AKADEMIA GÓRNICZO-HUTNICZA
Wydział Informatyki, Elektroniki i Telekomunikacji


PROJEKT INŻYNIERSKI

# Implementacja wybranych funkcjonalności silnika SQLxD

## Dokumentacja techniczna

Wersja 1.0.0 z dnia 06.01.2014

Zespół:
Adam Furmanek
Jakub Tokaj

Opiekun: dr inż. Robert Marcjan
Prowadzący pracownię projektową:
dr inż. Rafał Dreżewski

# OŚWIADCZENIE AUTORÓW PRACY

*Oświadczamy, świadomi odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonaliśmy osobiście i samodzielnie (w zakresie wyszczególnionym we wstępie) i że nie korzystaliśmy ze źródeł innych, niż wymienione w pracy.*

Adam Furmanek Jakub Tokaj

# Spis treści

# Wstęp

Niniejszy dokument opisuje implementację silnika bazodanowego *SQLxD*, operującego nie danymi relacyjnymi, jak większość obecnie dostępnych rozwiązań, ale na danych hierarchicznych zapisanych w formacie *XML*. Cechą szczególną *SQLxD* jest zapewnienie wysokiej zgodności z typową składnią zapytań dyktowaną przez standard *SQL/92* [1], a nie wykorzystywanie mechanizmów stworzonych dla danych hierarchicznych, jak *XPath* [2] czy *XQuery* [3].

Głównym problemem przy operowaniu na danych jest transformacja modelu hierarchicznego do modelu relacyjnego, który poprzez swoją konstrukcję jest modelem płaskim, składającym się z wielu tabel. Może się to wydawać wbrew idei baz danych *SQL* i postulatów E.F. Codda [4], jednak nawet sam standard *SQL/92* [1] nie mówi o danych relacyjnych, a o *danych SQL*, które jedynie są przechowywane w *tabelach SQL*. Ze względu na inną naturę dokumentów hierarchicznych, opracowano przynajmniej kilka interesujących mechanizmów pozwalających na operowanie danymi zapisanymi w formacie *XML* – *XPath* [2], *XQuery* [3], czy chociażby *LINQ to XML* [5]. Wszystkie te metody mają jednak jedną cechę wspólną – nie ukrywają hierarchicznej budowy danych przed użytkownikiem, co oznacza, że osoba wykorzystująca te mechanizmy musi być świadoma sposobu zapisu informacji. Nie inaczej jest również w przypadku mechanizmów zarządzania danymi hierarchicznymi, wbudowanych w systemy bazy danych, chociażby mechanizm dostępny w *MSSQL 2008* [6].

Nasz silnik bazy danych stara się ukryć przed użytkownikiem budowę hierarchiczną dokumentu, poprzez transformację struktury drzewiastej na strukturę relacyjną, czyli typową strukturę *SQL*. Sam proces transformacji został ideowo opisany przez Jakuba Wyrostka w pracy *Przetwarzanie dokumentów XML w oparciu o model quasi-relacyjny wraz z projektem języka zapytań* [7], naszym zadaniem było uszczegółowienie pewnych kwestii i zaimplementowanie silnika.

## Wprowadzenie

Silnik *SQLxD* składa się z kilku głównych elementów:

- Moduł parsera danych
- Moduł parsera zapytań
- Silnik
- Aplikacja okienkowa

Pierwszy z tych elementów, czyli moduł parsera danych, odpowiada za transformację dokumentu *XML* do postaci relacyjnej. Jest odpowiedzialny za zmianę struktury hierarchicznej w strukturę tabelaryczną, dzięki czemu możliwe będzie wykonywanie zapytań *SQL* działających na danych relacyjnych.

Drugi z elementów, czyli moduł parsera zapytań, odpowiada za analizę zapytania podanego przez użytkownika. Jego rolą jest zbudowanie drzewa zapytania, które potem może zostać wykonane przez silnik bazy danych.

Trzeci element – silnik – to serce całej aplikacji. Odpowiada za fizyczne wykonanie zapytania i zwrócenie danych, o które chodzi użytkownikowi.

Ostatnim elementem jest aplikacja okienkowa, której autorami są Michał Dąbrowski i Sebastian Kulpa. Aplikacja ta została zintegrowana z silnikiem *SQLxD*, dzięki czemu użytkownik może w przyjemny sposób wczytać plik *XML* i wykonać interesujące go zapytanie.

## Cel

Celem niniejszego dokumentu jest opisanie wymagań stawianych systemowi ze względu na jego przeznaczenie i sposób użycia. Ponadto opisana zostanie architektura systemu, technologie użyte podczas implementacji, sposoby i plan testów systemu. Opisana zostanie ogólna implementacja systemu, a dla najtrudniejszych problemów algorytmicznych i implementacyjnych dodane zostaną szczegółowe opisy w dalszej części dokumentu. Zostanie również przedstawiona gramatyka robocza akceptowana przez *SQLxD*, jej różnice w stosunku do gramatyki *SQL/92* [8], a także powody, dla których musiała być ona zmodyfikowana.

# Słownik

Adresacja - pozwala nam wskazać, z których węzłów dokumentu pobieramy dane do dalszego przetwarzania w silniku *SQLxD*. Składnia została opisana w rozdziale dotyczącym selektorów.

Agregat, funkcja agregująca - podobnie jak w przypadku języka *SQL*, jest to funkcja która wykonuję obliczenia na zestawie wartości i zwraca jedną wartość.

Aplikacja okienkowa - graficzny interfejs użytkownika pozwalający korzystać w łatwy sposób z biblioteki silnika *SQLxD*, napisana przez Sebastiana Kulpę i Michała Dąbrowskiego, opisana w [9].

