximálně s jedním řádkem z Pohledu. Tento vztah můžeme pozorovat ve výpisu kódu 4.3, který říká, že jeden řádek z Pohledu *produkt* může být spojen s více řádky Průzkumu *zboží_skladem*.

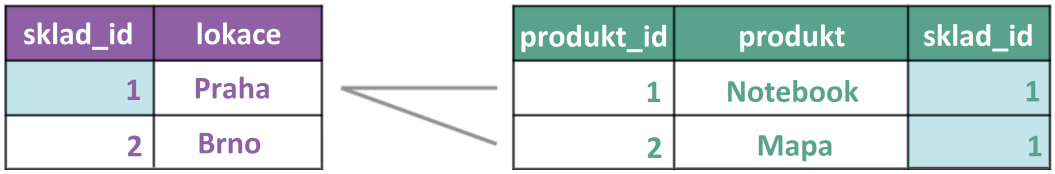
One_to_many značí, že jeden řádek z Průzkumu může mít libovolné množství vazeb s řádky v Pohledu, ale řádky Pohledu mohou být spojeny maximálně s jedním řádkem z Průzkumu. Jedná se tedy o opak vazby *Many_to_one*. Ukázku vztahu můžeme pozorovat na obrázku 4.2.

Many_to_many je ekvivalentní s tím, že každý řádek z Pohledu může být spojen s libovolným množstvím řádků z Průzkumu a to samé platí i v opačném směru.

Nastavení definované pomocí klíčového slova *sql_on* určuje, jaké sloupce jsou použity při mapování jednotlivých řádků a jakou podmínku musí splňovat jejich hodnoty, aby byly řádky spojeny. V ukázce kódu 4.3 u připojení Pohledu s názvem *produkt*, můžeme pozorovat nastavení, které vyžaduje, aby se hodnota sloupce *id* z Průzkumu s názvem *sklad* rovnala hodnotě

sloupce *sklad_id* z pohledu *produkt*. U připojení druhého pohledu s názvem *distribuční_centra* vidíme, že hodnoty ze sloupce Průzkumu, přidaného s předchozím pohledem, s názvem *produkty.distribuční_centra_id* se musí rovnat hodnotám sloupce z Pohledu s názvem *distribuční_centra_id*.

Obrázek 4.2 Ilustrace vazby řádků druhu *one_to_many* pro Průzkum a Pohled s názvy *sklad* a *produkt*. [31]




```
include: "/views/produkty.view"

explore: sklad {
  join: produkt {
    type: left_outer
    sql_on: ${sklad.id} = ${product.sklad_id}
    relationship: many_to_one
  }

  join: distribuční_centra {
    type: left_outer
    sql_on: ${produkty.distribuční_centra_id} = ${distribuční_centra_id}
    relationship: many_to_one
  }
}
```

Výpis kódu 4.3 Ukázka struktury souboru zvaného Model v jazyce LookML.

4.2 Druhy vizualizací

Looker nabízí dva druhy vizualizace Náhled a Nástěnku. V následujících podkapitolách představím tyto vizualizace, jejich závislost a propojení s LookML projektem.

4.2.1 Náhled

Náhled se specializuje na vizualizaci jednoho SQL dotazu, který je generován na základě Průzkumu definovaného v souboru s modelem v jazyce LookML. Při vytváření Náhledu můžeme využívat Měření a Dimenze, které byly definovány v Pohledech připojených k Průzkumu. Měření a Dimenze budu dále nazývat sloupci. Náhledy jsou používány jako mezikrok při vytváření Nástěnek. Každý Náhled může být vsazen do libovolného množství Nástěnek. Každá úprava Náhledu se automaticky propíše do všech Nástěnek, jež ho obsahují. Náhledům lze nastavit automatické periodické aktualizace, při kterých Náhled vygeneruje SQL dotazy pro jednotlivé Průzkumy. Tyto dotazy jsou použity k extrakci potřebných dat z databáze. Aktualizace lze provést i manuálně. Náhled podporuje velké množství vizualizací.

Tabulky: Náhledy podporují různé druhy tabulek. Tabulka jedné hodnoty (anglicky „Single Value Table“) zobrazí vždy hodnotu prvního řádku ze sloupce, který je nejvíce vlevo. Tabulka jednoho záznamu (anglicky „Single Record Table“) vizualizuje první řádek z vybraného Průzkumu. Klasická tabulka zobrazuje všechny řádky vybraných sloupců. Mračno slov (anglicky „World Cloud“) představuje uskupení hodnot jednoho sloupce, které jsou rotovány o násobky devadesáti stupňů pro navození podoby mraku.

Kartézské grafy: Náhledy nabízí sloupcové grafy, vodorovné i svislé (anglicky „Column Chart“ a „Bar Chart“). Bodové grafy (anglicky „Scatterplot“) a spojnicové grafy jsou složené z jedné i více spojnic (anglicky „Line Chart“ a „Area Chart“).

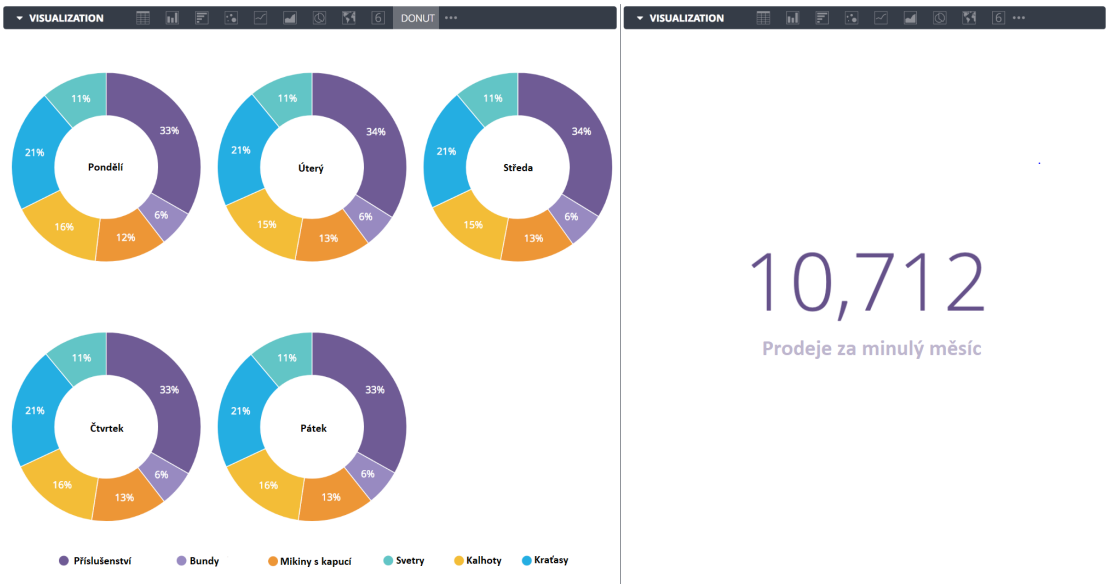
Progresivní grafy: Nabízí trychtýřové grafy (anglicky „Funnel Chart“) jak horizontální, tak vertikální, vizualizující hodnoty skupiny sloupců. Vodopádové grafy (anglicky „Waterfall Chart“) vizualizují jeden nebo dva sloupce. Grafy s časovou osou (anglicky „Timeline Chart“) vyžadují dva sloupce s časovým formátem a sloupec s daty libovolného formátu.

Koblihové a koláčové grafy: Koláčový graf (anglicky „Pie Chart“) vizualizuje právě jednu Dimenzi a jedno Měření. Koblihové grafy (anglicky „Donut Chart“) umožňují vizualizovat libovolné množství řádků, kdy je pro každý z nich vytvořena jedna kobliha.

Mapy: Mapy jsou dvou druhů, interaktivní a statické. Statické mapy mohou vizualizovat data pro jednotlivé regiony nebo body na mapě. Obě vizualizace statických map potřebují speciální Dimenzi typu lokace nebo typu řetězec obsahující třípísmenný kód regionu. Náhledy podporují velké množství specializovaných interaktivních map, jako například mapy silnic a dálnic, které si lze rozkliknout pro zobrazení tabulky s informacemi pro vybranou komunikaci.

Každý z výše zmíněných grafů lze dále stylizovat pomocí barev a přídavných popisků.

Obrázek 4.3 Ukázka koblihového grafu a tabulky jedné hodnoty pro Náhled.



4.2.2 Nástěnka

Nástěnky mohou být vytvořeny dvojím způsobem, skrze uživatelské rozhraní Lookeru nebo pomocí jazyka LookML. Oba typy Nástěnek se skládají ze tří druhů dlaždic.

Dlaždice s Náhledy obsahují referenci na předvytvořený Náhled s vizualizací. Tvorba Náhledů je popsána v předchozí podkapitole.

Textové dlaždice obsahují text psaný značkovacím jazykem Markdown, který slouží k tvorbě formátovaného textu. Looker však nepodporuje všechny jeho funkce. V dlaždicích můžeme používat nadpisy, tabulky, zvýrazňování textu, číslované a nečíslované odkazy, hyperlinky, zvýrazňování kódu, blokové uvozovky a dokonce i obrázky.

Dlaždice s dotazem obsahuje jednu vizualizaci dat. V podstatě se jedná o totožnou vizualizaci jakou je Náhled, ale vytvářená je až při tvorbě Nástěnky. Dlaždice s dotazem je plně integrovaná do Nástěnky a nelze ji znovu používat v jiných vizualizacích.

4.2.2.1 LookML Nástěnka

Vývojářské Nástěnky jsou tvořeny YAML soubory psanými v jazyce LookML. Každá Nástěnka tohoto typu musí obsahovat hlavičku s interním a zobrazovaným názvem Nástěnky (*dashboard* a *title*), se způsobem zobrazení Nástěnky a jednotlivých vizualizací (*preferred_viewer* a *layout*). Hlavička může dále obsahovat nepovinné údaje, jako je popisek Nástěnky (*description*), specifikace velikosti zobrazovaného názvu(*tile_size*) nebo interval automatických aktualizací Nástěnky (*refresh*). Nastavení těchto parametrů a možné hodnoty můžeme pozorovat v ukázce kódu 4.4.

```
- dashboard: interní_jméno_nástěnky
  preferred_viewer: dashboards | dashboards-next
  title: "název Nástěnky zobrazující se ve vizualizaci"
  description: "popisek Nástěnky"
  rows:
    - elements: [název_dlaždice, název_dlaždice, …]
  layout: tile | static | grid | newspaper
  refresh: 20 (seconds | minutes | hours | days)
  tile_size: 30
```

Výpis kódu 4.4 Ukázka struktury souboru s definicí Nástěnky v jazyce LookML.

