Uniwersytet Łódzki

Wydział Matematyki i Informatyki

Instytut informatyki

Andrzej Krupa

nr indeksu: 338689

Język wysokiego poziomu do tworzenia procesów ETL w Hurtowniach danych.

High-level language to create ETL process in data warehouse.

Praca magisterska

przygotowana w Zakładzie Katedry Informatyki Stosowanej

promotor: dr Jan Pustelnik

Łódź 2014

Spis treści

Wstęp **5**

1 Hurtownie danych **7**

1.1 Powody budowania hurtowni danych. 8

1.1.1 OLAP a OLTP 9

1.1.2 Wspomaganie decyzji 10

1.2 Architektura hurtowni danych 10

1.3 Projektowanie hurtowni danych. 12

1.4 Wielowymiarowy model danych 13

1.4.1 Schemat gwiazdy 14

1.4.2 Schemat płatka śniegu 16

2 Procesy zasilania hurtowni danych **19**

2.1 Ogólna koncepcja zasilania hurtowni 19

2.1.1 Ekstrakcja 19

2.1.2 Transformacja 20

2.1.3 Ładowanie 21

2.2 Analiza przykładowego procesu zasilania 22

2.2.1 Warstwa interfejsowa 24

2.2.2 Warstwa pośrednia 27

2.2.3 Warstwa docelowa 30

3 Teoria tworzenia języków wyskokiego poziomu. **33**

3.1 Gramatyka 33

3.1.1 Języki formalne 33

3.1.2 Gramatyka formalna 35

3.1.3 Klasyfikacja języków 36
3.1.4 Wyrażenia regularne 37
3.2 Język wysokiego poziomu 38
3.2.1 Analiza leksykalna 40
3.2.2 Analiza składniowa 41
3.2.3 Analiza semantyczna 42

4 Program **43**
4.1 Opis implementacji programu 43
4.2 Przykładowe działania programu. 49
4.2.1 Abstrakcyjny przykład schematu gwiazdy 49
4.2.2 Realizacja przykładu opisanego w rozdziale 2 52

Podsumowanie **55**

Spis rysunków **57**

Spis listingów **57**

Literatura **61**

Wstęp

Ojcem koncepcji hurtowni danych jest Bill Inmon. Napisał on ponad 40 książek związanych z tematyką hurtowni danych. Koncepcja ta tłumaczy, w jaki sposób wspomagać osoby zarządzające firmą lub korporacją w podejmowaniu działań strategicznych. Hurtownie danych odniosły sukces w rozwiązywaniu problemów biznesowych w zarządzaniu relacjami z klientem, w skrócie CRM *(ang. Customer Relationship Management)*. Projektowanie jak i tworzenie hurtowni danych jest procesem złożonym i kosztownym. Firmy podejmujące się inwestowania w rynek hurtowni danych są świadome, że głównym zadaniem zakupionego produktu nie jest generowanie zysków tylko dostarczanie wiarygodnych i rzetelnych informacji, na podstawie których możliwe będzie podejmowanie decyzji strategicznych. Jeżeli projekt hurtowni danych przechowuje niepoprawne dane, bądź nie jest odpowiednio przygotowany pod danego klienta, staje się dużą stratą dla firmy.[1]

Procesy zasilające hurtownie danych są elementem istotnym i koniecznym każdej hurtowni danych. Odpowiadają za cykliczne zasilanie hurtowni danymi pochodzącymi z różnych źródeł. Tworzenie poprawnych procesów jest konieczne do generowania raportów, które dostarczają rzetelnych i wiarygodnych informacji, na podstawie których mogą zostać podjęte strategiczne decyzje usprawniające funkcjonowanie firmy.

Głównym celem tej pracy dyplomowej jest stworzenie języka wysokiego poziomu, który będzie wspomagał tworzenie procesów zasilających hurtownie danych. Język wspomagający tworzenie takich procesów powinien być przede wszystkim intuicyjny i zrozumiały, powinien przyspieszać tworzenie procesów i tym samym zwiększać efektywność pracy programisty. Jednym z najważniejszych założeń owego języka jest generowanie:

- szablonu pobierającego dane (źródło),
- szablonu pgloader lub gotowego polecenia insert,
- kodu umożliwiającego utworzenie tabeli,
- kodu języka SQL zasilającego tabele.

Pierwszy rozdział pracy opisuje hurtownie danych, jej architekturę oraz przykłady wykorzystania hurtowni danych. Rozdział drugi opisuje, czym są procesy zasilania hurtowni danych oraz omawia przykładowe procesy zasilania, które zostały zrealizowane w ramach niniejszej pracy. Trzeci rozdział został w całości poświęcony teorii związanej z tworzeniem języków interpretowanych. Rozdział czwarty, to opis programu poparty przykładem. Wszystkie przykłady zamieszczone w niniejszej pracy zostały przetestowane w składni języka SQL, akceptowanego przez PostgreSQL 8.4.14, na systemie Ubuntu 10.04 LTS. Baza danych PostgreSQL została wybrana z następujących powodów:

- bezpłatne, dobre oprogramowanie do zastosowań komercyjnych,
- wykorzystywane obecnie w firmie, w której zdobywam doświadczenie zawodowe, pracując przy tworzeniu warstwy pośredniej hurtowni bazy danych.

Analizator składniowy i leksykalny został utworzony przy użyciu otwartego oprogramowania LEX i YACC, które w znaczący sposób ułatwiły pracę w pierwszych dwóch etapach tworzenia języków interpretowanych i kompilatorów.

Rozdział 1

Hurtownie danych

Definicję Hurtowni Danych *(ang. Data Warehouse)* przypisuje się Billowi Inmon'owi, który jako pierwszy opisał ją w 1992 roku. Zgodnie z tą definicją hurtownią danych jest baza danych mającą następujące cztery cechy [1].

- Zorientowaną na temat *(ang. Subject-oriented)* — dane są gromadzone w ściśle określony sposób, by możliwe było stworzenie czytelnego zestawienia danych. W hurtowni danych nie są przechowywane działania, czy operacje biznesowe. Hurtownia ograniczona się firmie do jednego działu lub wybranego obszaru (np. biznesowego). Taka hurtownia jest określana jako lokalna hurtownia danych lub tematyczna hurtownia danych *(ang. data mart)* stanowiąca podzbiór hurtowni danych.

- Nieulotność *(ang. Non-volatile)* — dane przechowywane w hurtowni danych nie są nigdy usuwane, ani modyfikowane. Dane przeznaczone są wyłącznie do odczytu w celu utworzenia raportu na podstawie zadanego zapytania SQL.

- Integracja *(ang. Intergrated)* — w hurtowni danych znajdują się informacje, przechowywanych przy wykorzystaniu dowolnych technologii. W związku z faktem, że dane pochodzą z całej firmy, musi wystąpić ujednolicenie typów danych.

- Zmienność w czasie *(ang. Time-Variant)* — musi zostać określone, co jaki okres czasu zostanie zapamiętany stan obecny w danej firmie.

1.1 Powody budowania hurtowni danych.

Uzasadnieniem budowania hurtowni danych może być:

- **Przetwarzanie analityczne danych** *(ang. On-Line Analytical Processing, OLAP)* – z danych zgromadzonych w hurtowni danych są tworzone zestawienia statystyczne, wykresy i raporty w różnych okresach czasowych.
- **Przeprowadzanie analizy danych bez ingerencji w operacyjną pracę systemów transakcyjnych** – analiza danych ze względu na bardzo dużą ich liczbę wymaga złożonych i czasochłonnych obliczeń. Dopuszczalne są zapytania kilkusekundowe, kilkuminutowe, mogą wystąpić nawet zapytania kilkudniowe. Zapytania te jednak nie mogą wpływać na pracę systemu obsługującego klienta w czasie rzeczywistym, w którym takie zadanie musi zostać zrealizowane w możliwie jak najkrótszym czasie.
- **Całościowy wgląd w dane firmy** – firmy bardzo często przechowują dane w różnych aplikacjach oraz na różnych środowiskach sprzętowych. Firma posiada głębszą wiedzę na temat zdarzeń, które miały miejsce w jej firmie, jeżeli ma możliwość zintegrowania danych. Na przykład osoba posiadająca sklep i warsztat samochodowy, która chciałaby wiedzieć ile zostało sprzedanych części i kto naprawiał samochód w jego warsztacie.
- **Dostęp do danych historycznych** – dzięki danym historycznym możliwe jest wykonywanie analiz, z których można wyciągnąć wnioski, przekładające się na realne korzyści dla firmy.
- **Ujednolicenie posiadanych informacji** – eliminuje tzw. problem wielu wersji prawdy firmy. Raporty w frimie opierają się na danych pochodzących z różnych sektorów, które mogą generować zyski bądź straty. Firma w jednym obszarze może prosperować bardzo dobrze, zaś w innym może generować duże straty, które mogą w późniejszym czasie stać się nawet przyczyną upadku tej firmy. Ujednolicenie informacji zebranych z różnych sektorów pozwoli na całościowy wgląd w obecny stan firmy.

- **Wspomaganie decyzji** *(ang. Decision Support, DS)* – wykonywanie analizy symulującej scenariusz biznesowy.

1.1.1 OLAP a OLTP

Przetwarzanie analityczne danych *(ang. On-Line Analytical Processing, OLAP)* i przetwarzanie transakcyjne *(ang. On-Line Transactional Processing, OLTP)* są to systemy optymalizowane pod kątem przetwarzania danych [1, 4].

System OLTP jest przeznaczony dla pracowników, komunikujących się z systemem bazodanowym w celu uzyskania określonych informacji np. sprawdzenie dostępnych miejsc na danym koncercie.

Podstawowymi cechami systemów OLTP są:

1. krótki czas realizacji bardzo dużej ilości zapytań, wykonywanych przez wielu użytkowników,
2. optymalizacja systemu bazodanowego pod kątem odczytu danych,
3. częste usuwanie lub modyfikacja pojedynczych rekordów w bazie danych,
4. aktualność danych przechowywanych w bazie.

System OLAP jest przeznaczony dla pracowników przygotowujących zestawienie danych, raportów dla kadry zarządzającej, jak również dla analityków, którzy na podstawie zadanych do hurtowni danych zapytań, mogą odkryć zależności występujące w firmie, a następnie wyciągnąć odpowiednie wnioski, które w ich opinii mogą dać firmie realny zysk.

Podstawowymi cechami systemów OLAP są:

1. wykonywanie małej ilości zapytań przez niewielką liczbę użytkowników na dużym obszarze danych,
2. cykliczne zasilane w ustalonych przedziałach czasowych,
3. brak konieczności aktualizacji danych w bazie w czasie rzeczywistym.