Dokument - zestaw danych zapisanych w pliku tekstowym przy użyciu języka *XML*.

Funkcja – przekształcenie, które na podstawie podanych argumentów zwraca pojedynczą wartość skalarną.

Gramatyka – opis języka formalnego.

GUID – Globally Unique Identifier – 128-bitowy identyfikator będący praktycznie unikalny na całym świecie.

Klauzula – wydzielony strukturalnie fragment zapytania.

Kolumna - zestaw wartości tego samego typu.

Komórka - pojedyncza wartość z relacji.

Krotka – zbiór komórek.

Model quasi-relacyjny - model opisany przez Jakuba Wyrostka w jego pracy magisterskiej [7], jest wzorowany na modelu relacyjnym znanym z języka SQL. Powstaje w wyniku transformacji modelu hierarchicznego *XML*.

Moduł - wydzielony fragment silnika *SQLxD*, odpowiadający za daną funkcjonalność.

Parser – mechanizm wykonujący parsowanie.

Parsowanie – transformacja do innej postaci w celu rozpoznania znaczenia wyrażenia.

Potomek - węzeł jest potomkiem dla innego węzła, gdy się w nim zawiera.

Projekcja - operacja pozwalająca na wybranie zestawu kolumn z relacji, na podstawie podanych kryteriów.

Przodek - węzeł jest przodkiem dla innego węzła, gdy zawiera ten węzeł.

Rekord - patrz wiersz.

Relacja - zbiór krotek.

Silnik - zestaw algorytmów do transformacji modelu hierarchicznego na model quasi-relacyjny. Udostępnia API do wykonywania zapytań.

SQL – język zapytań służący do pobierania danych z bazy danych.

SQL/92 – standard [1] języka SQL

*SQLxD* - autorski język zapytań, wzorowany na języku SQL, służy do wykonywania zapytań na modelu quasi-relacyjnym używanym w silniku.

Transformacja - zamiana modelu hierarchicznego, używanego w dokumentach *XML* na model quasi-relacyjny, używany w silniku *SQLxD*.

Użytkownik – osoba używająca aplikacji okienkowej lub silnika *SQLxD*.

Węzeł - nawiązując do modelu *DOM*, wszystko w dokumencie *XML* jest węzłem. W szczególności: cały dokument, każdy element *XML*, tekst w elemencie *XML*, atrybut i komentarz.

Węzeł główny dokumentu - najbardziej zewnętrzny węzeł w dokumencie, będący rodzicem dla wszystkich innych węzłów - tak zwany "korzeń" dokumentu.

Wiersz - pojedyncza krotka w relacji.

Zapytanie - poprawne wyrażenie napisane przy pomocy języka *SQLxD*, pozwalające na wykonanie określonej operacji na modelu quasi-relacyjnym.

# Wizja projektu

## Opis problemu

*SQLxD* jest autorskim rozwiązaniem umożliwiającym przetwarzanie dokumentów *XML* przy pomocy języka relacyjnego o składni podobnej do języka *SQL*. Autor języka zaimplementował jedynie podstawową funkcjonalność jako *proof of concept* (ang. „dowód koncepcji") oraz nakreślił w swojej pracy magisterskiej dalsze możliwości rozwoju *SQLxD*.

Celem naszego projektu jest implementacja w pełni funkcjonalnego silnika bazy danych *SQLxD* na podstawie planów rozwoju opisanych przez autora pracy.

Szczegółowy opis zagadnienia można znaleźć w pracy magisterskiej Jakuba Wyrostka Przetwarzanie dokumentów XML w oparciu o model quasi-relacyjny wraz z projektem języka zapytań [7].

Dodatkowym celem jest integracja stworzonego przez nas silnika z aplikacją *SQLxD Developer* stworzoną przez Michała Dąbrowskiego i Sebastiana Kulpę [9].

## Opis użytkownika

Użytkownikiem może być każdy programista lub osoba z podstawową znajomością języka *SQL*. Z założenia aplikacja ma pozwalać na manipulację danymi z dokumentów *XML* każdemu znającemu język *SQL*, bez konieczność nauki nowego języka (np. *XPath* [2] lub *XQuery* [3]).

## Opis produktu

Końcowym produktem będzie biblioteka zawierająca implementację silnika bazodanowego *SQLxD*. Silnik ma pozwalać na wczytywanie dokumentów *XML* i wykonywanie na nich operacji znanych z klasycznych silników baz danych. W szczególności wspierane mają być operacje: projekcja *SELECT*, wybieranie *FROM*, filtrowanie *WHERE*, złączenia naturalne, złączenia typu *LEFT/RIGHT/CROSS/FULL INNER/OUTER JOIN*, wykonywanie funkcji, grupowanie *GROUP BY*, sortowanie *ORDER BY* i obliczanie agregatów.

### Wymagania funkcjonalne

- Dostarczenie silnika bazy danych opartej o koncepcję *SQLxD*
- Możliwość wykonywania zapytań *SELECT ... FROM*
- Możliwość filtrowania wyników zapytania przez warunki zawarte w klauzuli *WHERE*
- Możliwość dokonywania złączeń naturalnych
- Możliwość dokonywania złączeń typu *LEFT/RIGHT/FULL/CROSS INNER/OUTER JOIN*
- Możliwość wykonywania grupowania z użyciem klauzuli *GROUP BY*
- Możliwość wykonywania funkcji agregujących
- Możliwość sortowania wyników zapytań z użyciem klauzuli *ORDER BY*
- Możliwość łatwej integracji silnika z aplikacją *SQLxD Developer*

### Wymagania niefunkcjonalne

- Technologia wykonania: *.NET 4.5*
- Język wykonania: *C# 5.0*
- Wspierany system operacyjny: rodzina systemów *Microsoft Windows*
- Dokumentacja techniczna
- Dokumentacja procesowa
- Dokumentacja użytkownika

## Wstępna analiza ryzyka

### Nieznajomość zagadnienia

Praca inżynierska będzie opierała się o pracę Jakuba Wyrostka, z którą spotykamy się pierwszy raz, więc de facto nie wiemy, jakich problemów możemy się spodziewać. Ponadto tego rodzaju baza danych nie jest popularna, więc nie mamy możliwości wzorowania się na cudzych implementacjach, całą pracę musimy oprzeć o wspomnianą już pracę magisterską.