Jak můžeme vidět ve výpisu kódu 4.4, parametr *preferred_viewer*, který určuje vzhled Nástěnky, nabývá dvou hodnot *dashbords* a *dashboards-next*. Rozdíl mezi oběma typy je minimální. V parametru *elements* zanořeném v parametru *rows* nastavíme interní názvy dlaždic, které chceme na Nástěnce zobrazit. Parametr *layout* určuje, jak budou jednotlivé dlaždice na Nástěnce uspořádány. Parametr může nabývat hodnot:

Newspaper: Vizualizace jsou zobrazeny v mřížce dvaceti čtyř sloupců a řádků. Výchozí velikost jednotlivých vizualizací je osm sloupců a šest řádků. Toto nastavení je použito i pro uživatelem vytvářené Nástěnky. Velikost vizualizací lze měnit přidáním parametrů *row* a *col* v jejich deklaracích. Počet sloupců a řádků mřížky lze měnit pomocí parametrů *width* a *height*.

Grid: Jednotlivé vizualizace se zobrazí s dynamickou velikostí. U tohoto nastavení nelze vynechat parametry *row* a *col* specifikující velikost vizualizace.

Static: Prvky Nástěnky se zobrazí v pořadí, ve kterém jsou definovány v YAML souboru pomocí parametrů *top* a *left*.

Tile: Pokud použijeme toto nastavení, tak jednotlivé dlaždice se na Nástěnce zobrazí ve stejném pořadí, v jakém byly definovány v LookML souboru.

Tvorbu jedné dlaždice můžeme pozorovat v ukázce kódu 4.5. Každá dlaždice musí mít přiřazený interní název *name*, Průzkum a jeho Měření nebo Dimenze, které budou použity ve vizualizaci. V Nástěnce definované pomocí LookML můžeme používat pouze Dlaždice s dotazem nebo textové dlaždice. Parametr *type* určí, o jakou vizualizaci se bude jednat. V případě ukázky 4.5 bude Nástěnka obsahovat tabulku jedné hodnoty, kterou si lze prohlédnout na obrázku 4.3.

```
elements:
- name: jméno_dlaždice
  type: single_value
  explore: objednávky
  measures: [objednávky.počet]
```

Výpis kódu 4.5 Ukázka definice dlaždice pro nástěnku v jazyce LookML.

Kapitola 5

Srovnání nástrojů Looker a Fivetran

V této kapitole popíšu vzájemné provázání nástrojů Looker a Fivetran a odhalím, jaký nástroj jsem si vybral pro implementaci prototypu.

Jak jsme se mohli dočíst v předchozích kapitolách, tak Looker a Fivetran jsou nástroje používané pro práci s daty. Způsob použití těchto nástrojů se velmi liší a oba nástroje pracují s daty v jiných částech jejich životního cyklu. Fivetran se specializuje na integraci dat z různých datových zdrojů do centralizovaných úložišť a při integraci data transformuje za účelem přípravy pro budoucí užití. Tato transformovaná centralizovaná data jsou využita nástrojem Looker. Ten z dat vytvoří interní datové modely pomocí jazyka LookML a ty poté používá ve svých vizualizacích.

Oba nástroje jsou zajímavé a k úkolům integrace a vizualizace dat přistupují originálním způsobem. Pro mou bakalářskou práci jsem se rozhodl pro implementaci skeneru pro nástroj Fivetran. Nástroj mě zaujal svým přístupem k ETL procesu a také integrovaným nástrojem dbt, který používá pro transformace. Fivetran jsme si vybral z důvodu jeho role v životním cyklu dat. Aby mohl nástroj Looker správně pracovat, musí být nejdříve použit nástroj Fivetran, který nám data centralizuje.

Kapitola 6

Návrh skeneru pro nástroj Fivetran

V této kapitole představím návrh skeneru pro nástroj Fivetran a jeho dva moduly, konektor a dataflow generátor. Také popíšu hierarchii souborů s metadaty a datový model, do kterého bude hierarchie nahrána.

6.1 Funkční požadavky

- F1: Skener bude schopný načíst metadata z manuálně dodaných souborů ve formátu JSON. Soubory musí být uložené v hierarchii adresářů, jejíž strukturu představím v následující podkapitole.
- F2: Skener bude schopný vygenerovat graf datového toku pro databázové konektory.
- F3: K vygenerování grafu datového toku použije skener metadata z nástroje Fivetran a použitých datových úložišť.
- F4: Vygenerovaný graf datového toku bude obsahovat informace na úrovni sloupců.
- F5: Skener bude schopný vytvořit graf datového toku jak pro základní, tak pro dbt transformační projekt.
- F6: Graf datového toku transformačních projektů bude vytvořen na základě metadat z nástroje dbt nebo z SQL skriptů základního SQL projektu.

6.2 Nefunkční požadavky

- N1: Skener bude řádně zdokumentovaný a čitelně implementovaný. Pro dokumentaci kódu použije JavaDocs. Všechny veřejné a chráněné metody tříd budou zdokumentovány.
- N2: Skener bude dodržovat strukturu modulů společnou pro skenery nástroje Manta.
- N3: Skener bude řádně otestovaný. Celkové pokrytí kódem bude minimálně 80%. Při implementaci bude použit nástroj SonarQube.
- N4: Skener bude lehce integrovatelný do nástroje Manta.

6.3 Modul Konektor

Konektor slouží k načtení metadat z hierarchie adresářů a JSON souborů. V této sekci popíšu, jak jsem tuto hierarchii navrhl. Dále také popíšu podmoduly konektoru, jejich funkce a závislosti. Na závěr této podkapitoly představím datový model, do kterého budou informace z metadat nahrány.

Také bych rád upozornil na dvojí význam slova konektor v této podkapitole. V nástroji Manta je konektor chápán jako modul a v nástroji Fivetran jako objekt participující v datovém toku.

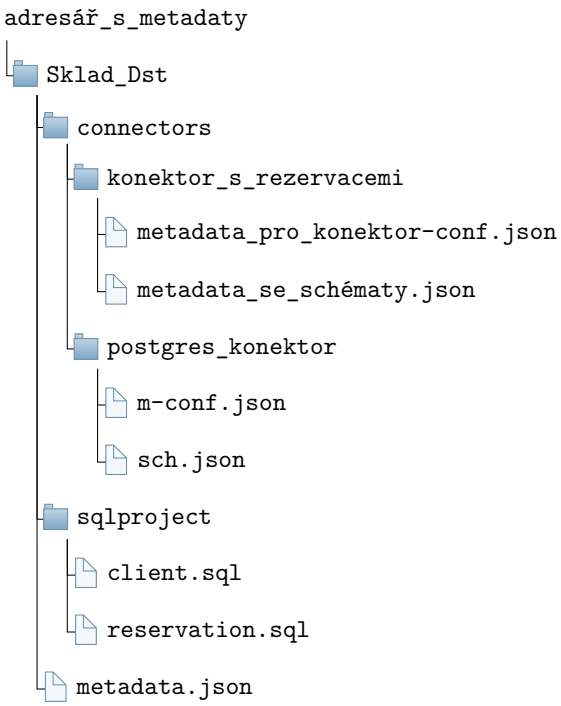
6.3.1 Struktura adresáře s metadaty

Hlavní adresář s metadaty je složen z podadresářů, kde každý z nich obsahuje veškerá potřebná metadata pro jednu destinaci. Jméno podadresáře je použito jako jméno destinace. Adresář jedné destinace můžeme vidět na obrázku 6.1 se jménem *Sklad_Dst*. Tento adresář obsahuje jeden konfigurační soubor destinace a dvě složky se jmény *connectors* a *sqlproject*. Jména těchto adresářů slouží jako jejich identifikátory a nesmí být změněny.

Adresář s názvem *connectors* obsahuje podadresáře s metadaty pro jednotlivé konektory. Na názvech podadresářů nezáleží a skener je ignoruje. Každý adresář by měl obsahovat dva soubory. Konfigurační soubor, jehož jméno obsahuje podřetězec *-conf*, podle kterého je identifikován při zpracovávání metadat konektoru. Druhý soubor obsahuje metadata o schématu konektoru.

Adresář s názvem *sqlproject* obsahuje metadata pro transformační projekt. Jak bylo zmíněno v kapitole 3, Fivetran podporuje dva druhy transformačních projektů. V případě dbt projektu bude adresář obsahovat pouze jeden soubor s veškerými metadaty projektu. V případě základního projektu bude adresář obsahovat jednotlivé SQL skripty. Na obrázku 6.1 můžeme vidět obsah adresáře pro základní SQL projekt skládající se ze dvou skriptů.

Obrázek 6.1 Ilustrace struktury adresáře s metadaty.



6.3.2 Struktura modulu

Modul obsahuje čtyři podmoduly, jejichž závislosti můžeme pozorovat na obrázku 6.2.

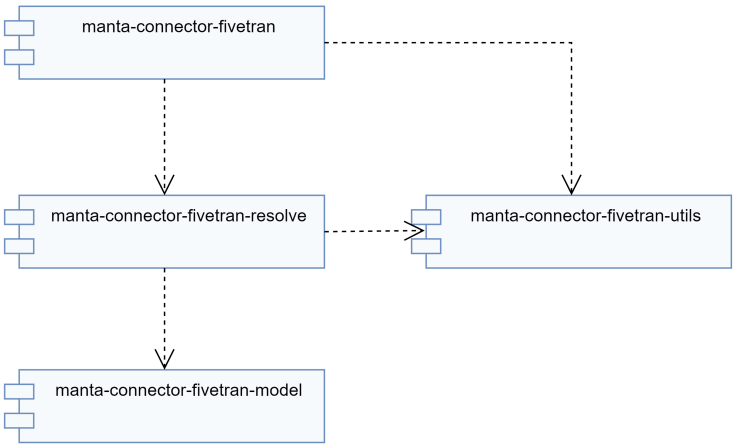
manta-connector-fivetran: je podmodul obsahující jednu třídu zvanou *FivetranReader*, která slouží jako vstupní bod pro nástroj Manta do modulu konektoru. Třída je používána k načtení co nejmenší části metadat, pro kterou lze vytvořit vizualizaci datového toku. V případě nástroje Fivetran se jedná o jednu destinaci. Manta předá třídě část metadat ze struktury vyobrazené v ukázce 6.1 obsahující metadata pro jednu destinaci. Třída využije funkcionalit ostatních podmodulů k načtení metadat do datového modelu, který Mantě vrátí. Ta využívá třídu v cyklech, dokud nejsou zpracovány veškeré destinace analyzované instance nástroje Fivetran.

manta-connector-fivetran-model: je podmodul, který obsahuje deklarace rozhraní (anglicky „interface“) a výčtové typy (anglicky „enumeration“) datového modelu, který uchovává metadata jedné destinace.

manta-connector-fivetran-resolver: implementuje rozhraní deklarovaná v podmodulu *manta-connector-fivetran-model* a obsahuje třídy pro zpracování souborů s metadaty.

manta-connector-fivetran-utils: podmodul obsahující pomocné třídy pro práci s JSON soubory a objekty.