1.1.2 Wspomaganie decyzji

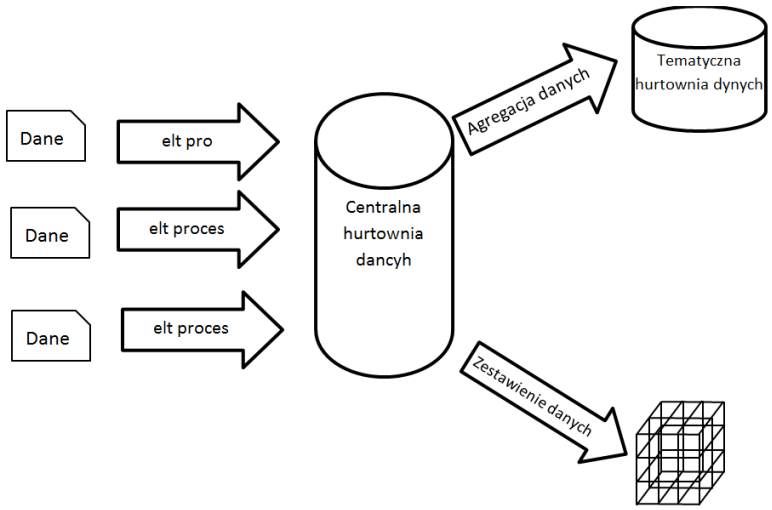
Systemy wspomagania decyzji *(ang. decision support systems)* tworzone są w celu minimalizacji kosztów prowadzonej działalności, lepszego przewidywania ryzyka podejmowanych działań, podniesienia jakości w dziale obsługi klienta. System OLAP jest jednym z takich narzędzi, które wspierają podejmowanie decyzji. Przykładowymi pytaniami, na które system powinien odpowiedzieć są [1]:

1. Jaki był dochód w rozbiciu na poszczególnych klientów?
2. Jaki był procentowy wzrost lub spadek dochodu w porównaniu z zeszłym miesiącem?
3. Jakie są cechy najlepszych/najgorszych klientów (cechy klientów muszą być ściśle określone)?
4. Listę klientów, dla których współczynnik odejścia jest wysoki, a przynoszą oni zysk firmie.

Hurtownie danych odniosły sukces związany z zarządzaniem relacjami z klientem *(ang. Customer Relationship Management, CRM)*, gdzie celem jest zatrzymanie najlepszych klientów oraz pozyskiwanie nowych, a także sprzedawanie im jak największej liczby produktów.

1.2 Architektura hurtowni danych

W niniejszym podrozdziale zostanie przestawiona podstawowa architektura hurtowni danych oraz proces związany z tworzeniem hurtowni, który jest bardzo drogi, czasochłonny, a żeby osiągnął sukces musi on być także ukierunkowany na klienta, czyli dostosowany do jego wymagań, które opierają się głównie na intuicji. Na rysunku 1.1 przedstawione zostały główne elementy hurtowni danych oraz kierunek przepływu danych. Strzałki pokazują kierunek przepływu danych [1, 6].



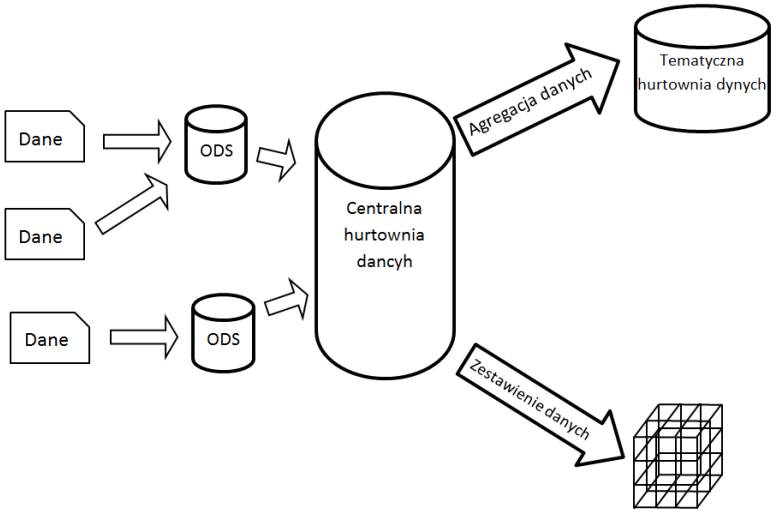
Rysunek 1.1: Architektura hurtowni danych.

Strukturę przepływu danych możemy podzielić na:

- **Źródło danych** ***(ang. source)*** — są to dane, które będą pobierane do hurtowni danych.
- **Proces ETL** ***(ang. extract, transfer, load)*** — procesem ETL nazywamy czynności wykonywane w celu pobrania danych źródłowych, przekształcenie danych na odpowiedni format, a następnie umieszczenie ich w centralnej hurtowni danych. Proces ETL zostanie dokładnie omówiony w rozdziale drugim.
- **Centralna hurtownia danych** ***(ang. center data warehouse)*** — jest to miejsce docelowe przetworzonych danych ze źródeł,
- **hurtownie tematyczne** ***(ang. data marts)*** — zawierają wybrane dane z centralnej hurtowni danych w sposób zagregowany, umożliwiające szybkie operowanie sporządzanie raportów,
- **zestawienie danych** — docelowym produktem hurtowni danych jest tworzenie odpowiednich zestawień danych. Na rysunku 1.1, został przedstawiona jako kostka.

Tworzenie hurtowni danych jest stosunkowo młodą dziedziną, która się dynamicznie rozwija. Dzięki zastosowaniom CRM hurtownie danych odniosły znaczący sukces,

co spowodowało większe zapotrzebowanie na przechowywanie i analizowanie danych historycznych. Przedstawiona architektura danych na rysunku 1.1, nie spełnia swojej roli jako hurtowni danych, w których przyrost danych jest bardzo duży. Poniżej została przedstawiona inna architektura danych.



Rysunek 1.2: Architektura hurtowni danych z magazynem danych ODS.

Do architektury z rysunku 1.1 został dodany magazyn danych operacyjnych *(ang. operational data store, ODS*), który pełni role magazynu danych. Ładowane są do niego dane pobrane ze źródeł i przetworzone w celu uzyskania zgodności typów danych. Kolejnym etapem jest załadowanie danych w sposób zagregowany do centralnej hurtowni danych.

1.3 Projektowanie hurtowni danych.

Projektowanie hurtowni danych tak jak relacyjnych baz danych polega na utworzeniu następujących modeli [1]:

- **Model pojęciowy** — przy użyciu języka biznesowego w danej firmie opisuje się cele biznesowe, które będzie można określić przez zgromadzenie ściśle określonych danych. Na modelu pojęciowym powinny, być zaznaczone nazwy kolumn, które mają być przechowywane w tabeli znajdującej się w hurtowni danych

- **Model logiczny** — jest to opis elementów logicznych hurtowni danych, wykonany np. w języku UML.
- **Model fizyczny** — jest to opis indeksowania, partycjonowania, opis sprzętu komputerowego, sieci rozmieszczenia poszczególnych zasobów fizycznych.

Najpopularniejszymi metodami przyjętymi podczas tworzenia hurtowni danych są:

- **Projektowanie wstępujące** (od szczegółu do ogółu)— polega na tworzeniu wszystkich etapów hurtowni danych jednocześnie, a następnie na integracji poszczególnych etapów ze sobą.
- **Projektowanie zstępujące** — dopóki jeden etap tworzenia hurtowni danych się nie skończy, następny nie może się rozpocząć. Jeżeli pojawią się błędy to wraca się do poprzedniego etapu i zaczyna się prace na kolejnym etapie od nowa.

1.4 Wielowymiarowy model danych

Wielowymiarowym modelem danych *(ang. Multidimensional Data Model*) nazywamy dane zorganizowane w:

- **Fakt** ***(ang. facts)*** — są to dane opisujące jakieś zdarzenie. Tabelę przechowującą te dane nazywamy *tablicą faktów*. Fakt opisywany jest przez wymiary i miary.
- **Wymiar** ***(ang. dimension)*** — jest jakąś cechą opisującą dany fakt. Cechy te znajdują się w tablicy wymiarów i są opisywane przez atrybuty.
- **Atrybut** *(ang. attribute)* — przechowuje dodatkowe informacje na temat wymiaru.
- **Miara** ***(ang. measures)*** — jest wartością mierzalną, przypisaną do pojedynczego rekordu w tablicy faktów.

Model ten jest zintegrowaną częścią systemu OLAP. Podstawowym atutem wielowymiarowego modelu danych jest proste zrozumienie hurtowni danych i poruszania się po niej w sposób efektywny, szybsze wykonywanie zapytań zadawanych do hurtowni danych. Jak również możliwość analizy danych w różnych wymiarach, które jest bardzo istotne ze względów biznesowych:

- Oglądanie informacji rozłożonych w czasie,
- Wyświetlanie informacji w sposób graficzny,
- Możliwość zmiany przekroju danych w dowolny sposób,
- Analizę danych pod kątem informacji istotnych dla danej firmy.

Podstawowymi schematami wielowymiarowego modelu danych są:

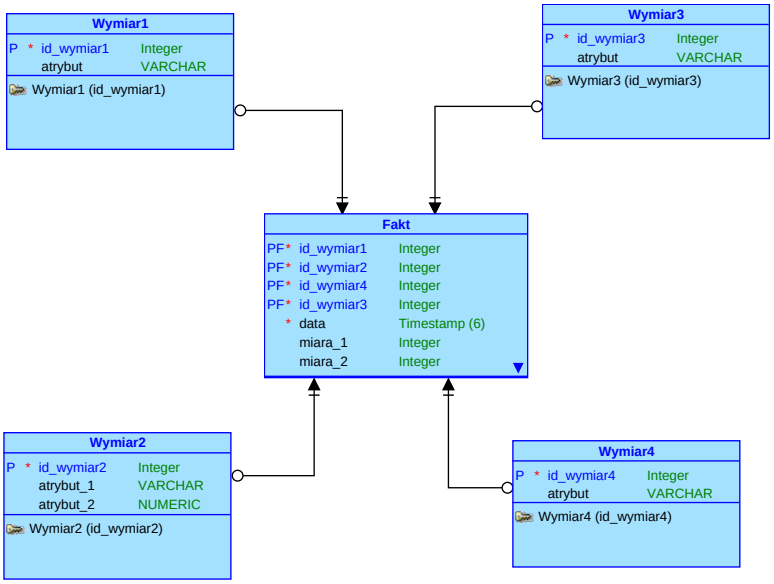
- schemat gwiazdy *(ang. Star schema)*
- schemat płatka śniegu *(ang. Snowflake schema)*

1.4.1 Schemat gwiazdy

Schemat gwiazdy jest podstawowym schematem wielowymiarowego modelu danych, w którym znajduje się jedna tabela faktów połączona z wieloma tabelami wymiarów. Tabela faktów w powyższym schemacie jest w trzeciej postaci normalnej, a tabela wymiarów jest w drugiej postaci normalnej. Dzięki takiej strukturze możliwe jest szybsze przeglądanie danych poprzez [1, 6]:

- poszczególne wymiary,
- sumowanie danych,
- agregację danych,
- filtrowanie danych

Na rysunku 1.3 został przedstawiona przykładowa architektura schematu gwiazdy.



Rysunek 1.3: Przykładowy schemat gwiazdy w postaci abstrakcyjnej.