### Użyta technologia

W toku studiów nie mieliśmy dużej styczności z językiem *C#* oraz platformą *.NET*, co może spowodować problemy i nieprzewidziane trudności.

### Nieznajomość aplikacji "SQLxD Developer"

Niestety przed rozpoczęciem prac nie mieliśmy styczności z aplikacją *SQLxD Developer* opisanej w [9], nie wiemy również, jak będzie wyglądał techniczny aspekt integracji naszego silnika bazodanowego z wspomnianą aplikacją, w związku z tym nie jesteśmy w stanie określić, ile czasu nam to zajmie.

### Brak czasu

Pracę inżynierską będziemy realizowali na siódmym semestrze studiów, co oznacza, że będziemy musieli łączyć jednoczesne rozwijanie silnika i studiowanie, co może niekorzystnie rzutować na ilość dostępnego czasu.

# Architektura systemu

## Wstęp

Biblioteka silnika *SQLxD* składa się aktualnie z 4 modułów: *Model*, *Parser*, *QueryLogic* i *QueryParser*. Do każdego modułu istnieje dedykowany moduł z testami, dodatkowo dodany został piąty moduł zawierający testy integracyjne do całego silnika. Sercem biblioteki są moduły zawierające model i logikę zapytań - zawierają one reprezentacje relacji, algorytmy do transformacji hierarchicznego modelu *XML* na model relacyjny i algorytmy implementujące zapytania *SQL*. W projekcie istnieją dwa moduły parserów, jeden służy do parsowania danych – dokumentów *XML* - i tworzenia z nich modelu wykorzystywanego później przy transformacji. Drugi moduł parsera służy do parsowania zapytań języka *SQL* i budowania na ich podstawie wewnętrznej reprezentacji zapytania, umożliwiając łatwe używanie biblioteki. Z punktu widzenia użytkownika biblioteki, najważniejsze są właśnie moduły parsujące, a konkretnie ich klasy wejściowe: *XmlDocumentParser*, oraz *QueryParser*, które zostaną opisane szczegółowo w kolejnych akapitach.

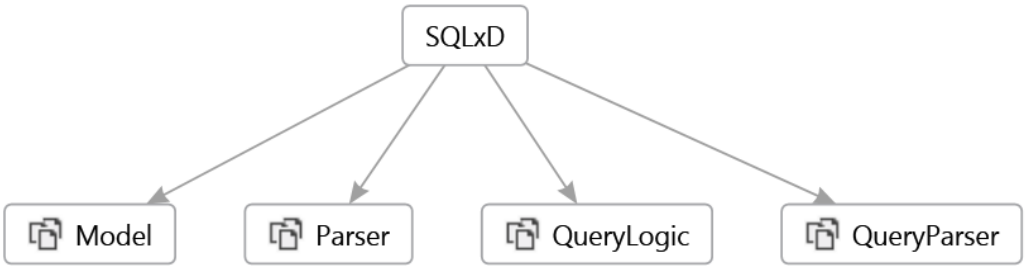



**Diagram architektury projektu**

## Moduły

### Model

Moduł jak nazwa wskazuję, zawiera klasy reprezentujące logiczny model naszej aplikacji. Można go podzielić na dwie grupy. Pierwsza grupa reprezentuje dokument *XML* i jest wykorzystywana przez moduł *Parsera* podczas przetwarzania dokumentu oraz przez moduł *QueryLogic* przy tworzeniu reprezentacji relacyjnej. Należą do niej klasy: *Node* i *NodeType*. Druga grupa służy do reprezentacji relacji i jest wykorzystywana przez moduły *QueryLogic* oraz *QueryParser*. Należą do niej klasy: *Relation*, *Row*, *Cell*, *RowBuilder*, *ColumnHeader*, *GuidCell* oraz *GuidHeader*.