Obrázek 6.2 Diagram modulů pro konektor.



6.3.2.1 Datový model pro uchovávání metadat

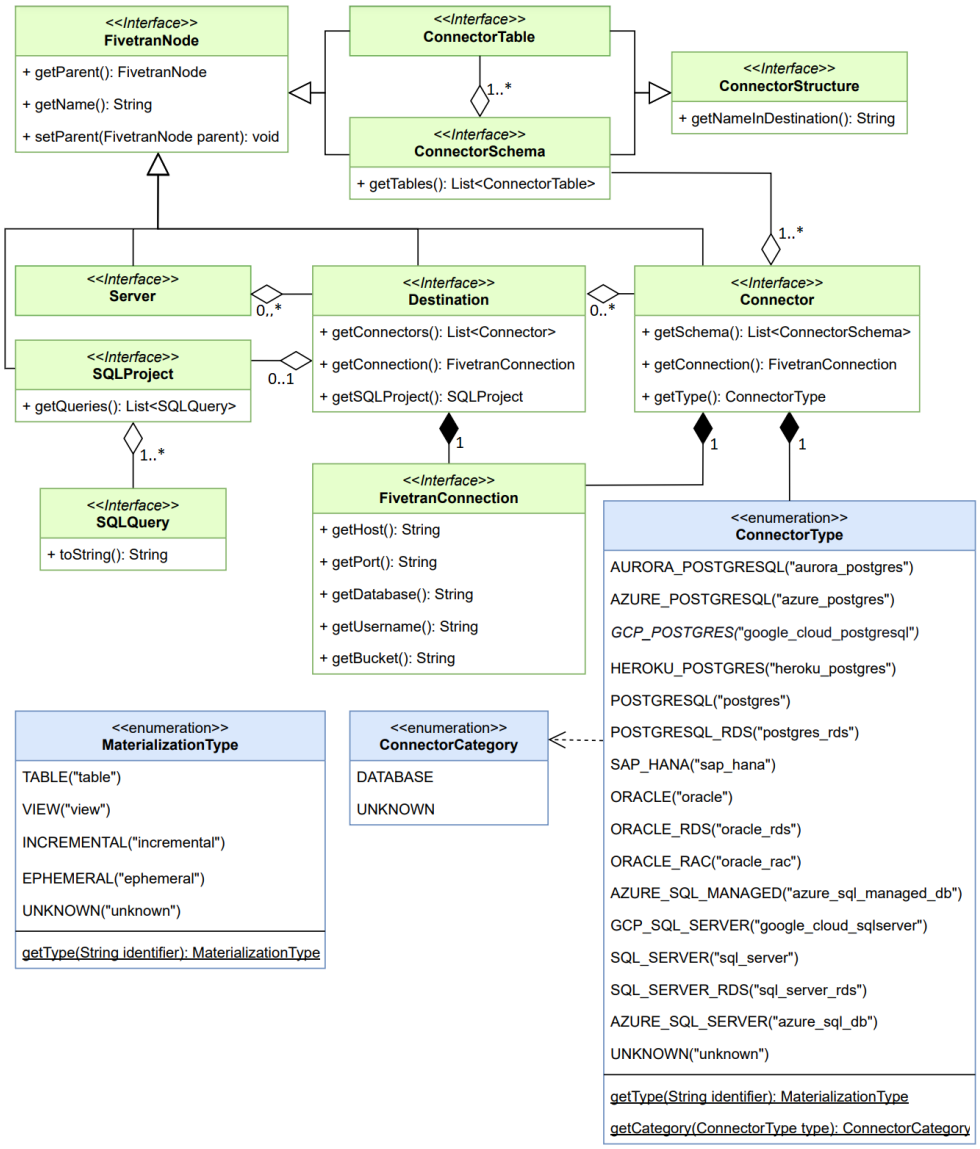
V této podkapitole popíšu návrh datového modelu z podmodulu *manta-connector-fivetran-model*, jehož diagram tříd je vyobrazen v ukázce 6.3. Jak jsem již zmínil, datový model je navržený tak, aby pojmul metadata jedné destinace. Destinace může obsahovat nejvýše jeden transformační SQL projekt a libovolný počet konektorů. Jediný povinný údaj, který musí destinace obsahovat, je připojení k cílové destinaci *FivetranConnection*.

SQL projekt se skládá z jednotlivých SQL skriptů. Každý konektor musí obsahovat údaje o zdrojové databázi *FivetranConnection*, typ konektoru a list schémat. Schémata a jejich tabulky obsahují informace o svém jméně jak ve zdrojové, tak cílové databázi. Tuto vlastnost získávají od rozhraní *ConnectorStructure*. Je nutné, aby obsahovaly názvy z obou úložišť, jelikož se mohou lišit. To je způsobeno jmennými konverzemi, o kterých jsem se zmiňoval v kapitole 3.

Na první pohled se může zdát, že rozhraní *Server* je v modelu zbytečné, ale není tomu tak. Server hraje důležitou roli v grafu datového toku, jelikož seskupuje destinace z jedné Fivetran instance. Toho je docíleno rozhraním *FivetranNode*, které přidává všem svým synům[1] schopnost pamatovat si svého otce[2]. Tato vlastnost je aplikována pro každou agregaci ve směru část-celek. Například destinace obsahuje list konektorů a každý z konektorů si pamatuje, jaká destinace je jeho otcem. Tato vlastnost je velmi užitečná při generování grafu datových toků.

Datový model obsahuje několik výčtových typů. Typ konektoru *ConnectorType*, který obsahuje průnik podporovaných databázových technologií nástrojů Fivetran a Manta. Kategorie konektoru *ConnectorCategory* rozděluje konektory dle typů zdroje dat. Prototyp skeneru podporuje pouze datové nástroje a tento výčtový typ je pouze příprava na budoucí rozšíření. Typ materializace *MaterializationType* je detailně popsán v kapitole 3 o nástroji Fivetran.

Obrázek 6.3 Diagram tříd pro podmodul *manta-connector-fivetran-model*.



[1]Syn ve smyslu dědění tříd.
[2]Otec v grafu datového toku

6.3.2.2 Hierarchie tříd pro zpracování souborů s metadaty.

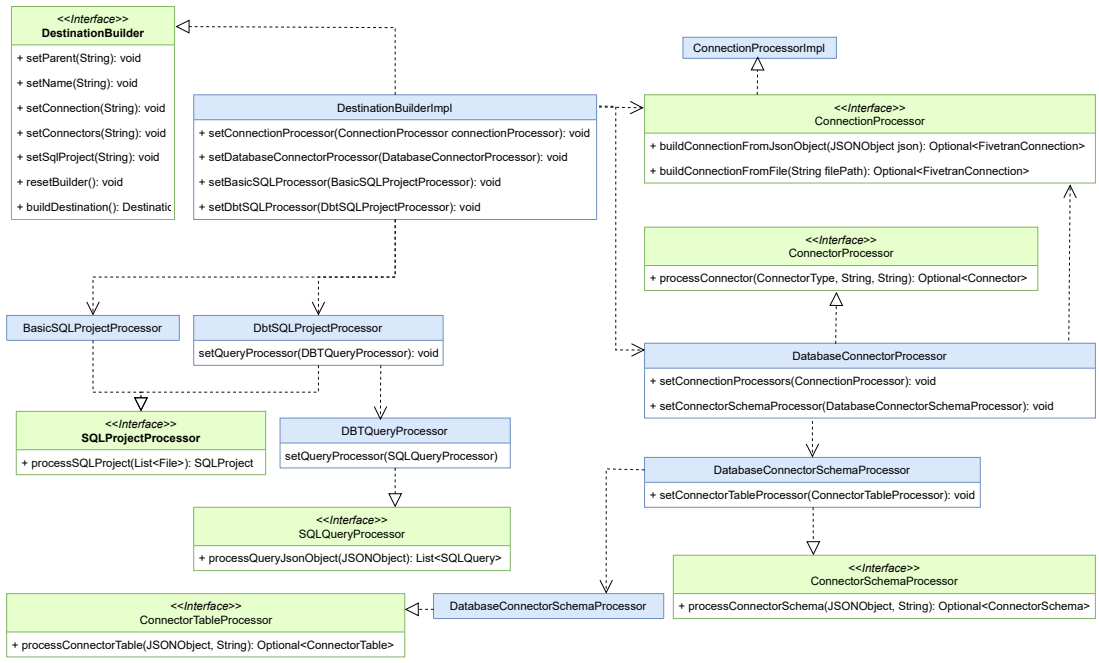
Za zpracování struktury adresářů a souborů s metadaty, na obrázku 6.1, je zodpovědný podmodul *manta-connector-fivetran-resolve*. Ten obsahuje hlavní řídící třídu zvanou *DestinationBuilder*, která řídí výstavbu datového modelu a hierarchii tříd zvaných procesory. Každý z těchto procesorů se specializuje na jeden objekt participující v datovém toku. Můžeme je pozorovat na diagramu 6.4.

Výstavba datového modelu začíná zavoláním metody *buildDestination*. Aby byl model úspěšně vytvořen, musí řídící třída obsahovat cestu ke konfiguračnímu souboru destinace a popřípadě i cesty k adresářům s konektory a SQL projektem. Tyto informace jsou jí předány třídou *FivetranReader* sídlící v modulu *manta-connector-fivetran*. Jako první se třída *DestinationReader* pokusí načíst data o cílovém úložišti destinace. Tento úkol je svěřen procesoru s názvem *ConnectionProcessor*, kterému je předána cesta ke konfiguračnímu souboru destinace. Pokud soubor obsahuje veškeré potřebné informace, pokračuje se ve výstavbě modelu. *DestinationBuilder* nahlédne do metadat konektorů a SQL projektu, zjistí jakého jsou typu a deleguje jejich výstavbu na konkrétní implementace procesorů *SQLProjectProccessor* a *ConnectorProcessor*.

Rozhraní *SQLProjectProccessor* má dvě implementace, jednu pro základní SQL projekt a druhou pro dbt projekt. Procesor základního SQL projektu dostane na vstupu list souborů s SQL skripty, které pouze načte do paměti. Procesor projektu dbt má však náročnější práci. Jeho vstupem je jeden JSON soubor s názvem *manifest*, dle kterého sestaví jednotlivé SQL skripty projektu.