Poniżej znajduje się listing zapytań do bazy danych w języku PostgreSQL, który realizuje model logiczny gwiazdy zawarty na rysunku 1.3

Listing 1.1: Listing kodu tworzący schemat gwiazdy.

```
1  DROP TABLE IF EXISTS fakt;
2  DROP TABLE IF EXISTS wymiar1;
3  DROP TABLE IF EXISTS wymiar2;
4  DROP TABLE IF EXISTS wymiar3;
5  DROP TABLE IF EXISTS wymiar4;
6  CREATE TABLE wymiar1
7  (
8    id_wymiar1 integer PRIMARY KEY
9  , atrybut varchar
10 );
11
12 CREATE TABLE wymiar2
13 (
14   id_wymiar2 integer PRIMARY KEY
15 , atrybut_1 varchar
16 , atrybut_2 numeric
17 );
18
19 CREATE TABLE wymiar3
```

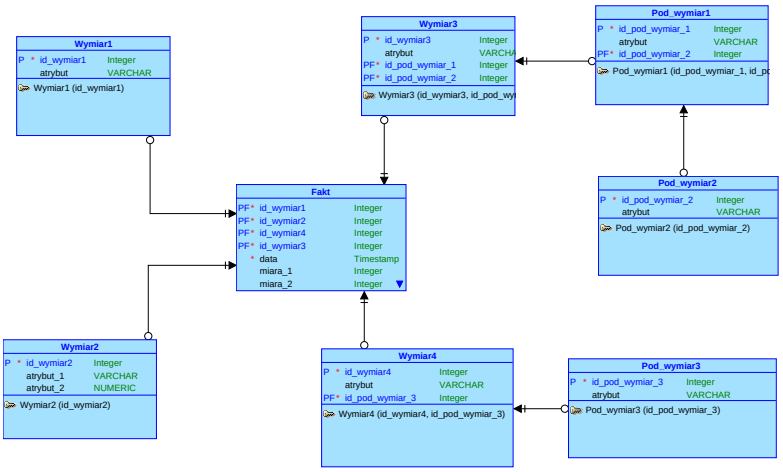
```
20 (
21   id_wymiar3 integer PRIMARY KEY
22 , atrybut varchar
23 );
24
25 CREATE TABLE wymiar4
26 (
27   id_wymiar4 integer PRIMARY KEY
28 , atrybut varchar
29 );
30
31 CREATE TABLE fakt
32 (
33   id_wymiar1 integer  references wymiar1(id_wymiar1)
34 , id_wymiar2 integer  references wymiar2(id_wymiar2)
35 , id_wymiar3 integer  references wymiar3(id_wymiar3)
36 , id_wymiar4 integer  references wymiar4(id_wymiar4)
37 , data       timestamp not null
38 , miara_1    integer  not null
39 , miara_2    integer  not null
40 );
```

1.4.2 Schemat płatka śniegu

Architektura schematu gwiazdy jest uproszczoną formą architektury płatka śniegu. Podstawową różnicą pomiędzy tymi schematami jest tabela wymiarów, która jest znormalizowana.

Schemat płatka śniegu jest stosowany wtedy, gdy tabela wymiarów osiąga duży rozmiar. Normalizuje się tabele wymiarów, aby zmniejszyć jej liczebność, dzięki czemu czas zapytań, powinien się znacząco skrócić. Wadą tego podejścia jest, że im bardziej znormalizowana jest tabela wymiarów, tym bardziej skomplikowane łączenia SQL muszę zostać użyte, aby pobrać odpowiednie dane z hurtowni danych. [1, 7]

Przykładowa architektura płatka śniegu został przedstawiona na rysunku 1.4 .



Rysunek 1.4: Przykładowy schemat płatka śniegu w postaci abstrakcyjnej.

Rozdział 2

Procesy zasilania hurtowni danych

2.1 Ogólna koncepcja zasilania hurtowni

Hurtownia danych jako system magazynujący dane i wspierający raportowanie stawia sobie jako jeden z głównych celów gromadzenie danych i takie ich przekształcanie, aby przyszłe raportowanie było jak najłatwiejsze w kontekście pytań biznesowych, jakie stawiają użytkownicy końcowi. Ogół procesów zasilania jest najczęściej określany skrótem ETL, pochodzącym od angielskich słów Extract, Transform, Load (ekstrakcja, transformacja, ładowanie), które oddają charakter procesów, oraz podsumowują cele, jakie są stawiane przed procesami zasilania hurtowni danych.

2.1.1 Ekstrakcja

Dane, które ostatecznie trafiają do hurtowni, pochodzą z różnych źródeł firmy lub spoza niej i różnią się sposobem dostępu. Źródłami mogą być systemy transakcyjne (transakcje bankowe, system płatności online, systemy obsługi klienta, zapisy partii szachowych online itp.), logi systemów (logi stron internetowych, systemów e-commerce, pliki z wykazem połączeń telefonicznych), publicznie dostępne pliki (dane giełdowe, wskaźniki i dane makroekonomiczne GUS czy nawet ręcznie generowane przez użytkowników biznesowych (arkusze kalkulacyjne, plany i cele sprzedaży). Różnorodność źródeł stawia po stronie hurtowni danych konieczność ekstrakcji danych z

formatu, w którym są dostępne, niezależnie od źródła i formatu (pliki tekstowe, bazy danych, arkusze kalkulacyjne, dane nieustrukturyzowane, obrazy itp.). Czasami trudność przedstawia samo znalezienie właściwego źródła danych, bądź znalezienie kilku źródeł, które razem zawierają potrzebne informacje, czasami najtrudniejsze jest wykonanie ekstrakcji (zwłaszcza, jeżeli mamy do czynienia z systemami, które nie są juz wspierane, a są nadal w użyciu). Mogą również pojawić się problemy wydajnościowe związane z transportem danych (np. danych jest na tyle dużo, że wąskim gardłem staje się przepustowość sieci i należy uciec się do kompresji danych jako do rozwiązania problemu). Jednym z problemów występujących przy ekstrakcji danych jest minimalizacja wpływu procesów ekstrakcji na funkcjonowanie źródła danych. Może okazać się, że próby ekstrakcji danych bezpośrednio z systemu źródłowego są tak wielkim obciążeniem wydajnościowym, że doprowadzają do nieakceptowalnych czasów działania systemu transakcyjnego. Do typowych rozwiązań należy harmonogramowanie procesów ETL w taki sposób, by ekstrakcja odbywała się np. w nocy, kiedy użytkowników systemu transakcyjnego jest mniej lub nie ma ich wcale, oraz korzystanie z kopii systemu źródłowego dla celów ekstrakcji (np. druga baza danych połączona z pierwszą w system „hot standby" lub po prostu zwykła kopia odtwarzania z codziennych backupów systemu transakcyjnego.

2.1.2 Transformacja

Po pobraniu danych, dane te muszą zostać przekształcone do wspólnej postaci. Wynika to z faktu, że różne źródła, nawet te podobne do siebie, przechowują dane w różnej postaci i zakładają różne zależności pomiędzy poszczególnymi elementami. Na przykład, bank powstały w wyniku fuzji kilku innych banków może korzystać z różnych systemów obsługi klienta. W jednym systemie kluczowi klienci biznesowi mogą mieć przypisanego jednego opiekuna i taka informacja jest dostępna w systemie źródłowym, modelowana jako relacja jeden do wielu (jeden opiekun dla wielu klientów), natomiast w drugim systemie może to być relacja wiele do wielu z archiwizacją historii przypisań opiekunów do klientów. Dane w systemach transakcyjnych bardzo często są zgodne z trzecią postacią normalną, natomiast hurtownia danych często przechowuje i prezentuje dane w postaci zdenormalizowanej, zwłaszcza jeśli do modelowania hurtowni został

wybrany schemat gwiazdy. Kontynuując przykład opiekuna klienta, model hurtowni danych może przewidywać opiekuna jako zwykły atrybut klienta, ignorując fakt, że w rzeczywistości relacja jest postaci jeden do wielu bądź wiele do wielu i tak jest zamodelowana w systemach źródłowych. Tego typu transformacje oraz zamiany kluczy z systemów źródłowych na własne klucze używane przez hurtownię (zazwyczaj zwane w środowisku hurtowni danych kluczami sztucznymi) są podstawowymi zadaniami procesów zasilania hurtowni. Do innych typowych przekształceń należy tworzenie wspólnych typów danych (np. ten sam atrybut może być opisywany w różnych systemach przez ciągi znaków różnej długości), ujednolicanie zawartości atrybutów (np. typ klienta może przybierać wartości „biznesowy", „business", 3, „firma" itp. w różnych systemach, w hurtowni chcemy przechowywać jedną wartość, wspólną dla wszystkich rekordów jednego typu), łączenie bądź rozdzielanie atrybutów (np. „małżeństwo z dziećmi" chcemy rozdzielić na dwa atrybuty, jeden opisujący „małżeństwo", drugi niosący informację, czy „ma dzieci"), łączenie atrybutów (np. rodzaj „firma", rozmiar „poniżej 200 pracowników" chcemy oznaczyć jako „SME"). Możliwe są również przekształcenia specyficzne dla danej dziedziny, np. wyliczanie stopy zwrotu czy procentowej zmiany cen z danych giełdowych.

2.1.3 Ładowanie

Przekształcone dane muszą trafić do docelowego modelu danych. Większość pracy została już wykonana na etapie ekstrakcji i transformacji, jednak pozostaje zadbać o spójność danych (czy podczas ładowania hurtownia pozwoli na dostęp do transakcji dla nowych klientów, których jeszcze nie zdążyliśmy załadować?). Często tego typu kwestie są rozwiązywane w zupełnie inny sposób niż w systemach transakcyjnych, w których spójność danych jest zapewniana mechanizmami bazodanowymi typu transakcje czy więzy integralności. Przenoszenie takich rozwiązań do hurtowni danych stwarza potencjał dla problemów wydajnościowych zależnych bądź nie od implementacji konkretnego systemu zarządzania bazami danych – np. więzy integralności spowalniają ładowanie, gdyż są sprawdzane wiersz po wierszu, a transakcje czasami wręcz nie są możliwe do użycia z uwagi na ograniczenia techniczne przy dużej ilości danych przetwarzanych w hurtowniach. Przykładowo, w systemie zarządzania bazą danych Oracle,

transakcje standardowo generują UNDO i REDO, więc zawarcie całości ładowania w jednej transakcji, nawet gdyby było technicznie możliwe (bazę można skonfigurować, aby udostępniała wystarczająco dużo miejsca na UNDO i REDO), stwarzałoby olbrzymie problemy wydajnościowe z uwagi na kilkukrotne zwiększenie ilości operacji wejścia/wyjścia (należy pamiętać, że UNDO też jest chronione przez REDO, więc ilość operacji dyskowych może wzrosnąć nawet czterokrotnie). Typową „sztuczką" praktyczną bywa np. wyłączenie generowania REDO na poziomie bazy danych. Jeśli chodzi o UNDO, to typowym rozwiązaniem jest podzielenie transakcji na mniejsze części, co redukuje ilość UNDO, które musi być przechowywane (każde zatwierdzenie transakcji pozwala na pozbycie się UNDO, które zostało wygenerowane przez daną transakcję), ale to już oznacza, że zapewnienie spójności musi leżeć po stronie procesów zasilania. Na szczęście zasilanie hurtowni danych jest procesem wsadowym, który w razie czego może zostać powtórzony, więc większość tradycyjnych mechanizmów bazodanowych zapewniających ochronę transakcji nie jest potrzebna. Zapewnienie spójności danych jest realizowane w samych procesach ładowania, chociażby przy ustalaniu kolejności zasileń. Przykładowo, ładując hurtownię zbudowaną w oparciu o schemat gwiazdy, tradycyjnie ładuje się najpierw wymiary, a potem fakty. Pomimo że nie zapewnia to spójności w najściślejszym znaczeniu tego pojęcia (mogą pojawiać się wiersze w tabelach wymiarów, które nie mają swoich odpowiedników w tabelach faktowych), to w praktyce jest to spójność, jakiej oczekują użytkownicy i personel utrzymujący hurtownie danych. Wynika to z faktu, że w modelu gwiazdy wymiary są używane do interpretacji danych faktowych, a więc ich znaczenie jest drugorzędne. Najbardziej istotne jest, aby wszystkie dane faktowe dostępne dla użytkownika były opisane przez wymiary, więc ładowanie wymiarów w pierwszej kolejności zapewnia ten stan rzeczy.