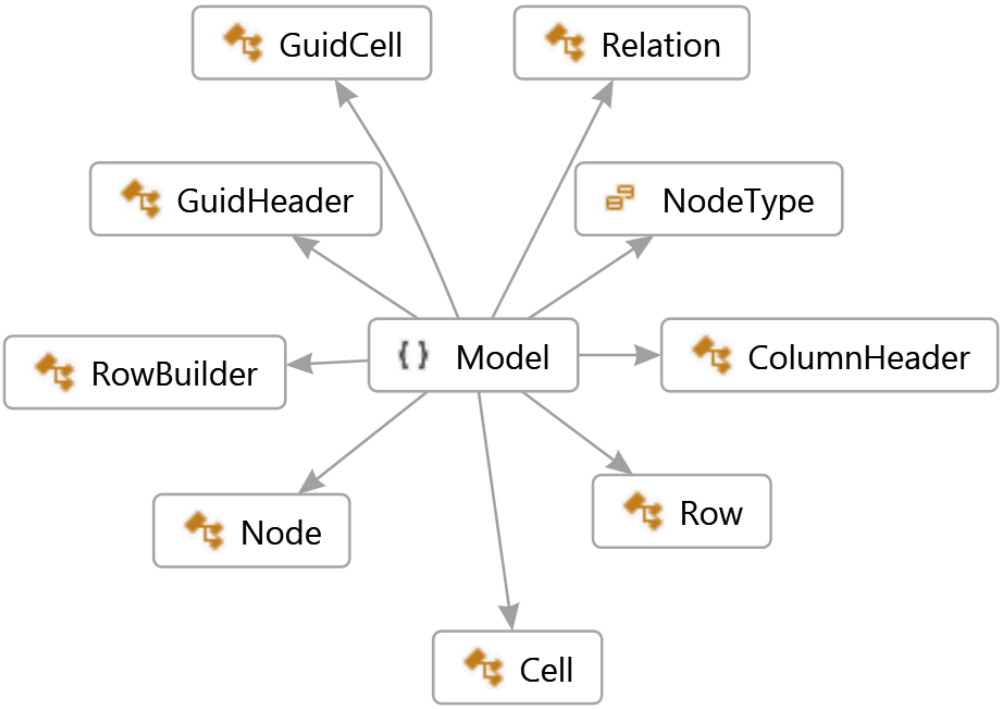



Diagram architektury Modelu

### *Reprezentacja dokumentu* XML

W pracy magisterskiej *Przetwarzanie dokumentów XML w oparciu o model quasi-relacyjny wraz z projektem języka zapytań* [7], autor wyróżnił trzy typy węzłów w dokumencie. Typ *Element* reprezentuję znacznik lub parę znaczników *XML*: `<znacznik></znacznik>`. Typ *Attribute* reprezentuję węzeł atrybutu, który może być przypisany do znacznika: `<znacznik atrybut=”węzeł końcowy”/>`. Typ *Text* reprezentuję końcowy węzeł dokumentu, zawierający tekst: `<znacznik>węzeł końcowy</znacznik>`. Inne typy węzłów na podstawie pracy nie są uznawane za prawidłowe.

Klasa *Node* reprezentuję pojedynczy węzeł z dokumentu *XML*. Węzeł posiada któryś z typów opisanych wyżej, unikatowy identyfikator (*GUID*), wartość, wskaźnik na węzeł rodzica oraz listę węzłów – dzieci. Węzeł typu *Text*, jako węzeł końcowy, nie może posiadać węzłów – dzieci.

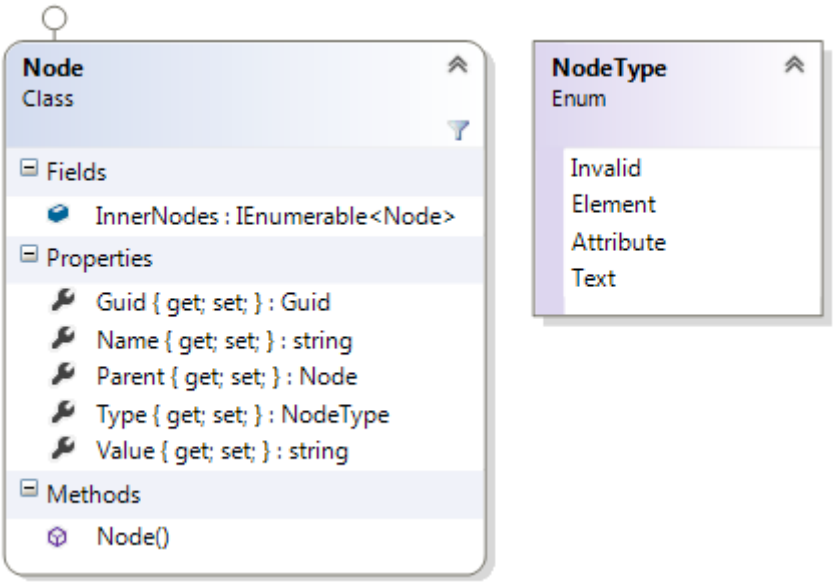
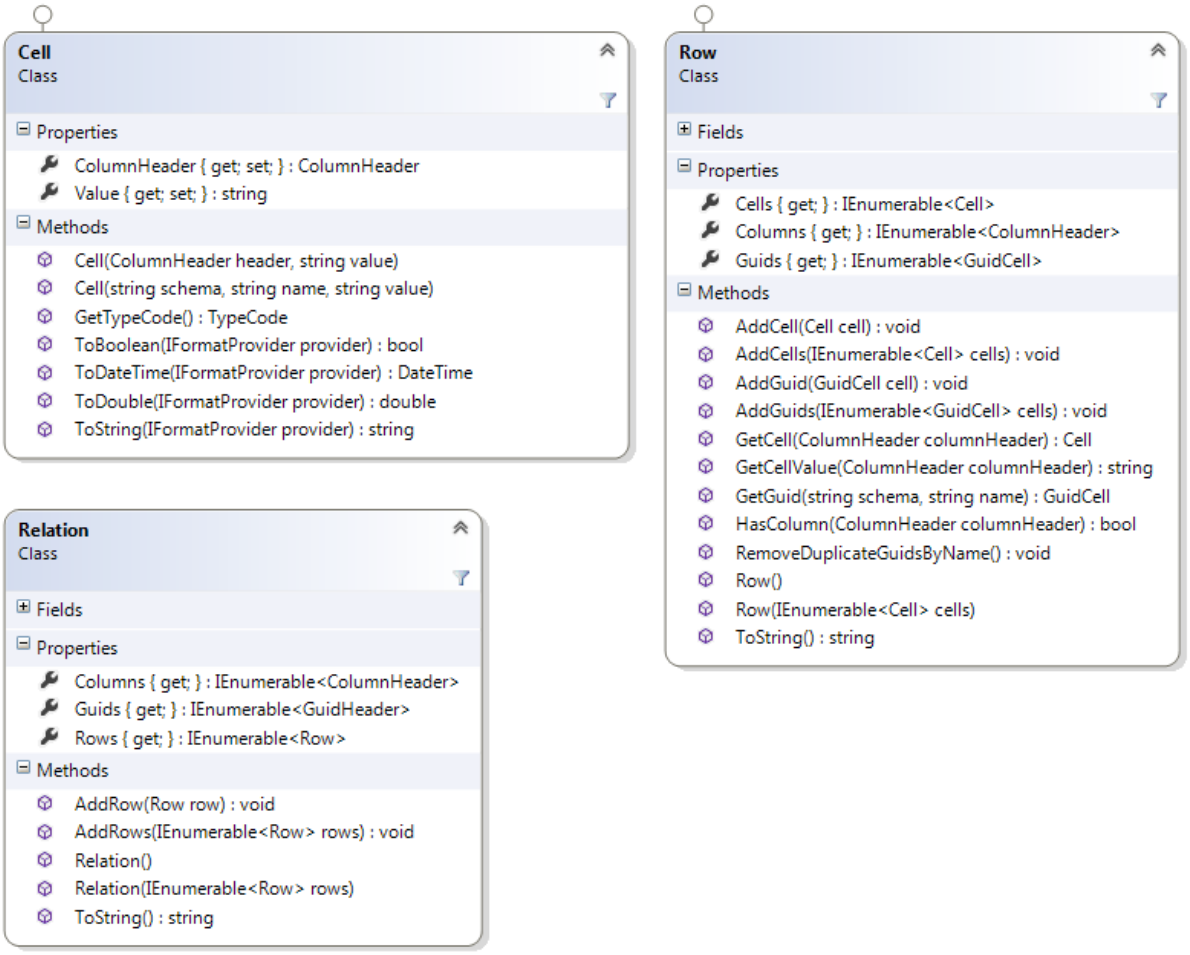