Rozhraní *ConnectorProcessor* má pouze jednu implemetaci, která zpracovává metadata databázových konektorů. Procesor pro konektory se jako první pokusí z metadat získat potřebné informace o připojení ke zdrojové databázi. Pokud se mu to podaří, deleguje úkol načítání jednotlivých schémat konektoru na *ConnectorSchemaProcessor*. Ten pak opět deleguje načítání jednotlivých tabulek. Metadata poskytnutá nástrojem Fivetran neobsahují informace o zpracovávaných sloupcích a z toho důvodu, nejsou zahrnuta ani v datovém modelu, který lze pozorovat na obrázku 6.3.

Obrázek 6.4 Diagram tříd pro podmodul *manta-connector-fivetran-resolver*.



6.4 Modul Dataflow Generátor

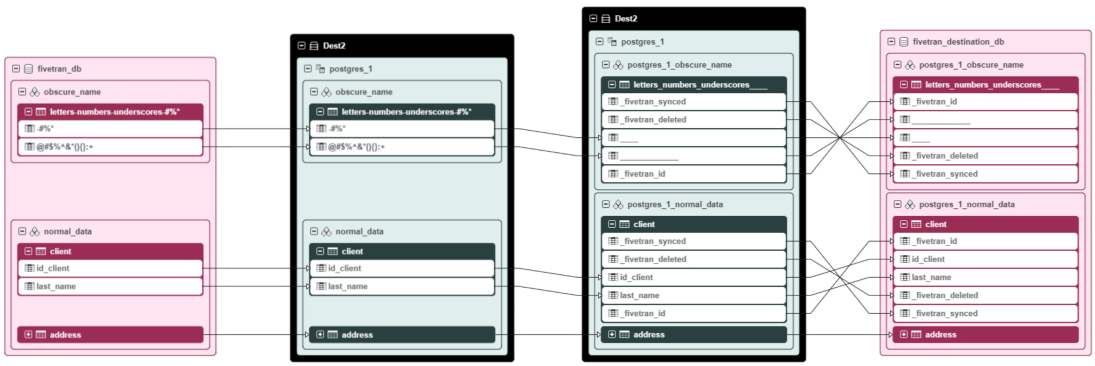
Generátor slouží ke generování grafu datového toku jedné destinace. Nástroj Manta používá třídy modulu v cyklech, dokud nevygeneruje grafy datových toků pro každou destinaci z instance, kterou analyzuje. Manta tyto grafy průběžně spojuje a výsledkem je graf obsahující vrcholy všech destinací instance Fivetran. V této sekci popíšu návrh vrcholů grafu vygenerovaného modulem.

6.4.1 Návrh vrcholů grafu datového toku

Jako první představím hierarchii uzlů pro fáze extrakce a načtení dat. Každý uzel v grafu nese jméno objektu, který reprezentuje. Jak jsem popsal v kapitole o nástroji Fivetran 3, data podstupují v konektorech různé úpravy. Aby tyto úpravy byly zachyceny ve výsledné vizualizaci, navrhl jsem dvě sady vrcholů pro konektory, zdrojovou a cílovou. Zdrojové uzly konektoru jsou vyobrazeny v ukázce 6.5 ve druhém bloku zleva. Tyto uzly reprezentují data před vstupem do konektoru. Cílové uzly konektoru, které lze pozorovat na obrázku 6.5 ve druhém bloku zprava, reprezentují data po úpravách, která v konektoru podstoupila. Růžové bloky z ukázky 6.5 představují strukturu zdrojové a cílové databáze.

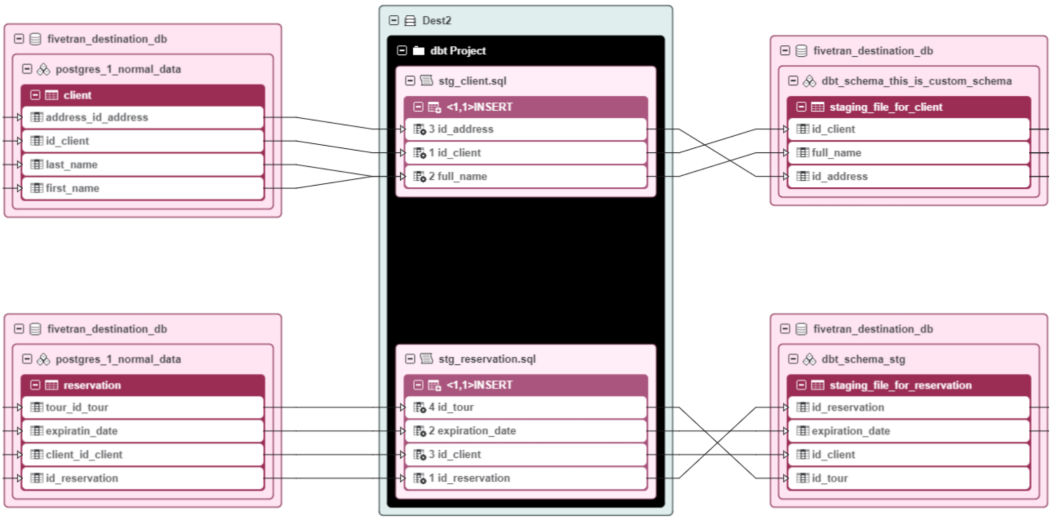
Vrchní uzly konektoru jsou vždy syny uzlu destinace, do které dodávají data. Uzel destinace můžeme v ukázce vidět pod názvem *Dest2* zahalený v černé barvě. Vrchní zdrojové a cílové uzly konektoru nesou v ukázce 6.5 název *postgres_1*. Konektory pracují s daty uloženými v databázích v hierarchii schémat, tabulek a sloupců. Tuto skutečnost je potřeba přenést do struktury grafu a vytvořit pro ni adekvátní uzly. Na obrázku 6.5 můžeme pozorovat uzly schémat jako je *normal_data* s uzly tabulek jako je *client* a v posledním zanoření i uzly sloupců, zobrazené v ukázce 6.5 s bílou barvou.

Obrázek 6.5 Ukázka hierarchie vrcholů grafu. Ukázku v plné velikosti jsem umístil do přílohy.



Hierarchii uzlů transformačního projektu si můžeme prohlédnout v ukázce 6.6. Vrchní uzel hierarchie nese název typu projektu *dbt Projekt* nebo *Basic SQL Projekt* a je synem vrcholu destinace, která v ukázce nese název *Dest2*. Uzel projektu obsahuje uzly jednotlivých SQL souborů, které se v projektu nacházejí. Tyto uzly můžeme v ukázce pozorovat pod jmény *stg_client.sql* a *stg_reservations.sql*. Každý SQL soubor obsahuje jeden SQL skript a tedy i každý uzel souboru obsahuje jeden uzel SQL skriptu. Tento uzel můžeme v ukázce pozorovat se jménem *<1,1>INSERT*, který obsahuje uzly výstupních sloupců. Růžové bloky na pravé a levé straně ukázky 6.6 reprezentují strukturu schématu databáze vstupních a výstupních dat skriptů.

Obrázek 6.6 Ukázka hierarchie vrcholů grafu SQL projektu. Ukázku v plné velikosti jsem umístil do přílohy.



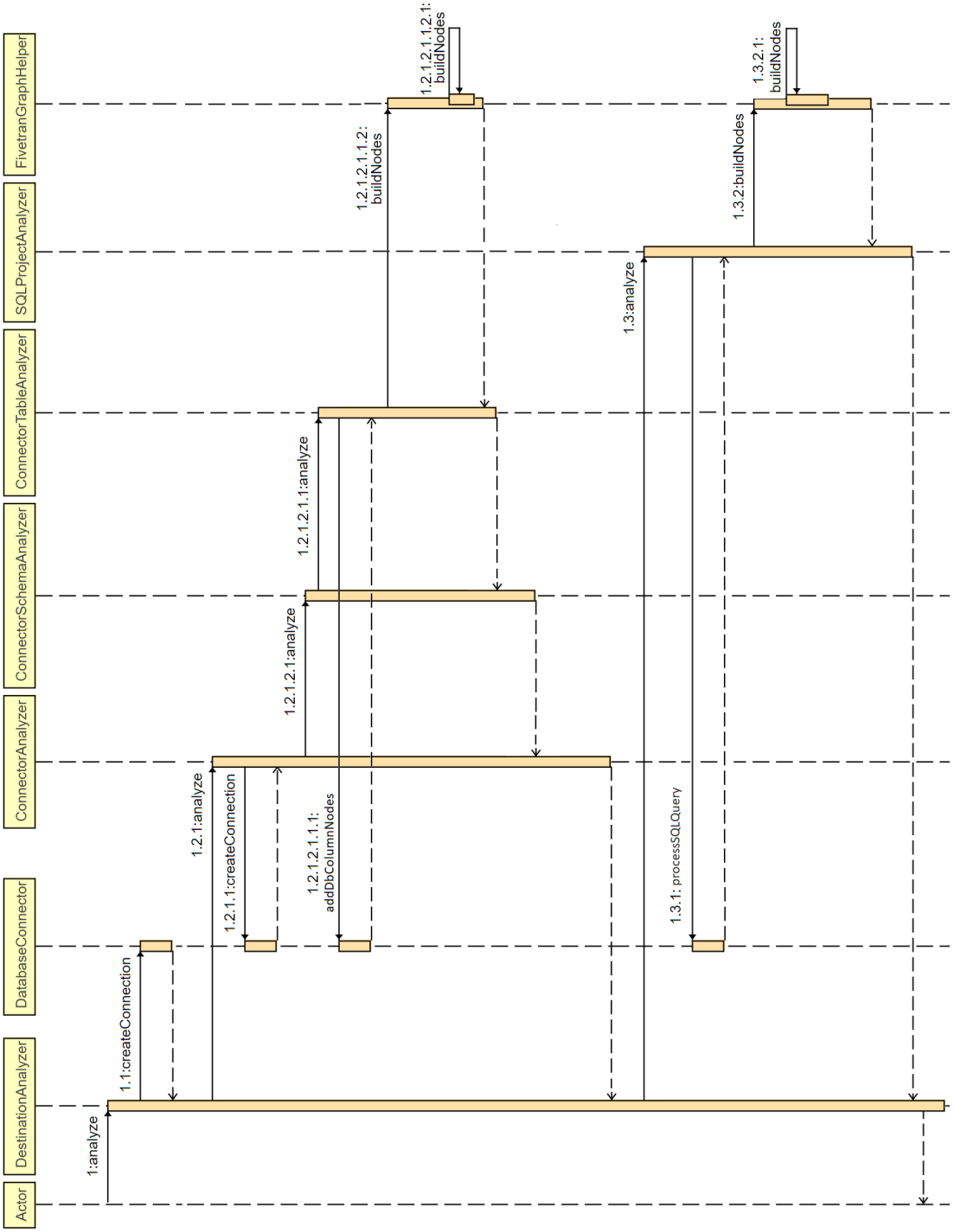
6.4.1.1 Hierarchie tříd pro generování grafu.