2.2 Analiza przykładowego procesu zasilania

Dla zilustrowania koncepcji języka do budowy hurtowni danych oraz jego praktycznego zastosowania, zostanie zbudowana przykładowa, uproszczona hurtownia danych wraz z procesami zasilania. Tematyką hurtowni będą dane giełdowe, a konkretnie notowania ciągłe z warszawskiej Giełdy Papierów Wartościowych (GPW). Wybór ten jest umotywowany powszechną dostępnością danych — każdy może sobie w dowolnej

chwili pobrać publicznie dostępne dane z wielu różnych stron internetowych. Nie bez znaczenia jest również prostota danych, które są intuicyjnie zrozumiałe dla większości osób i nie będą wymagały wyjaśniania.

Do zbudowania hurtowni danych zostanie użyty schemat gwiazdy, w którym występować będzie jedna tabela faktów, zaprezentowana na listingu 2.1, wraz z towarzyszącym jej wymiarem pokazanym na listingu 2.2.

Listing 2.1: Kod tworzący tabelę faktów.

```
drop table if exists public.gpw;
create table public.gpw
(
  npw_id integer
, data_notowania date
, otwarcie decimal(20, 2)
, max  decimal(20, 2)
, min decimal(20, 2)
, zamkniecie decimal(20, 2)
, wartosc decimal(20,3)
);
```

Listing 2.2: Kod tworzący tabelę wymiaru.

```
drop table if exists npw;
drop sequence if exists npw_kmap_seq;
create sequence  npw_kmap_seq
  increment by 1
  no minvalue
  no maxvalue
  start with 1
  cache 1
  cycle;

create table npw
(
  npw_id integer default nextval('npw_kmap_seq')
, nazwa varchar(50)
);
```

Docelowe rozwiązanie będzie składało się z trzech warstw:

- warstwy interfejsowej,

- warstwy pośredniej,
- warstwy docelowej,

które zostaną omówione w kolejnych podrozdziałach.

2.2.1 Warstwa interfejsowa

Warstwa interfejsowa będzie służyła do komunikacji ze światem zewnętrznym celem pobrania danych do hurtowni. Przed procesami zasilania będą postawione następujące zadania szczegółowe:

1. Wykrywanie nowych danych w systemie źródłowym
2. Pobieranie danych z systemu źródłowego
3. Ładowanie plików do bazy danych

Dane o cenach akcji GPW są publicznie dostępne w internecie i są już wyekstraktowane w postaci gotowych do ściągnięcia plików tekstowych. Zatem nie jest konieczne, aby procesy zasilania wykonywały ekstrakcję ze źródła (jakiegoś systemu operacyjnego). Zamiast tego, przydatne będzie wykrywanie, czy od ostatniego wykonania procesów zasilania pojawiły się nowe pliki z danymi i pobranie wyłącznie nowych danych.

Cel ten zostanie zrealizowany za pomocą skryptów powłoki systemu UNIX, (przedstawiony na listingu) oraz podstawowych narzędzi systemowych dostępnych z poziomu systemu operacyjnego.

Listing 2.3: Skrypt pobierający dane serwera bossa.pl .

```
1 # skrypt do ściągania danych giełdowych z GPW
2 # dane pochodzą ze strony bossa.pl/notowania/metastock
3 #(notowania ciągłe , dane bieżące, plik sesjacgl.prn)
4
5 if [ ! -f sesjacgl.prn ]
6 then
7  touch sesjacgl.prn
8  md5sum sesjacgl.prn > md5file.md5
9 fi
```

```
10
11  if [ ! -f md5file.md5 ]
12  then
13   md5sum sesjacgl.prn > md5file.md5
14  fi
15
16  t=$(date '+%s')
17  while [[ $(($(date '+%s') - $t)) -le 10800 ]]
18  do
19    # ściągamy plik jeśli jest nowszy niż uprzednio ściągnięty
20    wget http://bossa.pl/pub/ciagle/mstock/sesjacgl/sesjacgl.prn -N 2>/dev/null
21    sleep 5s
22    # Jeżeli suma kontrolna się zmieni, to znaczy, że plik pod w/w adresem się
        zmienił
23    # i zostanie wykonany warunek
24    # Jeżeli w zmiennej $? będzi 1, to plik został zmienony
25    md5sum -c md5file.md5
26    wynik=$?
27    echo $wynik
28    if [  $wynik -eq 1 ]
29    then
30      echo "------ Plik jest nowy ----------"
31      # Tworzony jest plik sesja.txt, który jest wykorzystywany do ładowania danych
        do bazy.
32      md5sum sesjacgl.prn > md5file.md5
33      cp ./sesjacgl.prn ./copySesjacgl.prn
34      tar -zcvf  ./arch_gpw/arch$(date '+%Y_%m_%d_%H%M%S').tar.gz sesjacgl.prn
35      cp  sesjacgl.prn sesja.txt
36      break
37    fi
38  done
```

Procesy ETL muszą pobrać dane i przygotować do ładowania. Są to wszystkie czynności, które programiści uznają za potrzebne w celu przygotowania danych do poprawnego załadowania danych do tabelki interfejsowej. Czynnościami tymi może być chociażby rozpakowanie danych i rozmieszenie ich w odpowiednich katalogach. W naszym rozważanym przykładzie pobieramy plik o nazwie *sesjacgl.prn*, który jest plikiem tekstowym.

Pliki muszą zostać załadowane do bazy danych w niezmienionej formie, celem ich udostępnienia do dalszych przekształceń. Trafiają one do tabelki interfejsowej, która ma typ zmiennych zgodny z pobranymi danymi źródłowymi. Tabelkę te nazywać będziemy *intf_pgw*, a ma ona strukturę zaprezentowaną na listingu 2.4 .

Listing 2.4: Kod tworzący tabelkę interfejsową.

```
drop table if exists intf_gpw;
create table intf_gpw
(
  nazwa varchar(50)
, data_notowania date
, otwarcie decimal(20, 2)
, max decimal(20, 2)
, min decimal(20, 2)
, zamkniecie decimal(20, 2)
, wartosc decimal(20,3)
);
```

Dla umożliwienia pełnej audytowalności procesów, konieczne jest przechowywanie ściągniętych plików przynajmniej przez jakiś czas. W razie wystąpienia wątpliwości odnośnie jakości danych, będzie możliwość weryfikacji danych i porównania hurtowni z danymi źródłowymi. Często w hurtowniach danych istnieje dedykowana warstwa służąca tylko temu celowi. W rozwiązaniu zbudowanym na potrzeby niniejszej pracy, archiwizacja będzie odbywać się za pomocą kompresji plików i przeniesienia ich do dedykowanego katalogu, co zostało pokazane na listingu 2.3.

Jeśli w komercyjnych rozwiązaniach, archiwizacja danych odbywa się za pomocą plików, najczęściej stosowane są specjalistyczne narzędzia do tworzenia kopii zapasowych.

W programie stworzonym na potrzeby tej pracy dyplomowej zostało wykorzystane narzędzie pgloader (stworzone przez Dimitri Fontaine'a), służące do załadowania danych do bazy. Narzędzie pgloader dobrze współpracuje z systemem zarządzania bazą danych PostgreSQL. Najważniejszą zaletą tego narzędzia jest ładowanie danych w sposób równoległy, co pozwala na wykorzystanie większej mocy obliczeniowej maszyny, poprzez uruchomienie wielu wątków. Na listingu 2.5 został pokazany skrypt pgloader'a, który realizuje to zadanie.

Listing 2.5: Skrypt pgloader'a ładujący dane do tabelki interfejsowej.

```
[pgsql]
base=dwh
host=localhost
user=etl
```

```
5  port=5432
6  pass=etl
7  log_mis_messages=INFO
8  client_min_messages=WARNING
9  pg_option_client_encoding='win-1250'
10 pg_option_standard_conforming_strings=on
11 pg_option_work_mem=128MB
12 copy_every=15000
13 empty_string="\ "
14 max_parallel_sections=4
15 null=NULL
16 [gpw]
17 table=intf_gpw
18 format=csv
19 datestyle=ymd
20 field_size_limit=512kB
21 field_sep=,
22 quotechar="
23 columns=*
24 skip_header_lines=0
25 truncate=True
26 filename=sesja.txt
27 reject_log=sesja.reject_log
28 reject_data=sesja.reject_data
```

2.2.2 Warstwa pośrednia

Celem warstwy pośredniej jest przekształcenie danych z formatu źródłowego, które znajdują się w tabeli interfejsowej, czyli w tabeli *intf_gpw*, pokazanej na listingu 2.4, na format umożliwiający załadowanie danych do tabeli docelowej. Szczegółowe cele zależą najczęściej od konkretnego rozwiązania i jego architektury, a także od ładowanych danych. W przykładzie stworzonym na potrzeby tej pracy będą to:

1. Usuwanie pobranych duplikatów.
2. Zasilanie tabel przejściowych wymiarów. W naszym przykładzie, dla uproszczenia będzie zasilana tabela wymiarów pokazana na listingu 2.2,
3. Zamiana klucza naturalnego, którym jest nazwa papieru wartościowego na wartość *integer*, nadawaną przy użyciu sekwencji dla każdej nowej nazwy pojawiającej się w tabeli.

Aby osiągnąć cel wymieniony w podpunkcie 1, musi zostać utworzona tabela o identycznej strukturze, co tabela *intf_gpw*, pokazana na listingu 2.4. Kod realizujący owe zadanie został przedstawiony na listingu 2.6

Listing 2.6: Usuwanie pobranych duplikatów.

```
insert into stg_gpw
(
  nazwa
, data_notowania
, otwarcie
, max
, min
, zamkniecie
, wartosc
)
select
  nazwa
, data_notowania
, otwarcie
, max
, min
, zamkniecie
, wartosc
from intf_gpw i
where not exists
(
  select 1
  from stg_gpw s
  where s.nazwa = i.nazwa
    and s.data_notowania = i.data_notowania
);
```

W podpunkcie 2, wspomniano, że dla uproszczenia przykładu do tabeli wymiarów ładowana jest tylko nazwa tabeli, więc z tego powodu w tym miejscu zasilamy tabele wymiarów nazw papierów wartościowych, a realizujemy to w sposób zaprezentowany na listingu 2.7. W przypadku gdyby przykład nie został uproszczony, to tabelka ta powinna zostać zasilona w warstwie docelowej opisanej na stronie 30, w podrozdziale 2.2.3.

Listing 2.7: Ładowanie danych do tabeli wymiaru – papierów wartościowych.

```
insert into npw
(
```

```
3     nazwa
4   )
5   select distinct
6     nazwa
7   from stg_gpw s
8   where not exists
9   (
10    select 1
11    from npw  k
12    where k.nazwa = s.nazwa
13  );
```

Kolejnym etapem jest nadawanie kluczy sztucznych w hurtowni, poprzez zastąpienie klucza naturalnego (nazwy papierów wartościowych). W omawianej warstwie dokonuje się również łączenia danych pochodzących z różnych źródeł. Dane, które będą zasilać tabelę faktów w naszej hurtowni danych pochodzą z jednego źródła, z tego powodu tabela przejściowa będzie miała bardzo podobną strukturę do tabel opisanych poprzednio. Tabelę tę będziemy nazywać *promo_gpw*. Zaprezentowano ją na listingu 2.8

Listing 2.8: Struktura tabelki promo_gpw.

```
1   drop table if exists promo_gpw;
2   create table  promo_gpw
3   (
4     npw_id integer
5   , data_notowania date
6   , otwarcie decimal(20, 2)
7   , max decimal(20, 2)
8   , min decimal(20, 2)
9   , zamkniecie decimal(20, 2)
10  , wartosc decimal(20,3)
11  );
```

Wykonanie zadania z podpunktu 3, ze strony 27, zostało zaprezentowane na listingu 2.9.