**Diagram klasy *Node* i typ wyliczeniowy *NodeType***

### *Reprezentacja relacji*

Relacja jest reprezentowana przez klasę *Relation*, która zawiera zbiór kolumn zawartych w relacji, oraz wiersze (*Row*) zawierające dane. Z kolei wiersze relacji składają się z komórek (*Cell*) reprezentujących „indywidualne wartości skalarne”. Relację są początkowo tworzone na podstawie dokumentu *XML*, później za pomocą modułu *QueryLogic* możliwe jest wykonywanie na nich operacji znanych z języka *SQL*.




Diagramy klas *Cell*, *Row*, *Relation*

## Parser

Moduł zawiera logikę służącą do zamiany tekstowego dokumentu *XML* na wykorzystywany wewnętrznie model dokumentu *XML*. Do parsowania dokumentu zostały użyte biblioteki `System.Xml` oraz `System.Xml.Linq`, dostarczane przez platformę *.NET*.

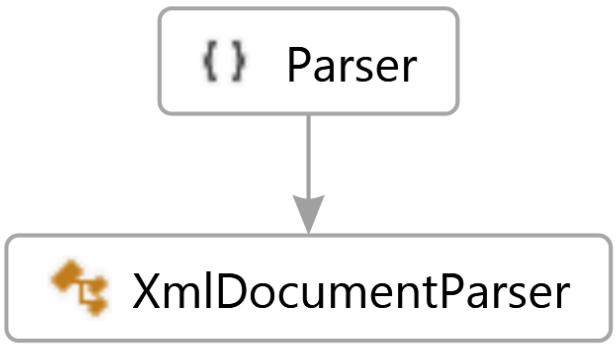


Diagram architektury modułu *Parser*

Klasa *XmlDocumentParser* udostępnia interfejs składający się z jednej metody: `Node Parse(string xml)`. Metoda przyjmuję obiekt tekstowy w formacie *XML* typu *String*, efektem wykonania metody jest obiekt *Node* będący węzłem - korzeniem dokumentu *XML*.

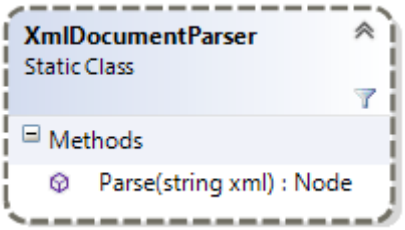



Diagram klasy *XmlDocumentParser*

## QueryLogic

Moduł zawiera logikę zapytań, jest sercem i jednocześnie największym modułem biblioteki. Zawiera logiczną reprezentację zapytania *SQL* oraz algorytmy potrzebne do wykonania takiego zapytania.

Został on podzielony na 12 pakietów, które reprezentują różne części zapytania z gramatyki SQL. Pakiety zostaną opisane szczegółowo w dalszej części dokumentu.