Pro generování grafu jsem navrhl hierachii rozhraní a tříd, jejichž vstupem je datový model jedné destinace, představený v ukázce 6.1. Vstupním bodem hierarchie je třída s názvem *Destination Analyzer*, která zpracuje připojení cílové databáze a deleguje generování uzlů jednotlivých konektorů a SQL projektu na třídy *ConnectorAnalyzer* a *SQLProjectAnalyzer*.

Třída *ConnectorAnalyzer* je zodpovědná za generování uzlů pro jeden konektor. Třída jako první vytvoří připojení ke zdrojovému úložišti, k němuž je analyzovaný konektor připojen. Poté třída deleguje analýzu schémat na třídu *ConnectorSchemaAnalyzer*. Ta je schopná analyzovat jedno schéma včetně všech jeho tabulek, čehož docílí postupným voláním metody *analyze* třídy *ConnectorTableAnalyzer*. Tato třída je nejzajímavější z celé hierarchie a stará se o zpracování dat sloupců jedné tabulky. Jak bylo zmíněno v předchozí podkapitole, datový model představený v ukázce 6.1 neobsahuje informace o sloupcích. Z tohoto důvodu musí třída *ConnectorTableAnalyzer* získat tyto informace z cílové a zdrojové databáze. K tomu slouží připojení vytvořená v předchozích třídách *Destination Analyzer* a *ConnectorAnalyzer*. Tato připojení jsou použita jako vstup pro interní nástroj *Manty* zvaný *Dataflow Query Service*, který je schopný vytvořit uzly sloupců databází a přidat je do grafu. Tyto uzly i s jejich umístěním v databázi můžeme pozorovat jako růžové bloky v ukázce 6.5. V sekvenčním diagramu 6.7 je tento nástroj reprezentován třídou *DatabaseConnector*. Jakmile získá třída *ConnectorTableAnalyzer* potřebné informace o sloupcích, deleguje vytvoření jednotlivých uzlů konektorů a destinace na třídu *FivetranGraphHelper*. Tyto vrcholy můžeme pozorovat jako dva prostřední bloky v ukázce 6.5

Analýza transformací je provedena třídou *SQLProjectAnalyzer*, která využije, v předchozím odstavci zmíněného, interního nástroje *Manty* zvaného *Dataflow Query Service*, který analyzuje jednotlivé SQL dotazy. Poté třída deleguje vytváření uzlů projektu na třídu *FivetranGraphHelper*. Vizualizaci výsledného grafu lze pozorovat v ukázce 6.6.

Obrázek 6.7 Sekvenční diagram popisující tvorbu grafu datového toku.



Kapitola 7

Implementace skeneru pro nástroj Fivetran

V této kapitole představím implementaci skeneru pro nástroj Fivetran. Zaměřím se na zajímavé a složité části modulů Konektoru a Generátoru.

7.1 Modul Konektor

Jak jsem již zmínil v kapitole návrhu modulu Konektoru 6.3, hlavním úkolem modulu je zpracovat metadata jedné destinace uložené ve struktuře JSON souborů, popsané v ukázce 6.1. V následujících podkapitolách představím ukázky kódu z jednotlivých podmodulů.

```
public List<File> listAllEntriesOfDirectory(String directoryPath) {
    List<File> entryList = new ArrayList<>();
    if (directoryPath == null) {
        return entryList;
    }
    try (DirectoryStream<Path> stream =
                Files.newDirectoryStream(Paths.get(directoryPath)))
    {
        for (Path path : stream) {
            entryList.add(path.toFile());
        }
    } catch (InvalidPathException | IOException e) {
        LOGGER.log(Categories.inputStructureErrors().processingFile().catching(e)
              .message("Unable to read files and directories from a directory " +
              "with path: " + directoryPath + "."));
        return Collections.emptyList();
    }
    return entryList;
```

Výpis kódu 7.1 Ukázka implementace metody *listAllEntriesOfDirectory* třídy *FileHelper*.

7.1.1 Metody podpůrných tříd Konektoru

Podpůrné třídy jsem umístil do podmodulu s názvem *manta-connector-fivetran-utils* a v této podkapitole představím některé jejich metody.

Metoda *listAllEntriesOfDirectory* třídy *FileHelper* slouží k načtení všech souborů a podadresářů jednoho adresáře určeného jeho cestou. Pokud metodě předám invalidní cestu k adresáři, metoda vrátí prázdný list. Jak můžeme vidět v ukázce 7.1, metoda nejdříve otestuje, jestli je cesta k adresáři definována nebo-li není *null*. Pro načtení jednotlivých elementů adresáře jsem se rozhodl použít *DirectoryStream*, který je rychlejší variantou oproti metodě *listFiles* nabízenou objektem *java.io.File*. Pokud cesta k adresáři není validní, může při vytváření *DirectoryStream* nebo při volání metody *Path.get* dojít k vyhození výjimky. Z tohoto důvodu jsem obalil tuto část kódu v *try-catch* bloku. *DirectoryStream* vyžaduje uzavření zdroje dat a z tohoto důvodu jsem využil variantu *try-with-resources*, která se o to při zachycení výjimky postará.

```
public Optional<JSONObject> getJSONObject(JSONObject json, String... keys) {
    if (json != null && keys != null && keys.length != 0) {
        try {
            for (String key : keys) {
                json = json.getJSONObject(key);
            }
            return Optional.of(json);
        } catch (JSONException e) {
            LOGGER.log(Categories.inputStructureErrors().parsingJson()
                  .file(json).catching(e));
        }
    }
    return Optional.empty();
}
```

Výpis kódu 7.2 Ukázka implementace metody *getJSONObject* třídy *JSONHelper*.

Při zpracovávání JSON objektů je často potřeba načíst jejich zanořený objekt. K tomuto účelu jsem implementoval metodu *getJSONObject* třídy *JSONHelper*. Její implementaci můžeme pozorovat v ukázce 7.2. Metoda na vstupu přijímá JSON objekt a pole objektů zvaných *String*, reprezentující zanořující se klíče hledaného JSON objektu. Pokud je vstup validní, metoda se pokusí v JSON objektu nalézt zanořující se hierarchii klíčů. Pokud se to metodě povede, zabalí nalezený JSON objekt v objektu zvaném *Optional*.

Optional je objekt používaný k indikaci skutečnosti, že metodou vrácený objekt nemusí být definovaný. K tomu může dojít, pokud JSON objekt ze vstupu není validní nebo neobsahuje vnořený objekt definovaný hierarchií klíčů. Pro tyto případy jsem v metodě použil *try-catch* blok, který oba tyto případy zachytí.

7.1.2 Metody zpracovávající soubory s metadaty

Třídy tohoto typu jsou umístěny v modulu *manta-connector-fivetran-resolver*. Jak jsem popsal v předchozí kapitole o návrhu modulu konektoru 6.3, třída *DestinationBuilder* řídí zpracovávání souborů s metadaty a výstavbu datového modelu, popsaného v ukázce 6.3. Tato třída je implementací takzvaného *Builder patternu*. K tomuto řešení jsem se uchýlil z důvodu komplexnosti objektu destinace. Dalším důvodem, proč jsem se rozhodl použít *Builder pattern*, je zaručení neměnitelnosti (anglicky „immutability“) objektu destinace a následně i ostatních objektů datového modelu.

Třídě jsou postupně předány cesty k adresářům a souborům s metadaty, popsané v ukázce 6.1. Jakmile má třída cesty ke všem potřebným souborům, je možné zavolat metodu *buildDestination*, jejíž implementaci si můžeme prohlédnout v ukázce 7.3. Třída deleguje vytváření svých částí na jednotlivé procesory a následně vytvoří instanci destinace, jejíž tvorba je dokončena voláním metody *setAsParent*. Vytváření jednotlivých částí destinace je pozdrženo až do metody *buildDestination* i přesto, že by se tyto části mohly vytvářet hned, jak je třídě předána cesta k odpovídajícímu souboru s metadaty. V takovém případě by ale docházelo k vytváření komplexních částí destinace, jako je třeba SQL Projekt, který by nebyl využit, protože metadata destinace, ke které objekt patří, neobsahují všechny povinné informace, jako je například připojení k cílové databázi. V takovém případě instanci destinace nelze vytvořit.

```
public Destination buildDestination() {
    if (parent != null && connectionFilePath != null) {
        Optional<FivetranConnection> connectionOpt = connectionProcessor
                    .buildConnectionFromFile(connectionFilePath);
        if (connectionOpt.isPresent()) {
            List<Connector> connectors = processConnectors();
            SQLProject project = processSQLProject();
            Destination destination = new DestinationImpl(
                parent, name, connectors, connectionOpt.get(), project
            );
            setAsParent(destination);
            return destination;
        }
    }
    return null;
}
```

Výpis kódu 7.3 Ukázka implementace metody *buildDestination* třídy *DestinationBuilder*.

Metoda *setASParent*, jejíž implementaci můžeme pozorovat v ukázce 7.4, nastaví objekt Destinace jako rodiče všem jeho součástem. Kvůli této části kódu nemohou být objekt destinace a ostatní části datového modelu plně neměnitelné (anglicky „immutable“). Tímto přístupem jsem docílil alespoň částečné neměnitelnosti, kdy jediným měnitelným atributem objektu je jeho rodič. Bohužel v tomto případě nejde docílit plné neměnitelnosti objektu, jelikož všechny části destinace musí být vytvořeny před vznikem samotné instance destinace. Kdybych vytvořil nejdříve destinaci a poté její části, atribut rodič by mohl být neměnitelný, ale narazil bych na problém s měnitelností jednotlivých částí destinace.

```
private void setAsParent(Destination destination) {
    destination.getConnectors().forEach(
            connector -> connector.setParent(destination)
    );
    SQLProject project = destination.getSQLProject();
    if (project != null) {
        project.setParent(destination);
    }
}
```

Výpis kódu 7.4 Ukázka implementace metody *setAsParent* třídy *DestinationBuilder*.

```
public Optional<FivetranConnection> buildConnectionFromJsonObject(JSONObject json) {
    Optional<JSONObject> jsonConfigOpt = JSONHelper.getJSONObject(
                json, JSON_KEY_DATA, JSON_KEY_CONFIGURATION
    );
    if (jsonConfigOpt.isPresent()) {
        JSONObject jsonConfig = jsonConfigOpt.get();
        if (jsonConfig.has(JSON_KEY_CONNECTION_DATABASE_HOST)) {
            return loadDatabaseConnection(jsonConfig);
        }
        if (jsonConfig.has(JSON_KEY_CONNECTION_BIGQUERY_HOST)) {
            return loadBigQueryConnection(jsonConfig);
        }
        LOGGER.log(Categories.inputStructureErrors().parsingJson().file(jsonConfig));
    }
    return Optional.empty();
}
```

Výpis kódu 7.5 Ukázka implementace metody *buildConnectionFromJsonObject* třídy *ConnectionProcessorImpl*.