Listing 2.9: Proces ładowania do tabeli promo_gpw.

```
1   truncate table promo_gpw;
2   insert into  promo_gpw
3   (
4     npw_id
```

```
5  , data_notowania
6  , otwarcie
7  , max
8  , min
9  , zamkniecie
10 , wartosc
11 )
12 select
13   n.npw_id
14 , s.data_notowania
15 , s.otwarcie
16 , s.max
17 , s.min
18 , s.zamkniecie
19 , s.wartosc
20 from stg_gpw s
21 join npw n  on n.nazwa=s.nazwa
22 ;
```

2.2.3 Warstwa docelowa

Celem warstwy docelowej jest załadowanie danych do tabel faktów i wymiarów, jak również udostępnienie ich dla użytkowników korzystających z hurtowni danych.

Dane, które będą zasilać tabelę faktową w naszej hurtowni są dziennymi danymi podsumowującymi cały dzień notowań ciągłych na giełdzie. Tego typu dane, z uwagi na swój charakter, nie zmieniają się po załadowaniu, dlatego zostanie pominięty UPDATE danych. Wykona jedynie zostanie operacja INSERT z danych przygotowanych w tabel *promo_gpw*, do tabeli faktów *gpw*, co zostało pokazane na listingu 2.10

Listing 2.10: Proces ładowania danych do tabeli gpw.

```
1  insert into  gpw
2  (
3    npw_id
4  , data_notowania
5  , otwarcie
6  , max
7  , min
8  , zamkniecie
9  , wartosc
10 )
11 select
```

```
12      npw_id
13    , data_notowania
14    , otwarcie
15    , max
16    , min
17    , zamkniecie
18    , wartosc
19  from promo_gpw p
20  where not exists
21  (
22    select
23      1
24    from gpw t
25    where
26          t.npw_id=p.npw_id
27    and   t.data_notowania=p.data_notowania
28  );
```

Rozdział 3

Teoria tworzenia języków wyskokiego poziomu.

3.1 Gramatyka

3.1.1 Języki formalne

Alfabet lub *słownik* oznaczają dowolny niepusty, skończony zbiór symboli. *Słowem* nazywamy ciąg symboli *alfabetu* o skończonej długości. Jeżeli słowo jest długości zero, to nazywamy go *słowem* pustym, które będziemy oznaczać przez małą literę grecką epsilon (ϵ). Synonimami *słowa* są *napis* i *zdanie*. [5]

Przykładami *alfabetu* mogą być:

- zbiór niektórych liter alfabetu polskiego,
- zbiór składający się z symbolu zera i jedynki,
- zbiór liczb całkowitych i zbiór symboli kodowania znaków UTF-8,
- zbiór { AA, BB }, w którym AA i BB są traktowane jako jeden symbol.

Przykładami *słów* dla *alfabetu* liczb całkowitych, który składa się ze znaków {+,-,.(kropka),0,1,2,3,4,5,6,7,8,9} mogą być: ϵ, 0, 1, 01, 10, 090, -1001, +098, -121, 100, +41, +0000010, -000011 itd.

Językiem formalnym (językiem) nazywamy podzbiór zbioru wszystkich słów nad skończonym alfabetem.

Przykładami języków formalnych mogą być:

- zbiór pusty, oznaczany jako Ø,
- zbiór zawierający tylko słowo puste $\{\epsilon\}$
- zbiór programów, które po skompilowaniu i uruchomieniu zawieszą dany komputer,
- zbiór wszystkich poprawnie napisanych nierówności.

W tabeli 3.1 zostały zdefiniowane prawa na językach.

Tabela 3.1: Definicja operacji na językach.

Termin	**Definicja**
suma L i M zapisywana $L \bigcup M$	$L \bigcup M = \{s : \quad s \in L \quad lub \quad s \in M\}$
złączenie L i M zapisywane LM	$LM = \{st : \quad s \in L \quad oraz \quad t \in M\}$
podnoszenie do potęgi	$L^0 = \{\epsilon\}$ $L^i = L^{i-1}L$
domknięcie L zapisywane $L*$	$L* = \bigcup_{i=0}^{+\inf} L^i$ $L*$ oznacza "zero lub więcej złączeń" L

Tabela 3.1 – Kontynuacja tabeli z poprzedniej strony.

Termin	**Definicja**
dodatnie domknięcie L zapisywane $L+$	$L+ = \bigcup_{i=1}^{+\inf} L^i$ $L*$ oznacza "co najmniej jedno złączenie" L

Rozważmy przykład. Niech L będzie zbiorem małych i dużych liter, a M zbiorem cyfr. Ponieważ symbole mogą być traktowane jako słowa o długości jeden, to zbiory L i M są językami skończonymi. Poniżej znajduje się kilka przykładów nowych języków utworzonych za pomocą L i M przy zastosowaniu operatorów zdefiniowanych w tabeli 3.1.

1. $L \bigcup M$ jest zbiorem liter i cyfr.
2. ML jest zbiorem słów składających się z cyfry i występującej po niej litery.
3. $L*$ jest zbiorem wszystkich słów złożonych z liter, włączając w to słowo puste.
4. $L+$ jest zbiorem wszystkich słów złożonych z liter, bez słowa pustego.
5. $L(L \bigcup M)*$ jest zbiorem wszystkich słów złożonych z liter i cyfr, zaczynających się od litery.

3.1.2 Gramatyka formalna

Języki formalne opisywane są przez *gramatyki formalne*, to jest uporządkowane czwórki (T,N,P,S), gdzie[5, 11]:

- T jest skończonym zbiorem symboli terminalnych (inaczej alfabetem),
- N jest skończonym zbiorem symboli nieterminalnych, przy czym, N$\bigcap$T=Ø,
- P jest skończonym zbiorem reguł produkcji postaci R_1: R_2;, gdzie R_1 i R_2, to symbole które reprezentują ciągi, o skończonej długości, *symboli terminalnych* i *symboli nieterminalnych*, przy czym, symbol R_1 musi zawierać co najmniej jeden symbol nieterminalny.

- S jest symbolem startowym i należy do zbioru symboli nieterminalnych. Od symbolu startowego zaczyna się wyprowadzanie wszystkich słów danego *języka formalnego.*

Rozpatrzmy przykład gramatyki G, która opisuje język akceptujący słowa postaci $\{ ({}^{n})^{n} : n \in \mathbb{N} \}$, Gramatyka G ma postać:

G = ({ (,) } , {S} , {S:(S), S: ϵ } ,S)

Słowo ((())) możemy wyprowadzić: S: (S) : ((S)) : (((S))) : ((()))

Do tak opisanego języka należy każde słowo, dla którego możliwe jest wyprowadzenie (utworzenie), przy użyciu reguł produkcji. Jeżeli nie jest możliwe wyprowadzenie słowa to nie należy do języka.

3.1.3 Klasyfikacja języków

Avram Noam Chomsky badał języki formalne, czyli podzbiór wszystkich słów nad skończonym alfabetem wyniku tych badań w 1956r. podał klasyfikację języków formalnych, która powszechnie uznawana jest za standard.

Hierarchia ta składa się z czterech klas [5, 9, 10]:

- języki typu 3 - regularne, są to języki opisywane za pomocą gramatyki regularnej, w której reguły produkcji mogą mieć postać:
  - N:TN
  - N:T
  - N:N
  - N:ϵ

  gdzie N jest symbolem terminalnym, a T symbolem nieterminalnym.
- języka typu 2 - bezkontekstowe, są to języki opisane za pomocą gramatyk bezkontekstowych, w której reguły produkcji mogą mieć postać:

– N:C

gdzie N jest symbolem nieterminalnym, a C to symbole które reprezentują ciągi, o skończonej długości, symboli terminalnych i symboli nieterminalnych,

- języka typu 1 - kontekstowe, opisywany jest przez gramatykę kontekstową, w której lewa strona produkcji nie może zawierać mniej symboli terminalnych i nieterminalnych niż prawa strona,
- języka typu 0 - rekurencyjnie przeliczalne, opisywany przez gramatykę rekurencyjnie przeliczalną, w której reguły produkcji nie zostały ograniczone.

Mówimy, że język należy do danej klasy wtedy, gdy jest możliwe zbudowanie gramatyki, która generuje dany język, a reguły produkcji nie wykraczają poza ograniczenia dla danej klasy.

3.1.4 Wyrażenia regularne

Wyrażenie regularne nad alfabetem Σ nazywamy ciąg znaków ϵ,), (, *, + oraz symboli z alfabetu Σ następującej postaci:

1. ϵ (słowo puste) jest wyrażeniem regularnym,
2. wszystkie symbole należące do alfabetu są wyrażeniami regularnymi,
3. niech r i s będą wyrażeniami regularnymi, to są nimi również:
   - $r|s$ (suma),
   - rs (łączność) ,
   - $r*$ (domknięcie),
   - $r+$ (dodatnie domknięcie)
   - (r) (grupowanie).
4. wszystkie wyrażenia regularne są postaci opisanej w punkcie 1 – 3.

Wyrażenie regularne r służy do opisywania języka regularnego, który będziemy oznaczać $L(r)$. Język opisywany przez wyrażenie regularne ma następującą postać:

- $\mathrm{L}(\epsilon) = \{\epsilon\}$
- $\mathrm{L}(a) = \{a\}$, gdzie a jest dowolnym symbolem z alfabetu Σ

Załóżmy, że r i s jest wyrażeniami regularnymi oznaczającymi języki $L(r) = M$ i $L(s) = L$, wtedy

- $r|s = M \bigcup L$
- $rs = ML$
- $r* = M*$
- $r* = M+$

Operatory na językach zostały opisane w tabeli 3.1, na stronie 34. Rozważmy przykłady. Niech alfabet Σ będzie zbiór liter języka polskiego oraz cyfr i znaków matematycznych:

1. wyrażenie regularne a|b oznacza zbiór {a,b },
2. wyrażenie regularne (a|b)* oznacza zbiór { ϵ, a, b, aa, ab, bb, ba, aaa, ... },
3. wyrażenie regularne (a|b)|(a|b) oznacza zbiór {aa,ab,ba,bb},
4. wyrażenie regularne $(-|+)((1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)|0$ oznacza zbiór wszystkich liczb całkowitych.