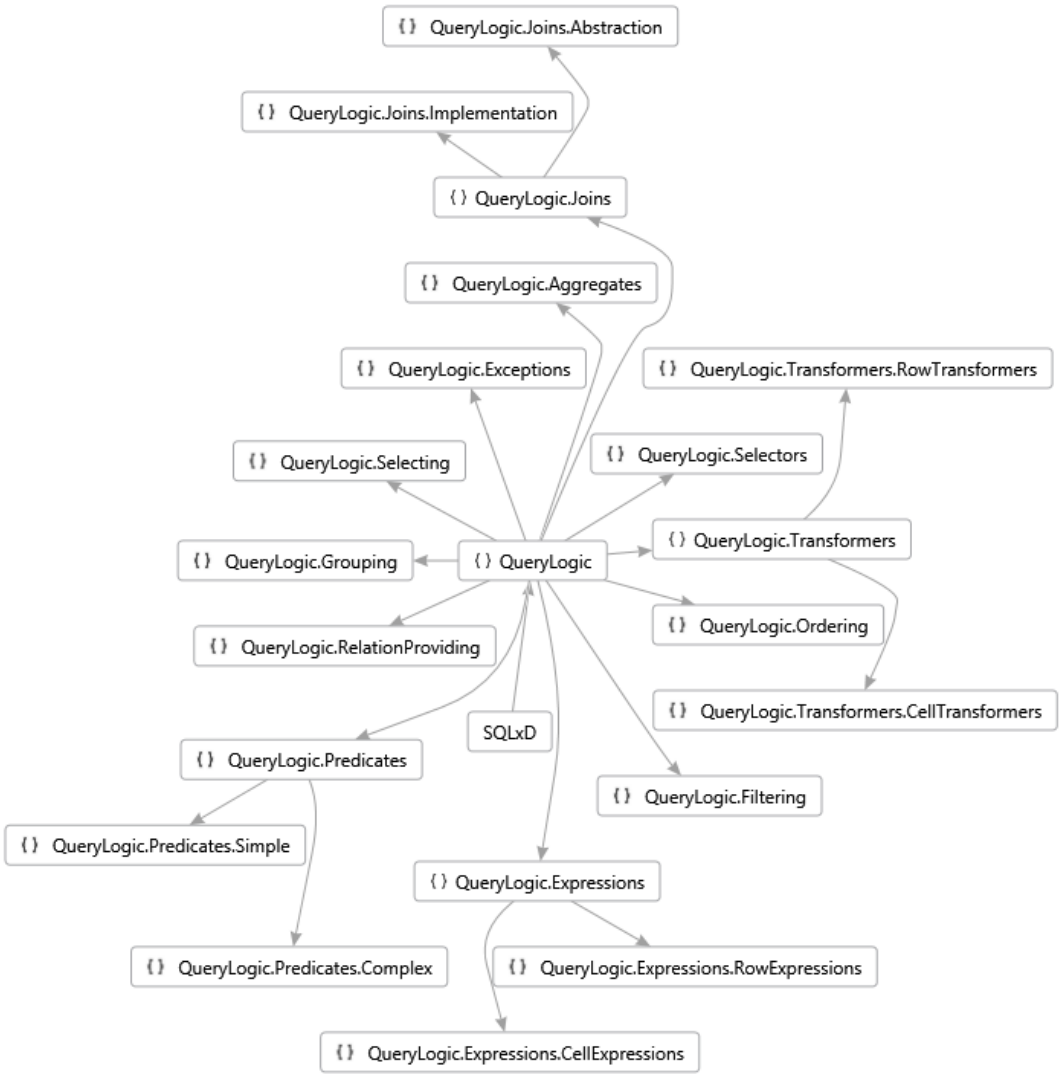
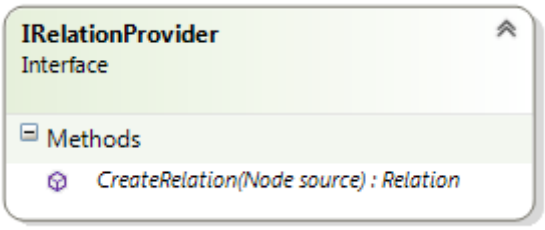



Diagram architektury modułu *QueryLogic*

### *RelationProviding*

Pakiet zawiera interfejs *IRelationProvider*, implementowany przez każdą klasę, która potrafi stworzyć relację.




Interfejs *IRelationProvider*

Kolejną klasą zawartą w pakiecie jest *From*. Odpowiada ona za reprezentację klauzuli FROM `selektor AS alias`, która jest elementem początkowym każdego zapytania SQL. W odróżnieniu

od klauzuli *FROM* ze standardowego języka *SQL*, nadawanie aliasów jest wymagane, ze względu na sposób adresowania kolumn. Zostało to dokładnie opisane w pracy magisterskiej, na której opiera się nasza implementacja.

Najważniejszym zadaniem klasy *From* jest wybranie odpowiednich węzłów źródłowego dokumentu *XML* i transformowanie otrzymanego zbioru do postaci relacyjnej. Do wyboru węzłów dokumentu używany jest mechanizm selektorów, opisany w kolejnym akapicie.

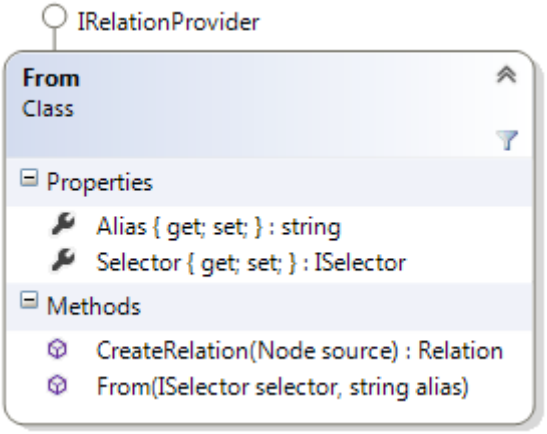


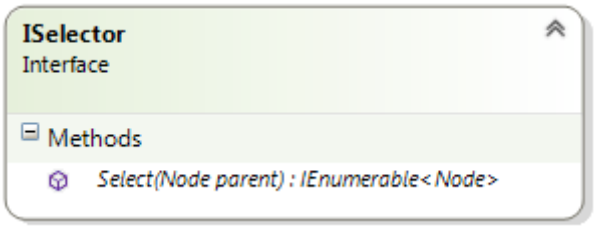

Diagram klasy *From*

### *Selectors*

W pakiecie znajduję się interfejs *ISelector* oraz klasy go implementujące. Selektory reprezentują adresację węzłów w dokumencie *XML* i są używane w klauzuli *FROM*. Sama idea oraz teoretyczny opis adresacji został opisany w pracy magisterskiej będącej podstawą implementacji.