Posledním typem tříd z modulu *manta-connector-fivetran-resolver* jsou takzvané procesory, které vytvářejí jednotlivé části datového modelu. Toho třídy docílí zpracováním souborů s metadaty nebo již načtených JSON objektů. Pro ukázku těchto tříd jsem si vybral třídy *ConnectionProcessorImpl* a *DBTQueryProcessor*.

```
{
  "code": "Success",
  "data": {
    "id": "brilliancy_laboratory",
    "group_id": "brilliancy_laboratory",
    "service": "postgres_warehouse",
    "region": "GCP_EUROPE_WEST3",
    "time_zone_offset": "-8",
    "setup_status": "connected",
    "config": {
      "host": "17.125.56.214",
      "database": "fivetran_destination_db",
      "password": "******",
      "port": "5454",
      "user": "fivetran_destination_user",
      "connection_method": "Directly"
    }
  }
}
```

Výpis kódu 7.6 JSON soubor obsahující metadata s připojením k cílové databázi.

Metoda *buildConnectionFromJsonObject* třídy *ConnectionProcessorImpl* přijme JSON objekt, jehož strukturu můžeme pozorovat v ukázce 7.6 a využije metodu *getJSONObject* z ukázky 7.2 k načtení zanořeného JSON objektu definovaného klíči *data* a *config*. Pokud takový vno-

řený objekt existuje, tak se metoda podívá na hodnotu klíče *host*, podle které vytvoří objekt *FivetranConnection* s připojením pro klasickou databázi nebo pro *BigQuery*. Následně metoda vytvořenou instanci obalí objektem Optional a vrátí ji volajícímu. Tato metoda je použita k vytvoření připojení k databázi destinace i konektoru.

```
"model.project_for_fivetran.clients_view": {
        ...
      "config": {
        ...
        "materialized": "view",
        ...
      },
      "database": "fivetran_destination_db",
      "schema": "dbt_schema_client_info",
      ...
      "alias": "clients_view",
      "compiled_sql": ...
      ...
}
```

Výpis kódu 7.7 JSON objekt obsahující vybraná metadata jednoho SQL dotazu dbt projektu.

Třída *DBTQueryProcessor* slouží k vytvoření jednoho SQL dotazu pro dbt projekt. Toho třída docílí zpracováním JSON objektu, který můžeme pozorovat v ukázce 7.7. Jelikož metadata neobsahují celý SQL dotaz, ale pouze jeho tělo, informace o jménu databáze, schématu, tabulky a jakým způsobem bude výsledek dotazu materializován, implementoval jsem třídu tak, aby správně sestavila SQL dotaz z poskytnutých informací. Jednotlivé části dotazů dbt projektu jsem popsal v kapitole 3.3.2.

Třída jako první načte materializaci SQL dotazu a následně i jméno databáze, schématu a tabulky. Tyto hodnoty můžeme pozorovat v ukázce 7.7, pod klíči *database*, *schema* a *alias*. Jakmile třída získá tyto informace, sestaví hlavní SQL příkaz, kterým obalí tělo dotazu obsaženého v metadatech pod klíčem *compiled_sql*. Příklad sestavení takovéhoto SQL příkazu jsem umístil do ukázky 7.8. Do projektu nejsou zahrnuty SQL dotazy s materializací typu *ephemeral*, jelikož na datový tok nemají žádný vliv.

```
private Optional<String> generateMainStatement(String identifiers,
                                               MaterializationType type) {
    switch (type) {
        case TABLE:
            return  Optional.of("create table " + identifiers + " as ");
        case VIEW:
            return Optional.of("create view " + identifiers + " as ");
        case INCREMENTAL:
            return Optional.of("insert into " + identifiers + " ");
        default:
            return Optional.empty();
    }
}
```

Výpis kódu 7.8 Ukázka metody *generateMainStatement* třídy *DBTQueryProcessor*

7.2 Modul Dataflow Generátor

Modul generátoru obsahuje dva hlavní typy tříd. Jedním z nich jsou analyzátory zodpovědné za zpracování struktury datového modelu, vytvořeného v modulu Konektor. Druhým z nich jsou pomocné třídy, jako je například *FivetranGraphHelper*, která vytváří jednotlivé uzly grafu.

Nejzajímavější částí třídy *FivetranGraphHelper* je metoda *buildNodes*, kterou jsem umístil do ukázky 7.9. Tato metoda přijme na vstupu objekt *FivetranNode*, ze kterého dědí každá třída datového modelu, jež reprezentuje objekt participující v datovém toku. Metoda zjistí, o jaký objekt z datového toku se jedná pomocí metody *assignSourceNodeType* a zjistí, jestli byl uzel již dříve vytvořen pomocí metody *isCached*. Pokud uzel ještě nebyl vytvořen, tak metoda rekurzivním voláním sama sebe vytvoří hierarchii všech rodičů uzlu a následně vytvoří i uzel samotný pomocí metody *createNode*.

```
public Node buildNodes(FivetranNode fivetranNode) {
    NodeType type = assignSourceNodeType(fivetranNode);
    NodeCacheKey cacheKey = new NodeCacheKey(fivetranNode, type);
    if (isCached(cacheKey)) {
        return getCachedNode(cacheKey);
    }

    String name = fivetranNode.getName();
    Node parentNode = (fivetranNode.getParent() != null)
            ? buildNodes(fivetranNode.getParent())
            : null;

    return createNode(cacheKey, name, type, parentNode);
}
```

Výpis kódu 7.9 Ukázka metody *buildNodes* třídy *FivetranGraphHelper*.

Třídy zvané analyzátory se starají o zpracování datového modelu popsaného v ukázce 6.2. Každý z analyzátorů zpracovává právě jeden objekt datového modelu. Jako zástupce těchto tříd jsem si vybral *DestinationAnalyzer* a *ConnectorTableAnalyzer* s metodami *analyze*.

```
public void analyze(Destination destination, FivetranGraphHelper gh) {
    Connection destinationConnection = databaseConnector.createConnection(
            destination.getConnection(), ConnectorType.UNKNOWN
    );

    List<Connector> connectors = destination.getConnectors();
    connectors.forEach(
            connector -> connectorAnalyzer.analyze(connector, destinationConnection, gh)
    );
    SQLProject project = destination.getSQLProject();
    if (project != null) {
        projectAnalyzer.analyze(project, destinationConnection, gh);
    }
}
```

Výpis kódu 7.10 Ukázka metody *analyze* třídy *DestinationAnalyzer*

Metodu *analyze* třídy *DestinationAnalyzer*, jejíž implementaci jsem umístil do ukázky 7.10, vytváří připojení k cílové databázi destinace a deleguje analýzu jejích jednotlivých částí na ostatní analyzátory. Připojení k cílové databázi je později využito při analýze SQL projektu a tabulek konektorů.

```
private void addDirectFlowBetweenColumns(List<Node> sourceColumns,
                                         Map<String, Node> dstColumns,
                                         ConnectorType connectorType,
                                         FivetranGraphHelper gh) {
    sourceColumns.forEach(node -> {
        String convertedName = NodeNameConverter.applyNamingConventions(
                node.getName(), connectorType
        );
        Node resultNode = dstColumns.get(convertedName);
        if (resultNode != null) {
            gh.addDirectFlow(node, resultNode);
        }
    });
}
```

Výpis kódu 7.11 Ukázka metody *addDirectFlowBetweenColumns* třídy *ConnectorTableAnalyzer*

Mnohem zajímavějším analyzátorem je třída *ConnectorTableAnalyzer*. Tato třída slouží pro zpracování datového modelu jedné tabulky konektoru a všech jejích sloupců. Jak jsem již zmínil v kapitole 6.3, soubory s metadaty neobsahují informace o sloupcích tabulky a z tohoto důvodu je neobsahuje ani datový model. Třída použije připojení ke zdrojové a cílové databázi vytvořená předchozími analyzátory k získání seznamu sloupců, které tabulka obsahuje. Jak bylo zmíněno v kapitole 3, při vytváření konektoru si můžeme vybrat, která schémata, tabulky a sloupce, chceme do cílové databáze integrovat. Z tohoto důvodu potřebujeme seznam sloupců tabulky z obou databází, abych je mohl porovnat.

Poté, co třída získá seznamy sloupců, vytvoří zdrojové a cílové uzly konektoru, které přidá do grafu datového toku. Posledním úkolem třídy je spojení vytvořených uzlů sloupců hranou. Toho třída docílí metodou *addDirectFlowBetweenColumns*, kterou můžeme pozorovat v ukázce 7.11. Tato metoda přijímá na vstupu list uzlů zdrojových sloupců, mapu uzlů cílových sloupců a jejich jmen, typ konektoru a třídu *FivetranGraphHelper*. Metoda aplikuje na jména uzlů zdrojových sloupců vhodnou jmennou konverzi a pokusí se zdrojovým uzlům najít jejich cílové protějšky. Pokud se to povede, tak metoda spojí uzly hranou. V opačném případě je zdrojový uzel ignorován.

Kapitola 8
Testování

V této kapitole představím, jakým způsobem byl prototyp testován a předložím i ukázky testů.

Veškerá implementace je testována pomocí jednotkových testů (anglicky „unit tests“) za použití nástrojů JUnit 4.13. a Mockito. Oba nástroje jsem představil v kapitole 2.

Prvním typem testů jsou takzvané parametrizované testy (anglicky „parameterized test“), které jsem použil k testování metod bez vedlejších účinků. To jsou metody, které neupravují stavové proměnné a pro každý vstup mají vždy stejný výstup. Příklad parametrizovaného testu jsem umístil do ukázky 8.2. Každý parametrizovaný test musí obsahovat právě jednu metodu s testem, označenou anotací *@Test*. Dále musí test obsahovat pole s hodnotami používaných v testovací metodě. Toto pole definuji pomocí anotace *@Parameterized.Parameters*. A v poslední řadě definuji parametry, které budou v poli, jež jsem zmínil dříve v tomto odstavci. Tyto parametry definuji anotací *@Parameterized.Parameter* [32]. V ukázce 8.2 testuji správnou funkčnost metody aplikující jmenné konverze na názvy zdrojových uzlů sloupců.

```
@Test
public void destinationAnalyzer_connectors() {
    dstAnalyzer.analyze(dstMock, ghMock);
    verify(connAnalyzerMock, times(2))
        .analyze(connectorMock, connectionMock, ghMock);
    verify(connAnalyzerMock, times(2))
        .analyze(any(), any(), any());
    }
```

Výpis kódu 8.1 Ukázka testu nástroje Mockito.