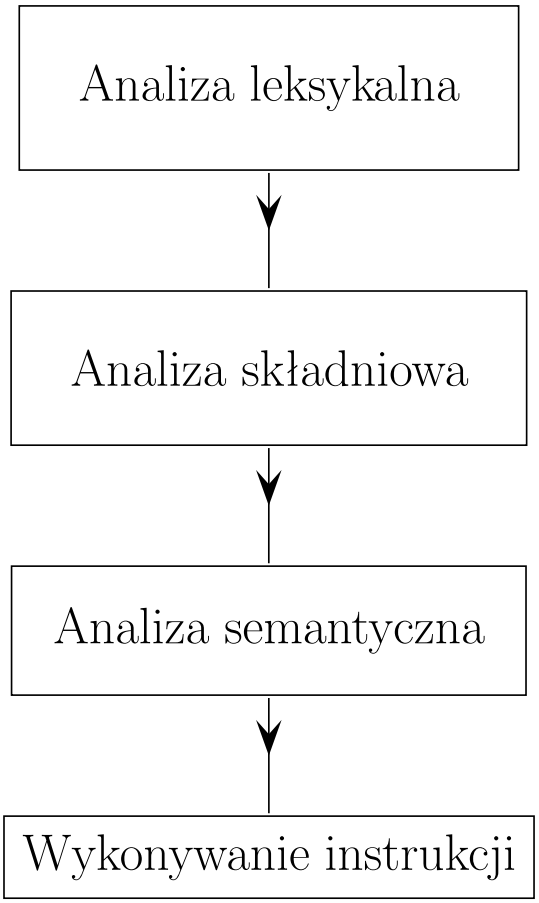
3.2 Język wysokiego poziomu

Językiem wysokiego poziomu *(ang. High-level language)* nazywamy język, którego składnia i słowa kluczowe języka ułatwiają użytkownikom napisanie programu, bądź skryptu, jak również powinien być on wolny od zależności sprzętowych i systemowych.

Celem niniejszej pracy jest napisanie języka wysokiego poziomu, który będzie językiem interpretowanym. Na rysunku 3.1 został przedstawione etapy budowania, są to:

1. Analizator leksykalny — czynności związane z analizą leksykalną, które została przedstawiona w podrozdziale 3.2.1,
2. Analizator składniowy — czynności związane z analizą składniową, które została przedstawiona w podrozdziale 3.2.2,
3. Analizator semantyczny — czynności związane z analizą semantyczną, które została przedstawiona w podrozdziale 3.2.3,
4. Wykonywanie instrukcji — Po przekazaniu polecenia jest ono wykonywane, zgodnie z założeniem programisty lub zwracany jest błąd.

Pierwsze trzy etapy budowania oprogramowania są takie samie jak dla kompilatorów [5].



Rysunek 3.1: Etapy tworzenie języka wysokiego poziomu.

3.2.1 Analiza leksykalna

Zadaniem analizatora leksykalnego jest tworzenie symboli leksykalnych z przesłanego przez użytkownika strumienia znaków na wejście. Symbole te są następnie przesyłane do analizatora składniowego. Symbole leksykalne tworzone są na podstawie tablicy wyrażeń regularnych i odpowiadającym i zadaniom, czyli jeżeli przesłany strumień znaków należy do n-tego wyrażenie regularne w tabeli, to zostanie wykonana powierzone mu zadanie, którym może być na przykład [5]:

- nie wykonanie żadnego działania (strumień znaków zostanie pominięty)
- przesłanie symbolu leksykalnego do analizatora
- przesłanie symbolu leksykalnego wraz z ciągiem znaków, które należą do wyrażenia,

- przesłanie symbolu leksykalnego wraz z modyfikowanym ciągiem znaków.

Podczas analizy leksykalnej możemy zaprojektować analizator leksykalny, aby mógł wykrywać następujące błędy:

- znaki pojawiające się na wejściu nie stanowiące żadnego symbolu leksykalnego,
- przypadku, w których na wejściu pojawia się znak nieobsługiwany,
- sytuacje, w których istnieje możliwość przewidywania błędnych ciągów znaków w danym języku, odpowiadających danemu symbolowi leksykalnemu.

3.2.2 Analiza składniowa

Analizator składniowy otrzymuje od analizatora ciąg symboli leksykalnych, które traktowane są jak symbole terminalne w gramatyce. Zadaniem analizatora składniowego – jest grupowanie symboli leksykalnych i tworzenie drzewa składniowego zgodnie z regułami gramatyki. Zatem wynikiem działania analizatora składniowego jest drzewo składniowe lub wyświetlenie komunikatów o błędach.

Drzewem składniowym *(ang. parse tree)* nazywamy hierarchiczną strukturą danych, w której węzły odpowiadają:

- liściom drzew, które są symbolami terminalnymi, czyli symbolami leksykalnymi, przesyłanymi przez analizator leksykalny,
- węzłom wewnętrznym drzewa, które są symbolami nieterminalnymi.

Najpopularniejszymi metodami wykorzystywanymi w analizatorach składniowych dla gramatyk są zstępujące i wstępujące. W metodzie zstępującej drzewo jest tworzone od korzenia do liści, a wstępującej odwrotnie od liści do korzenia. W obu metodach wejście jest przeglądane od lewej strony.

Analizator składniowy może wykryć błędy w przypadku, gdy strumień symboli leksykalnych nie jest akceptowany przez gramatykę języka. Jeżeli zostanie wykryty

błąd to analizator składniowy powinien próbować odzyskać kontrolę w celu wykrycia kolejnych błędów i poinformować o nich użytkownika. Poniżej zostały wymienione następujące strategie [5]:

Tryb paniki — po natrafieniu na błąd, analizator usuwa symbole leksykalne z wejścia, aż natrafi na symbol z ustalonego zbioru, od których może rozpocząć dalsze poszukiwanie błędów.

Poziom frazy — pozwala zmienić lub dodać symbole leksykalne, które umożliwią dalsze dopasowania,

Produkcja dla błędu — jeżeli wiemy, gdzie pojawiają się najczęściej błędy, to możemy dopisać odpowiednią regułę produkcji w gramatyce.

3.2.3 Analiza semantyczna

Kolejnym etapem z rysunku 3.1, na stronie 40 jest analiza semantyczna, której zadaniem jest sprawdzenie zgodości semantyki języka z jego definicją. Poprawnie wykonana analiza semantyczna definuje jaką rolę powinny pełnić zdania w danym języku.

Wejściem do analizy semantycznej jest poprawne zbudowanie drzewa składniowego według gramatyki [5].

Błędy semantyczne możemy sklasyfikować w następujący sposób [8]:

Błędy krytyczne — błędy uniemożliwiające dalszą analizę.

Błędy tworzące nieoczekiwany stan — istnieją błędy, które nie zostaną przechwycone i zostanie zwrócony nieoczekiwany wynik. Bardzo częstym przykładem tego typu błędu jest program napisany w języku C, który czyta nie ze swojej pamięci,

Niepotrzebnie rozbudowane polecenia — są to polecenia, w które zostały podane nadmiarowe dane.

Rozdział 4

Program

4.1 Opis implementacji programu

Analiza leksykalna została zrealizowana, przy użyciu narzędzia *Lex*, który został stworzony przez M. E. Lesk and E. Schmidt [12]. Oprogramowanie to w znaczący sposób ułatwia tworzenie pierwszego etapu analizy, który został przedstawiony na rysunku 3.1 na stronie 40 i dodatkowo opisany w podrozdziale 3.2.1 Poniżej został przedstawiony listing skryptu lex'a, który w całości realizuje owe zadanie.

Listing 4.1: Analiza leksykalna przy użyciu LEX'a.

```
1  %{
2  #include <stdio.h>
3  #include "y.tab.h"
4
5  char *p;
6  int t_nawiasy =0;
7  int error_lex=0;
8  int error_lex_nieoczekiwany=0;
9  %}
10
11 alpha [a-zA-Z_]
12 digit [0-9]
13 %%
14 "("                    {   ++t_nawiasy;     return LEFTPARENTHESIS;}
15 ")"                    {
16                          if(--t_nawiasy <0)
17                          {
18                          printf("ERROR LEX: Brakuje nawiasu otwierającego '('\n");
```

```
19                              error_lex_nieoczekiwany=2;
20                              }
21                              return RIGHTPARENTHESIS;
22                             }
23
24  E|e|exit|EXIT              {return EXIT;}
25  make|MAKE                  {return MAKE;}
26  create|CREATE              {return CREATE;}
27  key|KEY                    {return KEY;}
28  fact|FACTi                 {return FACT;}
29  dimension|DIMENSION        {return DIMENSION;}
30  save_dir|SAVE_DIR          {return SAVE_DIR;}
31  table_name|TABLE_NAME      {return TABLE_NAME;}
32  base_name|BASE_NAME        {return BASE_NAME; }
33  user_name|USER_NAME        {return USER_NAME;}
34  site_web|SITE_WEB          {return SITE_WEB;}
35  pgloader|PGLOADER          {return PGLOADER;}
36  "http://"[^;\ ]+           {
37                             p=(char *)calloc(strlen(yytext)+1,sizeof(char));
38                             strcpy(p,yytext);
39                             yylval= (int) p;
40                             return SITE_WEB_ADDRESS;
41                             }
42  {digit}+                   {
43                             p=(char *)calloc(strlen(yytext)+1,sizeof(char));
44                             strcpy(p,yytext);
45                             yylval= (int) p;
46                             return NUMBER;
47                             }
48  {alpha}+(\.{alpha})?({alpha}|{digit})*    {
49                             p=(char *)calloc(strlen(yytext)+1,sizeof(char));
50                             strcpy(p,yytext);
51                             yylval= (int) p;
52                             return IDENTIFIER;
53                             }
54  ";"                        {return SEMICOLON;}
55  ","                        {return COMMA;}
56  \n                         /* ignoruj koniec linii */;
57  [ \t]+                     /* ignore białe znaki */;
58  .    {return LEX_ERROR;} /* jakiś znak nie został obsłużony */
59  %%
```

Tworzone symbole leksykalne są przekazywane do kolejnego etapu, który nazywamy analiza składniową. Polega ona na przekształceniu symboli leksykalnych, przy użyciu gramatyki w drzewo składniowe. Na listingu 4.2 został przedstawiony opis utworzonej gramatyki w rozszerzonej notacja Backusa-Naura *(ang. EBNF, Extended Backus-Naur Form*), w której:

- symbolami terminalnymi są:

  ```
  EXIT  SAVE_DIR TABLE_NAME BASE_NAME USER_NAME
  MAKE NUMBER IDENTIFIER
  CREATE KEY FACT DIMENSION SITE_WEB SITE_WEB_ADDRESS PGLOADER
  LEFTPARENTHESIS RIGHTPARENTHESIS  SEMICOLON COMMA  LEX_ERROR
  ```

- symbolami nieterminalnymi są nazwy pisane małymi literami z prefiksem y_

Listing 4.2: Gramatyka języka w notacji EBNF.

```
1  Symbolamim termianlnymi w utworzenej gramatyce są:
2  EXIT  SAVE_DIR TABLE_NAME BASE_NAME USER_NAME
3  MAKE NUMBER IDENTIFIER
4  CREATE KEY FACT DIMENSION SITE_WEB SITE_WEB_ADDRESS PGLOADER
5  LEFTPARENTHESIS RIGHTPARENTHESIS  SEMICOLON COMMA  LEX_ERROR
6
7
8  y_polecenie_srednik  =   y_polecenie SEMICOLON
9
10 y_polecenie =   EXIT
11   | MAKE
12   | SAVE_DIR IDENTIFIER
13   | USER_NAME IDENTIFIER
14   | BASE_NAME IDENTIFIER
15   | SITE_WEB SITE_WEB_ADDRESS IDENTIFIER
16   | PGLOADER IDENTIFIER
17   | y_fakt_polecenie
18   | y_wymiar_polecenie
19   | epsilon
20 ;
21 y_fakt_polecenie =  CREATE FACT IDENTIFIER LEFTPARENTHESIS y_list_fact_kolumn
       RIGHTPARENTHESIS;
22
23 y_list_fact_kolumn =  y_fact_kolumna COMMA y_list_fact_kolumn | y_fact_kolumna ;
24
25 y_fact_kolumna= y_prosta  | y_prosta KEY | y_prosta DIMENSION IDENTIFIER;
```

```
26
27  y_wymiar_polecenie= CREATE DIMENSION IDENTIFIER LEFTPARENTHESIS
        y_list_wymiar_kolumn RIGHTPARENTHESIS ;
28
29  y_list_wymiar_kolumn= y_wymiar_kolumna COMMA y_list_wymiar_kolumn |
        y_wymiar_kolumna ;
30
31  y_wymiar_kolumna= y_prosta  | y_prosta KEY DIMENSION ;
32
33  y_prosta= IDENTIFIER IDENTIFIER
34   | IDENTIFIER IDENTIFIER LEFTPARENTHESIS NUMBER RIGHTPARENTHESIS
35   | IDENTIFIER IDENTIFIER LEFTPARENTHESIS NUMBER COMMA NUMBER RIGHTPARENTHESIS
36  ;
```

Etap ten został zrealizowany przy użyciu programu YACC, utworzonego przez Stephen C. Johnson. Programy LEX i YACC, współpracują ze sobą, dzięki czemu tworzenie drzewa składniowego, którego definicja został podana w rozdziale 3.2.2, staje się znacznie prostsza. Na początku listingu 4.3, po słowach kluczowych token zostały przedstawione symbole leksykalne, a w dalszej części pełna gramatyka tworzonego języka.