W adresacji możemy wybrać element dokumentu poprzez jego adres, którego początek jest w korzeniu dokumentu. Możemy również wybrać wszystkie węzły z któregoś z poziomów w hierarchii dokumentu *XML* lub z wszystkich poziomów zaczynających się od wyznaczonego węzła.

Mechanizm selektorów umożliwia budowę każdego dopuszczalnego adresu w dokumencie za pomocą łączenia kolejnych elementów adresu.




Interfejs *ISelector*

### *Selecting*

Pakiet zawiera klasę *Select*, odpowiadającą za reprezentację klauzuli `SELECT wyrażenie (...) GROUP BY kolumny ORDER BY kolumny` z języka *SQL*. *Select* umożliwia projekcję istniejącej relacji. Jeśli o klauzuli *FROM* możemy powiedzieć, że jest ona punktem wejściowym i właściwie służy do ładowania danych, to klauzula *SELECT* jest punktem wyjściowym, ostatnią klauzulą wykonywaną podczas przetwarzania zapytania *SQL*. *Select* przyjmuję relację źródłową, wykonuję na nie wszystkie wyrażenia sprecyzowane w zapytaniu i zwraca relację wynikową zawierającą określone kolumny. W szczególności wyrażenia mogą być po prostu nazwami kolumn, funkcjami wykonywanymi na

wartościach w kolejnych komórkach lub agregatami. Relacja wynikowa w zależności od zapytania może być posortowania i pogrupowana.

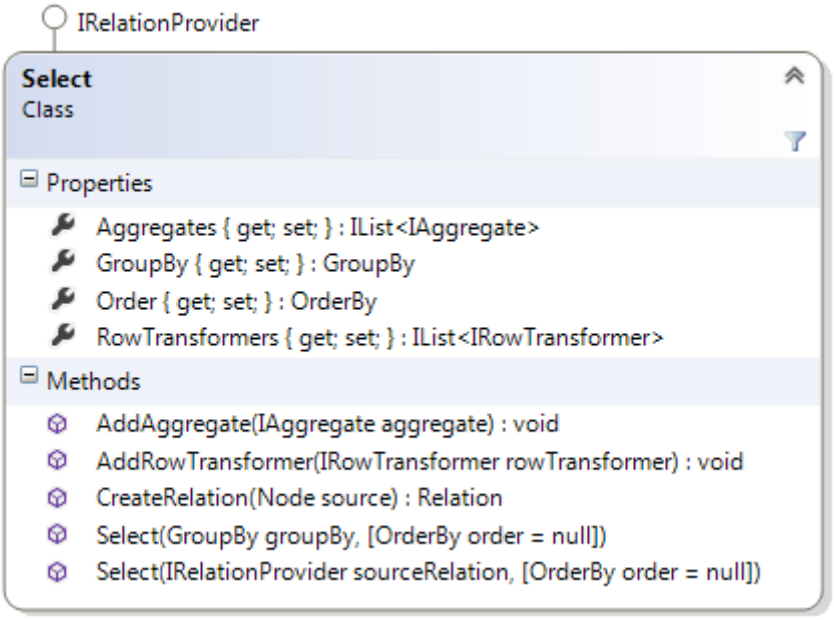
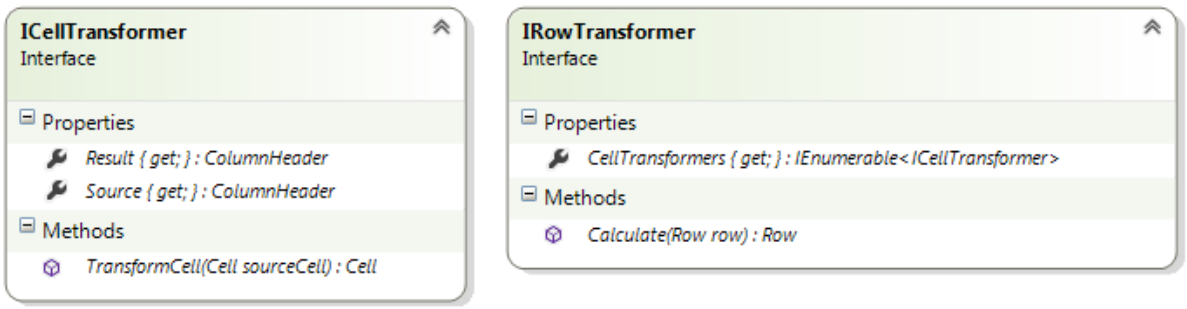



Diagram klasy *Select*

### *Transformers*

Pakiet *Transformers* dzieli się na dwa mniejsze pakiety: *CellTransformers* oraz *RowTransformers*, zawiera on mechanizmy, używane podczas konstrukcji klauzuli SELECT. Jest to abstrakcja umożliwiająca wybieranie określonych kolumn z wiersza relacji (*RowTransformers*) oraz wykonywanie funkcji na wartościach wybranych komórek (*CellTransformers*). Pakiet jest ściśle powiązany z pakietem *Expressions*, który zawiera implementację wyrażeń wykonywanych przez przekształcenia na poszczególnych wierszach.