Do ukázky 8.1 jsem vložil jeden z testů, který kontroluje, jestli třída *DestinationAnalyzer* správně volá metody ostatních analyzátorů pro zpracování jednotlivých konektorů. Tento typ testů, používající nástroj *Mockito*, jsem využil při testování metod, jejichž výsledek je závislý na výsledcích metod jiných tříd.

```
@RunWith(Parameterized.class)
public static class SchemaConversionTest {
    @Parameterized.Parameter(0)
    public String name;
    @Parameterized.Parameter(1)
    public String result;

    @Parameterized.Parameters(name = "NodeNameConverter_schemaConversion")
    public static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][]{
                {"My_Name", "my_name"},
                {"My Name", "my_name"},
                {"MyName", "myname"},
                {"MyName!1", "myname_1"},
                {"1My!__Name", "1my___name"}
        });
    }

    @Test
    public void nodeNameConverter_schemaConversion() {
        parameterConsoleDescription(name);
        String result = NodeNameConversionDirector.applyNamingConventions(
            name, ConnectorType.POSTGRESQL
        );
        assertEquals(this.result, result);
    }
}
```

Výpis kódu 8.2 Ukázka parametrizovaného testu.

Kapitola 9
Závěr

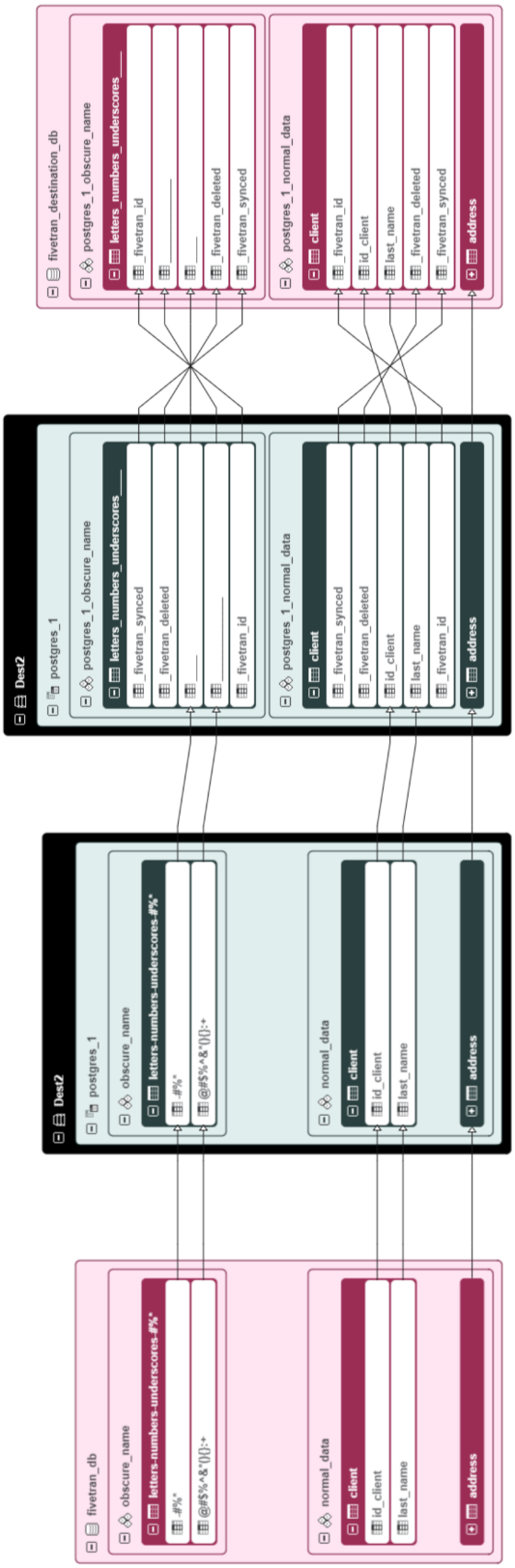
Jedním z cílů mé práce bylo analyzovat nástroje Looker a Fivetran z pohledu transformací dat. Fivetran je nástroj specializující se na integraci dat z různých datových zdrojů do centralizovaných úložišť, jejich čištění a přípravu na budoucí zpracování. Looker je nástroj, který analyzuje a vizualizuje data uložené v jednom úložišti.

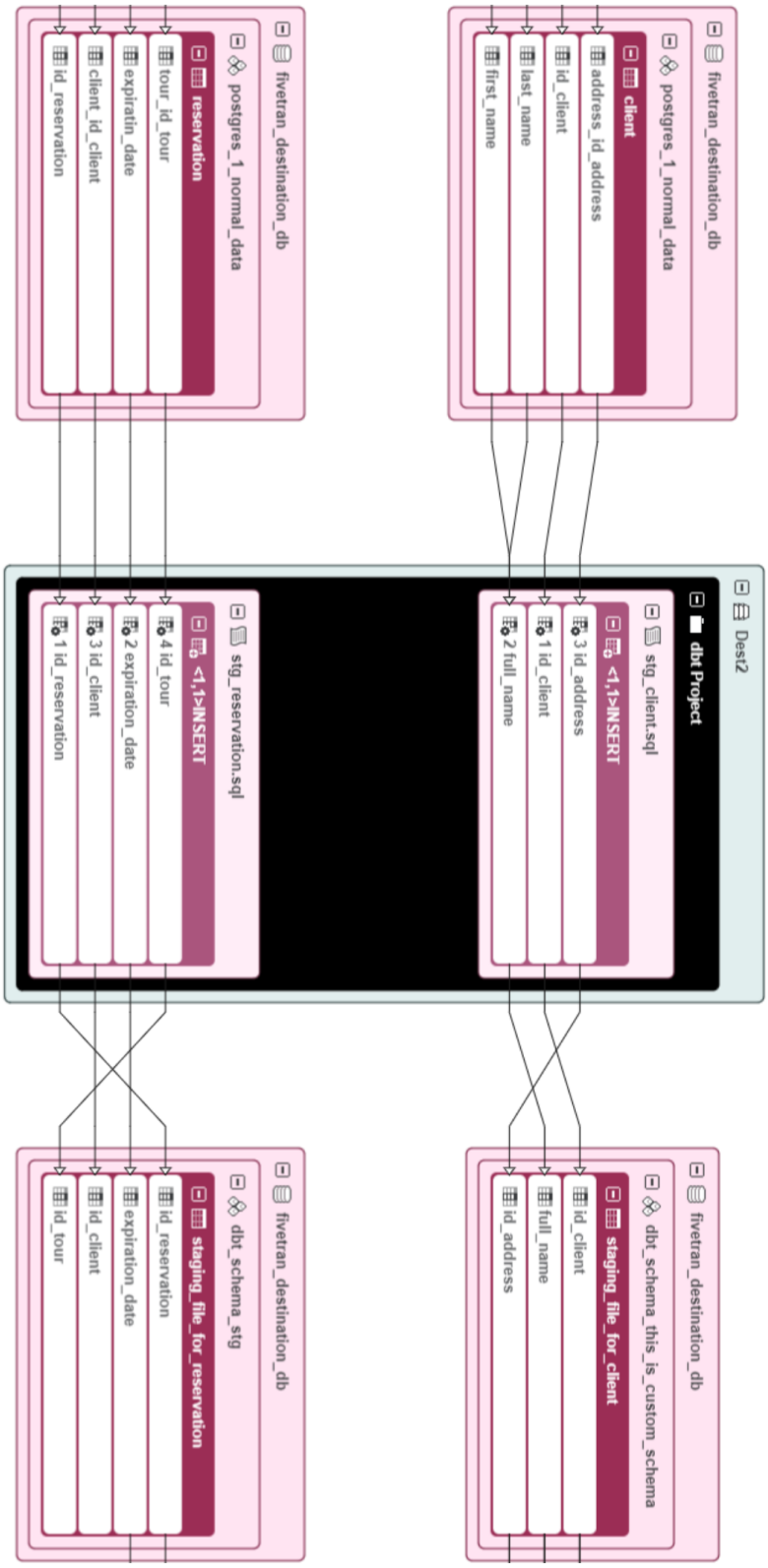
Druhým cílem mé práce bylo na základě analýzy těchto nástrojů si jeden vybrat a vytvořit prototyp, který zpracuje soubory s metadaty a vygeneruje graf datového toku nástroje. K implementaci prototypu jsem si vybral nástroj Fivetran. Prototyp se skládá ze dvou hlavních modulů zvaných Konektor a Dataflow Generátor. Konektor zpracovává metadata uložená ve struktuře adresářů, jejíž návrh a popis je součástí práce, stejně jako návrh datového modelu, ve kterém budou informace ze souborů uloženy. Generátor zpracuje model vytvořený modulem Konektor a vygeneruje graf datového toku. Prototyp je řádně otestován jednotkovými testy a podporuje pouze databázové konektory, jak bylo požadováno v cílech práce.