Listing 4.3: Gramatyka języka w składni języka YACC.

```
1   %token EXIT  SAVE_DIR TABLE_NAME BASE_NAME USER_NAME
2   %token MAKE  NUMBER IDENTIFIER
3   %token CREATE KEY FACT DIMENSION SITE_WEB SITE_WEB_ADDRESS PGLOADER
4   %token  LEFTPARENTHESIS RIGHTPARENTHESIS  SEMICOLON COMMA  LEX_ERROR
5
6   %%
7
8   // Polecenie ze średnikiem
9   y_polecenie_srednik:
10    y_polecenie SEMICOLON
11  ;
12
13  // polecenie bez średnika
14  y_polecenie:
15    EXIT  // wyjście z programu
16    |
17    MAKE  // tworzyenie sqryptów sql'owy
18    |
19    SAVE_DIR IDENTIFIER  // ustawienie katalogu do zapisu
20    |
21    USER_NAME IDENTIFIER // ustawia nazwę  użytkownika bazy danych
22    |
```

```
23    BASE_NAME IDENTIFIER //  ustawia nazwę bazy danych
24    |
25    SITE_WEB SITE_WEB_ADDRESS IDENTIFIER  // tworzy skrypt pobierające dane
26    |
27    PGLOADER IDENTIFIER  // tworzenie scryptu pgloadera
28    |
29    y_fakt_polecenie    // symbol nieterminalny do tworzenia tablicy faktowej
30    |
31    y_wymiar_polecenie // symbol nieterminalny do tworzenia tablicy wymiarów
32  ;
33
34  y_fakt_polecenie:
35    CREATE FACT IDENTIFIER LEFTPARENTHESIS y_list_fact_kolumn RIGHTPARENTHESIS
36  ;
37
38  y_list_fact_kolumn:
39   y_fact_kolumna COMMA y_list_fact_kolumn
40   |
41   y_fact_kolumna
42  ;
43
44
45  y_fact_kolumna:
46   y_prosta
47  |
48   y_prosta KEY
49  |
50   y_prosta DIMENSION IDENTIFIER
51  ;
52
53  y_wymiar_polecenie:
54    CREATE DIMENSION IDENTIFIER LEFTPARENTHESIS y_list_wymiar_kolumn
        RIGHTPARENTHESIS
55  ;
56
57  y_list_wymiar_kolumn:
58   y_wymiar_kolumna COMMA y_list_wymiar_kolumn
59   |
60   y_wymiar_kolumna
61  ;
62
63  y_wymiar_kolumna:
64   y_prosta
65   |
66   y_prosta KEY DIMENSION
67  ;
68
69  y_prosta:
70  IDENTIFIER IDENTIFIER
```

```
71 |
72 IDENTIFIER IDENTIFIER LEFTPARENTHESIS NUMBER RIGHTPARENTHESIS
73 |
74 IDENTIFIER IDENTIFIER LEFTPARENTHESIS NUMBER COMMA NUMBER RIGHTPARENTHESIS
75 ;
```

W niniejszym programie sprawdzenie poprawności budowania drzewa składniowego, zostało zrealizowane przy użyciu testów jednostkowych *cppunit*, które powstały z wykorzystaniem biblioteki *log4cpp*.

Program nie sprawdza poprawności atrybutów, ponieważ zajmują się tym systemy zarządzające bazą danych. Jeśli atrybuty będą nie poprawne, system zwróci odpowiedni komunikat. Dzięki takiemu rozwiązaniu program utworzony na potrzeby niniejszej pracy staje się wolny od ograniczeń związanych z dostępnymi typami w dowolnej bazie danych. Odpowiedzialność za poprawne wpisywanie typów danych, zostaje przeniesiona na programistę, ponieważ to on powinien wiedzieć jakiego typu danych się oczekuje i czy oczekiwany typ danych jest obsługiwany przez dany system zarządzania bazą danych.

Ostatnim etapem jest wykorzystanie drzewa składniowego w określony sposób. W naszym programie węzłem może być:

- ciąg znaków,
- klasa Tabela,
- klasa Kolumna,

które są umieszczane w klasie Wprowadzone lub od razu zostają wykonywane na nich działania.

Działanie programu polega na przyjmowaniu ciągu znaków zakończonych symbolem średnika (;), które będziemy nazywać poleceniem. Przykładowe polecenia mogą wyglądać następująco:

- exit; — wyjście z programu.

- save_dir — nazwa katalogu do zapisu, w folderze w którym został uruchomiony program.
- user_name login; — login użytkownika bazy danych wykorzystywany przez skrypt pgloadera.
- base_name nazwa; — nazwa bazy danych wykorzystywana przez skrypt pgloadera.
- pgloader nazwa — utworzenie skryptu pgloadera. Nazwa odpowiada nazwie tabeli w hurtowni danych bez prefiksu inf_. Przed użyciem wygenerowanego skryptu musi być zdefiniowana tabela intf_nazwa.
- site_web adres_strony nazwa_pliku — tworzy skrypt linux'owy o rozszerzeniu *.sh z prefiksem dane, który został przedstawiony na listingu 2.3, na stronie 24.
- make; — Na podstawie zgromadzonych tabel wymiarów i jednej tabeli faktów wygenerowane zostają pliki sql, odpowiedzialne za tworzenie tabeli i kolejnych procesów zasilania.

Powyższe polecenia przesyłane są do analizy leksykalnej i składniowej. Wynikiem tych działań jest dodanie odpowiednich wartości do klasy Wyprowadzone lub od razu wykonanie danego działania.

4.2 Przykładowe działania programu.

4.2.1 Abstrakcyjny przykład schematu gwiazdy

Na listingu 4.4 został przedstawiony abstrakcyjny przykład, którego schemat widoczny jest na rysunku 1.3. Pierwsze trzy linie zostały wyjaśnione w poprzednim podrozdziale. Następnie utworzone zostają cztery tabele wymiarów i jedna faktowa (kolejność tutaj nie ma znaczenia). Zapisywane są one w klasie Wprowazdzone. W linii 30 na omawianym listingu zostało wykorzystane plecenie *make;*, które generuje następujące pliki:

- pliki z prefiksem *create_* są to pliki tworzące tabele dla odpowiednich procesów. Dla każdej tabeli wymiarów są to pliki
  - create_intf_nazwa_wymiaru,
  - create_stg_nazwa_wymiaru,
  - create_nazwa_wymiaru, w naszym przykładzie mamy 4 tabele, więc powinniśmy otrzymać 12 plików odpowiedzialnych za utworzenie dwunastu tabel

  Dla tabeli faktów dodatkowo utworzona zostaje jedna tabela create_promo_nazwa_faktu, związku z tym otrzymujemy 4 pliki, łącznie mając ich 16. Program poinformuje nas o utworzonych plikach, tak jak zostało to zaprezentowane na listingu.

- pliki, które nie zaczynają się od *create_*, są plikami, wykorzystywanymi do cyklicznego zasilania. Kolejność tworzenia plików nie jest tutaj przypadkowa. Powinny być one uruchamiane w takiej kolejności, w jakiej zostały utworzone. Jest ich łącznie 11, po dwa dla każdego wymiaru i trzy dla faktu.

Listing 4.4: Przykład działania programu – Schemat gwiazdy

```
Program rozpoczął działanie
=> save_dir etl_gwiazda;
=> user_name etl;
=> base_name dwh;
=>
=> CREATE DIMENSION wymiar1
-> (
->  atrybut varchar key dimension
-> );
=>
=> CREATE DIMENSION wymiar2
-> (
->   atrybut_1 varchar key dimension
-> , atrybut_2 numeric
-> );
=>
=> CREATE DIMENSION wymiar3
-> (
->  atrybut varchar  key dimension
-> );
=>
=> CREATE DIMENSION  wymiar4
-> (
```

```
24 -> atrybut varchar key dimension
25 -> );
26 =>
27 => CREATE fact fakt
28 -> (
29 ->   id_wymiar1 integer  dimension wymiar1
30 -> , id_wymiar2 integer  dimension wymiar2
31 -> , id_wymiar3 integer  dimension wymiar3
32 -> , id_wymiar4 integer  dimension wymiar4
33 -> , data       timestamp
34 -> , miara_1    integer
35 -> , miara_2    integer
36 -> );
37 =>
38 => make;
39 Utworzono: ./etl_gwiazda/create_intf_fakt.sql
40 Utworzono: ./etl_gwiazda/create_stg_fakt.sql
41 Utworzono: ./etl_gwiazda/create_promo_fakt.sql
42 Utworzono: ./etl_gwiazda/create_fakt.sql
43 Utworzono: ./etl_gwiazda/create_intf_wymiar1.sql
44 Utworzono: ./etl_gwiazda/create_stg_wymiar1.sql
45 Utworzono: ./etl_gwiazda/create_wymiar1.sql
46 Utworzono: ./etl_gwiazda/create_intf_wymiar2.sql
47 Utworzono: ./etl_gwiazda/create_stg_wymiar2.sql
48 Utworzono: ./etl_gwiazda/create_wymiar2.sql
49 Utworzono: ./etl_gwiazda/create_intf_wymiar3.sql
50 Utworzono: ./etl_gwiazda/create_stg_wymiar3.sql
51 Utworzono: ./etl_gwiazda/create_wymiar3.sql
52 Utworzono: ./etl_gwiazda/create_intf_wymiar4.sql
53 Utworzono: ./etl_gwiazda/create_stg_wymiar4.sql
54 Utworzono: ./etl_gwiazda/create_wymiar4.sql
55 Utworzono: ./etl_gwiazda/stg_fakt.sql
56 Utworzono: ./etl_gwiazda/stg_wymiar1.sql
57 Utworzono: ./etl_gwiazda/stg_wymiar2.sql
58 Utworzono: ./etl_gwiazda/stg_wymiar3.sql
59 Utworzono: ./etl_gwiazda/stg_wymiar4.sql
60 Utworzono: ./etl_gwiazda/promo_fakt.sql
61 Utworzono: ./etl_gwiazda/wymiar1.sql
62 Utworzono: ./etl_gwiazda/wymiar2.sql
63 Utworzono: ./etl_gwiazda/wymiar3.sql
64 Utworzono: ./etl_gwiazda/wymiar4.sql
65 Utworzono: ./etl_gwiazda/fakt.sql
66 =>
67 => exit;
68 Koniec działania programu
```

4.2.2 Realizacja przykładu opisanego w rozdziale 2

W niniejszym podrozdziale został zaprezentowana realizacja omówionego przykładu w podrozdziale 2.2. Poprzednim podrozdziale zostało wspomniane, że kolejność tworzenia tabel nie ma znaczenia, jest to spowodowane tym, że po słowach kluczowych *dimension* i *key* podczas tworzenia kolumny musi wystąpić nazwa wymiaru. Jeżeli wymiar o tej nazwie nie zostanie zdefiniowany wcześniej to zostanie wygenerowany omówiony przykład.

W tabeli faktowej w kolumnie data_notowania pojawiło się słowo kluczowe języka: key. W ten sposób sygnalizujemy, że dana wartość jest kluczem w tabeli faktów i nie chcemy tworzyć dla nie klucza sztucznego.

Listing 4.5: Przykład działania programu – Notowania GPW bez tabeli wymiarów.

```
Program rozpoczął działanie
=> save_dir etl_a;
=> user_name etl;
=> base_name dwh;
=> site_web http://bossa.pl/pub/ciagle/mstock/sesjacgl/sesjacgl.prn gpw;
Utworzono: ./etl_a/script_gpw.sh
=> pgloader gpw;
Utworzono: ./etl_a/pgloader_gpw.conf
=>
=>  create fact gpw (
->   nazwa varchar(50) dimension npw
-> , data_notowania date key
-> , otwarcie decimal(20, 2)
-> , max decimal(20, 2)
-> , min decimal(20, 2)
-> , zamkniecie decimal(20, 2)
-> , wartosc decimal(20,3)
-> );
=>
=> make;
Utworzono: ./etl_a/create_intf_gpw.sql
Utworzono: ./etl_a/create_stg_gpw.sql
Utworzono: ./etl_a/create_promo_gpw.sql
Utworzono: ./etl_a/create_gpw.sql
Utworzono: ./etl_a/create_npw.sql
Utworzono: ./etl_a/stg_gpw.sql
Utworzono: ./etl_a/kmap_from_gpw_to_npw.sql
Utworzono: ./etl_a/promo_gpw.sql
Utworzono: ./etl_a/gpw.sql
=>
```

```
31 => exit;
32 Koniec działania programu
```

Na listingu 4.6 została przedstawiona pełna realizacja owego przykładu. Została w nim dodana tabela wymiarów, polecenie tworzące skrypt do pobierania i skrypt pgloadera. Niestety pobierane dane do tabeli wymiarów wymagają modyfikacji, aby mogły być one załadowane do bazy danych przy użyciu skryptu pgloadera. Modyfikacje obejmują:

- usunięcie trzech pierwszych linii,
- usunięcie ostatniej linii,
- rozdzielenie kolumn znakiem

Listing 4.6: Przykład działania programu – Notowania GPW.

```
1  Program rozpoczął działanie
2  => save_dir etl_b;
3  => user_name etl;
4  => base_name dwh;
5  =>
6  => site_web http://bossa.pl/pub/ciagle/mstock/sesjacgl/sesjacgl.prn gpw;
7  Utworzono: ./etl_b/script_gpw.sh
8  => site_web http://bossa.pl/pub/ciagle/mstock/metacgl.lst npw;
9  Utworzono: ./etl_b/script_npw.sh
10 =>
11 => pgloader gpw;
12 Utworzono: ./etl_b/pgloader_gpw.conf
13 => pgloader npw;
14 Utworzono: ./etl_b/pgloader_npw.conf
15 =>
16 =>  create fact gpw (
17 ->   nazwa varchar(50) dimension npw
18 -> , data_notowania date key
19 -> , otwarcie decimal(20, 2)
20 -> , max decimal(20, 2)
21 -> , min decimal(20, 2)
22 -> , zamkniecie decimal(20, 2)
23 -> , wartosc decimal(20,3)
24 -> );
25 =>
26 => create dimension npw
27 -> (
```

```
28 -> data_od timestamp
29 -> , data_do timestamp
30 -> , nazwa varchar (50) key dimension
31 -> , opis varchar(200)
32 -> );
33 =>
34 => make;
35 Utworzono: ./etl_b/create_intf_gpw.sql
36 Utworzono: ./etl_b/create_stg_gpw.sql
37 Utworzono: ./etl_b/create_promo_gpw.sql
38 Utworzono: ./etl_b/create_gpw.sql
39 Utworzono: ./etl_b/create_intf_npw.sql
40 Utworzono: ./etl_b/create_stg_npw.sql
41 Utworzono: ./etl_b/create_npw.sql
42 Utworzono: ./etl_b/stg_gpw.sql
43 Utworzono: ./etl_b/stg_npw.sql
44 Utworzono: ./etl_b/promo_gpw.sql
45 Utworzono: ./etl_b/npw.sql
46 Utworzono: ./etl_b/gpw.sql
47 =>
48 => exit;
49 Koniec działania programu
```

Podsumowanie

Najważniejszym celem niniejszej pracy było stworzenie języka wysokiego poziomu wspomagającego tworzenie procesów zasilających hurtownie danych. Język ten odpowiada za częściową automatyzację pracy programisty, najbardziej powtarzalnych i najmniej ciekawych czynności, a zatem także najbardziej podatnych na błędy.

Dzięki pracy nad projektem języka wysokiego poziomu udało się osiągnąć zamierzony efekt, którym było zminimalizowanie ilości napisanego kodu, a tym samym zwiększenie efektywności pracy programisty. Osiągnięty cel potwierdzają listingi przykładowych użyć programu, które zostały zamieszczone w podrozdziale 4.2.

Stworzony na potrzeby pracy program posiada następujące cechy:

- skrypt odpowiedzialny za pobranie pliku z danymi zostaje utworzony przy użcyiu jednego polecenia,
- skrypt pgloader'a odpowiedzialny za załadowanie danych do bazy wygenerowany zostaje również przy użyciu jednego polecenia,
- raz utworzona tabela wymiarów lub tabela faktów jest wielokrotnie wykorzystywana do tworzenia procesów zasilających,
- automatycznie tworzone przez program tabele przejściowe i tabele docelowe pozwalają uniknąć błędów, które pojawiają się często podczas pisania kodu SQL,
- język, w którym został napisany program jest prosty i intuicyjny, co pozwala uniknąć błędów składniowych.
- elastyczność działania wynikająca z niezależności względem wykorzystywanego systemu bazodanowego.

Dzięki wielu godzinom poświęconym hurtowniom danych, zauważyłem jak niewiele w tej dziedzinie zostało poświęcone odpowiednim procesom automatyzującym i przyspieszającym tworzenie hurtowni danych. Dzięki zdobytemu podczas pisania tej pracy doświadczeniu spostrzegłem jak istotne jest to zagadnienie dla dziedziny jaką są hurtownie danych i jak procesy usprawniające tworzenie hurtowni mogą wpłynąć na efektywność pracy programistów. Dlatego też za cel obrałem sobie stworzenie języka wysokiego poziomu, który by tą pracę usprawniał, a zarazem czynił ją łatwiejszą. Program, który został napisany na potrzeby tej pracy dyplomowej może być wykorzystywany jako język generujący hurtownie danych "w locie" albo może po prostu usprawniać pracę programisty zajmującego się hurtowniami danych.

Tworzenie hurtowni danych jest procesem bardzo złożonym, dlatego perspektywa dalszego rozwoju języka daje również szerokie możliwości, które nie kończą się bynajmniej na funkcjonalnościach opisanych w powyższej pracy. Dostrzegam wiele potencjalnych funkcjonalności, o które można by rozbudować ten język. Do tych funkcjonalności zaliczam:

1. stworzenie skryptu, którego zadaniem byłoby uruchamianie w sposób cykliczny utworzonych procesów zasilających,
2. dodawanie lub usuwanie nowych elementów w warstwach procesów zasilających,
3. zapisywanie informacji o utworzonych procesach w bazie danych w sposób umożliwiający stworzenie zapytania zwracającego poprawną kolejność uruchamianych procesów,
4. poszerzenie składni języka, która mogła by posłużyć do zapisywania ważnych informacji dotyczących tabel, dzięki czemu będzie możliwe automatyczne tworzenie częściowej dokumentacji w języku LaTeX,
5. utworzone pliki można by przechowywać na dysku w uporządkowanej strukturze katalogów,
6. uniwersalność języka sprawia, że może być on w przyszłości rozwijany niezależnie od wykorzystywanego systemu bazodanowego.

Spis rysunków

1.1 Architektura hurtowni danych. 11

1.2 Architektura hurtowni danych z magazynem danych ODS. 12

1.3 Przykładowy schemat gwiazdy w postaci abstrakcyjnej. 15

1.4 Przykładowy schemat płatka śniegu w postaci abstrakcyjnej. 17

3.1 Etapy tworzenie języka wysokiego poziomu. 40

Listingi kodu

1.1 Listing kodu tworzący schemat gwiazdy. 15

2.1 Kod tworzący tabelę faktów. 23

2.2 Kod tworzący tabelę wymiaru. 23

2.3 Skrypt pobierający dane serwera bossa.pl 24

2.4 Kod tworzący tabelkę interfejsową. 25

2.5 Skrypt pgloader'a ładujący dane do tabelki interfejsowej. 26

2.6 Usuwanie pobranych duplikatów. 28

2.7 Ładowanie danych do tabeli wymiaru – papierów wartościowych. 28

2.8 Struktura tabelki promo_gpw. 29

2.9 Proces ładowania do tabeli promo_gpw. 29

2.10 Proces ładowania danych do tabeli gpw. 30

4.1 Analiza leksykalna przy użyciu LEX'a. 43

4.2 Gramatyka języka w notacji EBNF. 45

4.3 Gramatyka języka w składni języka YACC. 46

4.4 Przykład działania programu – Schemat gwiazdy 50

4.5 Przykład działania programu – Notowania GPW bez tabeli wymiarów. 52

4.6 Przykład działania programu – Notowania GPW. 53

Bibliografia

[1] Chris Todman, *Projektowanie Hurtowni Danych. Wspomaganie zarządzania relacjami z klientem*, Wydawnictwa HELION 2011.

[2] W.H. Inmon, *Building the Data Warehouse, Fourth Edition*, Wydawnictwo Wiley Publishing, Inc. 2005

[3] Kimball R., Ross M., *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*, Wydawnictwo John Wiley and Sons, Inc. 2004

[4] Rainardi V., *Building a data warehouse with exmaples in SQL server*, Wydawnictwo Springer-Verlag New York, Inc. 2008

[5] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Kompilatory*, Wydawnictwa Naukowo-Techniczne, Warszawa 2002

[6] `http://edu.pjwstk.edu.pl/wyklady/hur/scb/`, dnia 03.III.2014

[7] `http://etl-tools.info/pl/bi/hurtownia_danych_schemat-gwiazdy.htm`, dnia 06.IV.2014

[8] `http://dbs.informatik.uni-halle.de/sqllint/semerr_techrep.pdf`, dnia 14.VI

[9] `http://en.wikipedia.org/wiki/Chomsky_hierarchy`, dnia 20.IV.2014

[10] `http://pl.wikipedia.org/wiki/Hierarchia_Chomsky'ego`, dnia 20.IV.2014

[11] `http://en.wikipedia.org/wiki/Formal_grammar`, dnia 20.IV.2014

[12] `http://dinosaur.compilertools.net/`, dnia 25.VI.2014