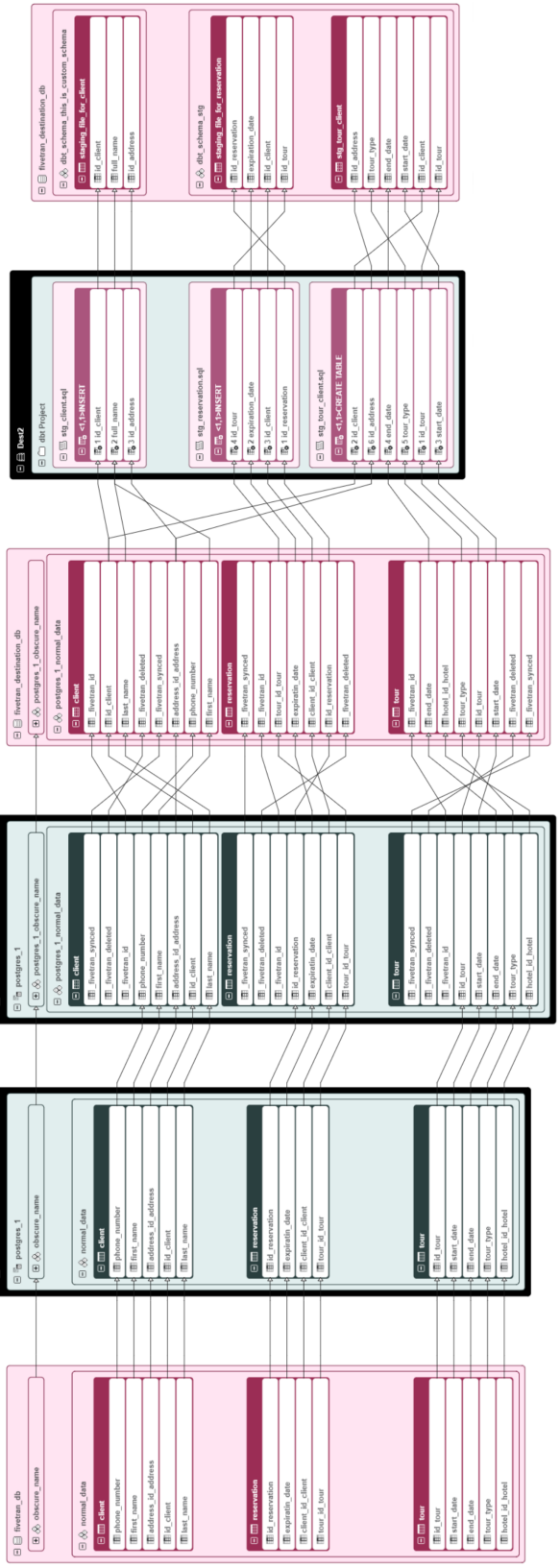
Posledním cílem mé práce bylo integrovat prototyp do nástroje Manta, který využije graf vygenerovaný prototypem k vizualizaci datového toku. To jsem učinil a v práci jsem uvedl několik výsledných vizualizací datového toku.

Příloha A

Ukázky vizualizace datového toku nástroje Fivetran


fivetran_db
obscure_name
letters-numbers-underscores-#%*
_#%*
@#$%^&*(){}:+
normal_data
client
id_client
last_name
address
Dest2
postgres_1
obscure_name
letters-numbers-underscores-#%*
_#%*
@#$%^&*(){}:+
normal_data
client
id_client
last_name
address
Dest2
postgres_1
postgres_1_obscure_name
letters_numbers_underscores___
_fivetran_synced
_fivetran_deleted
___
_____
_fivetran_id
postgres_1_normal_data
client
_fivetran_synced
_fivetran_deleted
id_client
last_name
_fivetran_id
address
fivetran_destination_db
postgres_1_obscure_name
letters_numbers_underscores___
_fivetran_id
_____
___
_____
_fivetran_deleted
_fivetran_synced
postgres_1_normal_data
client
_fivetran_id
id_client
last_name
_fivetran_deleted
_fivetran_synced
address


fivetran_destination_db
postgres_1_normal_data
client
address_id_address
id_client
last_name
first_name
fivetran_destination_db
postgres_1_normal_data
reservation
tour_id_tour
expiratin_date
client_id_client
id_reservation
Dest2
dbt Project
stg_client.sql
<1,1>INSERT
3 id_address
1 id_client
2 full_name
stg_reservation.sql
<1,1>INSERT
4 id_tour
2 expiration_date
3 id_client
1 id_reservation
fivetran_destination_db
dbt_schema_this_is_custom_schema
staging_file_for_client
id_client
full_name
id_address
fivetran_destination_db
dbt_schema_stg
staging_file_for_reservation
id_reservation
expiration_date
id_client
id_tour



Bibliografie

1. TECHOPEDIA. What is dataflow? Definition from Techopedia. In: *Techopedia.com* [online]. Techopedia, 2017 [cit. 2022-04-10]. Dostupné z: `https://www.techopedia.com/definition/6743/dataflow`.
2. JOSHI, Sagar. What Is Data Lineage? Why It's Important to Track Data Flow. In: *Learn Hub* [online]. Learn Hub, 2021 [cit. 2022-04-10]. Dostupné z: `https://learn.g2.com/data-lineage`.
3. IBM. What is ETL (extract, transform, load)? In: *IBM Cloud Learn Hub* [online]. IBM Cloud Education, 2020 [cit. 2022-04-10]. Dostupné z: `https://www.ibm.com/cloud/learn/etl`.
4. FRUHLINGER, Josh; PRATT, Mary. What is business intelligence? Transforming data into business insights. In: *CIO* [online]. CIO, 2019 [cit. 2022-04-10]. Dostupné z: `https://www.cio.com/article/272364/business-intelligence-definition-and-solutions.html`.
5. CANN, Alex. 7 Reasons Business Intelligence Is Vital To Business Success. In: *Maximizer CRM Software* [online]. Maximizer CRM, 2020 [cit. 2022-04-10]. Dostupné z: `https://www.maximizer.com/7-reasons-why-business-intelligence-is-vital-to-business-success/`.
6. KRANZ, Garry. Metadata. In: *WhatsIs.com* [online]. TechTarget, 2021 [cit. 2022-04-10]. Dostupné z: `https://www.techtarget.com/whatis/definition/metadata`.
7. BACA, M. *Introduction to Metadata: Third Edition*. J. Paul Getty Trust, 2016. Introduction To. ISBN 9781606064795. Dostupné také z: `https://books.google.cz/books?id=xgZVDQAAQBAJ`.
8. KIMBALL, R.; CASERTA, J. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. John Wiley & Sons, 2011. ISBN 9781118079683. Dostupné také z: `https://books.google.cz/books?id=TCLfzU2ilVkC`.
9. AUTOR NEZNÁMÝ. Fivetran. In: *Wikipedia* [online]. Wikimedia Foundation, 2022 [cit. 2022-04-10]. Dostupné z: `https://en.wikipedia.org/wiki/Fivetran`.
10. AUTOR NEZNÁMÝ. Transformations. In: [online]. Fivetran, 2022 [cit. 2022-04-10]. Dostupné z: `https://fivetran.com/docs/transformations`.
11. GROSS, Paige. Fishtown analytics renamed to DBT labs, and raised a $150M series C. In: *Technical.ly* [online]. Technically Media, 2021 [cit. 2022-04-10]. Dostupné z: `https://technical.ly/startups/dbt-labs-series-c/`.
12. TRISTAN, Handy. What, exactly, is DBT? In: *dbt blog* [online]. dbt blog, 2021 [cit. 2022-04-10]. Dostupné z: `https://blog.getdbt.com/what-exactly-is-dbt/`.

13. PARESH, Dave. Google to buy analytics software firm Looker for $2.6 billion. In: *Reuters* [online]. Thomson Reuters, 2019 [cit. 2022-04-10]. Dostupné z: https://www.reuters.com/article/us-looker-m-a-alphabet-idUSKCN1T71QF.
14. AAKASH, Raman. Understanding Looker ML: 5 Easy Steps to Learn. In: *Hevo* [online]. Hevo, 2021 [cit. 2022-04-10]. Dostupné z: https://hevodata.com/learn/understanding-looker-ml/.
15. AUTOR NEZNÁMÝ. Manta. In: *Wikipedia* [online]. Wikimedia Foundation, 2022 [cit. 2022-04-10]. Dostupné z: https://cs.wikipedia.org/wiki/MANTA_(firma).
16. AUTOR NEZNÁMÝ. How MANTA Works. In: *Manta Portál* [online]. 2022 [cit. 2022-04-10]. Dostupné z: https://getmanta.com/about-the-manta-platform/.
17. AUTOR NEZNÁMÝ. Supported technologies. In: *Manta Portál* [online]. 2022 [cit. 2022-04-10]. Dostupné z: https://getmanta.com/technologies/data-integration/.
18. SÝKORA, J. *Incremental update of data lineage storage in a graph database*. 2018. Dipl. pr. České vysoké učení technické v Praze, Fakulta informačních technologií.
19. KOŠVANEC, Petr. *Analýza datových toků v reportovacích nástrojích*. 2018. Dipl. pr. České vysoké učení technické v Praze, Fakulta informačních technologií.
20. IBM CLOUD EDUCATION. What is java? In: *IBM* [online]. IBM, 2019 [cit. 2022-04-10]. Dostupné z: https://www.ibm.com/cloud/learn/java-explained.
21. IBM CLOUD EDUCATION. Relational Databases. In: *IBM* [online]. IBM, 2019 [cit. 2022-04-10]. Dostupné z: https://www.ibm.com/cloud/learn/relational-databases.
22. AUTOR NEZNÁMÝ. SQL. In: *Wikipedia* [online]. Wikimedia Foundation, 2022 [cit. 2022-04-10]. Dostupné z: https://en.wikipedia.org/wiki/SQL.
23. HAMILTON, Thomas. Junit tutorial for beginners. In: *Guru99* [online]. Guru99, 2022 [cit. 2022-04-10]. Dostupné z: https://www.guru99.com/junit-tutorial.html.
24. LARS VOGEL, Fabian Pfaff. Unit tests with Mockito. In: *Vogella* [online]. vogella GmbH, 2021 [cit. 2022-04-10]. Dostupné z: https://www.vogella.com/tutorials/Mockito/article.html.
25. BRETT PORTER Jason Zyl, Olivier Lamy. Welcome to Apache Maven. In: [online]. Apache Maven, 2022 [cit. 2022-04-10]. Dostupné z: https://maven.apache.org/.
26. CRUSOVEANU, Loredana. Inversion of control and dependency injection with Spring. In: *Baeldung* [online]. Baeldung, 2022 [cit. 2022-04-10]. Dostupné z: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring.
27. YILDIRIM, Soner. What is Git and Why is It So Important? In: *Towards Data Science* [online]. Towards Data Science, 2020 [cit. 2022-04-10]. Dostupné z: https://towardsdatascience.com/what-is-git-and-why-is-it-so-important-dce559b27833.
28. IntelliJ IDEA overview. In: *JetBrains* [online]. JetBrains s.r.o, 2022 [cit. 2022-04-10]. Dostupné z: https://www.jetbrains.com/help/idea/discover-intellij-idea.html#developer-tools.
29. AUTOR NEZNÁMÝ. Documentation. In: *Fivetran* [online]. Fivetran, 2021 [cit. 2022-04-10]. Dostupné z: https://fivetran.com/docs/getting-started.
30. AUTOR NEZNÁMÝ. Get started. In: *dbt* [online]. dbt, 2022 [cit. 2022-04-10]. Dostupné z: https://docs.getdbt.com/.
31. AUTOR NEZNÁMÝ. Looker documentation. In: *Looker* [online]. Looker, 2022 [cit. 2022-04-10]. Dostupné z: https://docs.looker.com/.
32. VOGEL, Lars. Unit Testing with JUnit 4 - Tutorial. In: *Vogella* [online]. Vogella GmbH, 2021 [cit. 2022-04-10]. Dostupné z: https://www.vogella.com/tutorials/JUnit4/article.html.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `BP-Podrábský-Michal.pdf` ..... text práce ve formátu PDF