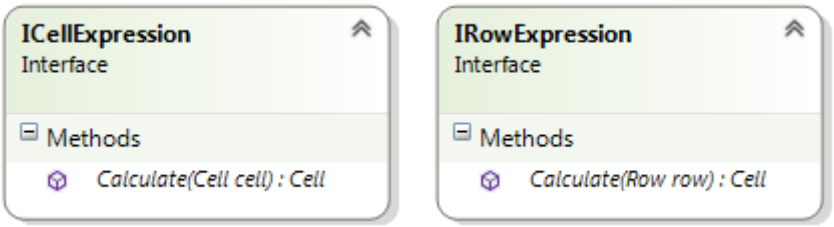



Interfejsy *ICellTransformer* i *IRowTransformer*

### *Expressions*

Pakiet, podobnie jak *Transformers*, dzieli się na dwa mniejsze: *CellExpressions*, oraz *RowExpressions*. Wyrażeniem w przypadku *CellExpressions* nazywamy każdą operację, która na podstawie wybranej komórki z wiersza zwraca nową komórkę. W szczególności wyrażenie może zwrócić komórkę nie zmieniając jej wartości lub wykonać na niej jakaś funkcję (na przykład `LENGTH(komórka)`).

W przypadku *RowExpressions*, wyrażenie to operacja, która zwraca komórkę na podstawie wiersza. Wyrażenia tego typu są używane do wybierania określonych komórek z wiersza i mogą wykonywać na nich wyrażenia typu *CellExpressions*.




Interfejsy *ICellExpression* i *IRowExpression*

### *Filtering*

Pakiet *Filtering* zawiera klasę *Where*, odpowiadającą za reprezentację klauzuli `WHERE predykat` z języka *SQL*. Logika umożliwia eliminacje wierszy z dostarczonej relacji za pomocą określonych predykatów, w wyniku czego zwrócona zostaję odpowiednio odfiltrowana relacja.

*Where* korzysta ze stworzonego przez nas mechanizmu predykatów, pozwalającego na sprawdzanie określonych warunków oraz porównywanie komórek w wierszach. Po wykonaniu dostarczonych predykatów na wierszu, *Where* decyduje, czy wiersz powinien zostać wyeliminowany z relacji.

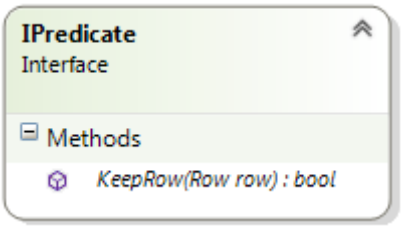
### *Predicates*

Pakiet dostarcza interfejs *IPredicate*, używany w klauzuli `WHERE`, oraz implementację różnych znanych predykatów. Predykat wykonuję określone wyrażenia na porównywanych wierszach i porównuje ich wartości. Pakiet dzieli się na dwa mniejsze pakiety: *Simple* i *Complex*.

Pakiet *Simple* zawiera klasy implementujące różne podstawowe predykaty. W szczególności są to: porównanie (=), mniejszość (<), większość (>), mniejszość lub równość (<=), większość lub równość (>=), sprawdzenie czy jest wartością *NULL* (`IS NULL`) i podobieństwo łańcuchów znakowych (`LIKE`).

Predykat LIKE obsługuje opisane w standardzie *SQL/92* [1] znaki specjalne `_` *oraz* `%`, umożliwiające odpowiednio dopasowanie jednego znaku i dopasowanie dowolnych znaków. Obsługuje również „znak ucieczki” (ang. *escape character*) oraz grupy `[]` działające analogicznie, jak w implementacji serwera *MS SQL2008 R2* [10].

Pakiet *Complex* służy do łączenia predykatów w określony sposób. Za jego pomocą możemy otrzymać alternatywę (OR) lub koniunkcję (AND) podanych predykatów. Umożliwia to konstrukcję bardzo złożonych predykatów przy użyciu jednolitego interfejsu.

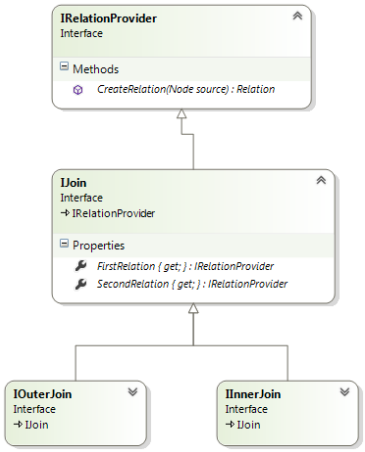



Interfejs *IPredicate*

### *Joins*

Pakiet zawiera klasy odpowiadające za implementację różnych typów klauzul `JOIN warunek` z języka *SQL*. Obecnie zaimplementowane zostały następujące typy złączeń: *CROSS JOIN, FULL OUTER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, INNER JOIN* oraz specyficzny dla silnika *NATURAL JOIN*.

Najbardziej bazową operacją złączenia w projekcie jest *CROSS JOIN*, który tworzy iloczyn kartezjański na podstawie dwóch relacji. Każda z pozostałych implementacji operacji *JOIN*, wyłączając *NATURAL JOIN*, wykorzystuję operację *CROSS JOIN*, później w określony sposób eliminując wiersze za pomocą klasy *Where* i dodając wiersze w przypadku złączeń zewnętrznych.




**Interfejsy *IJoin*, *IOuterJoin* i *IInerJoin***

Aby wytłumaczyć mechanikę działania każdego złączenia, najłatwiej będzie posłużyć się przykładami. Trzeba jednak pamiętać, że służą one tutaj jedynie do łatwiejszego zrozumienia sposobu działania i nie są pełnym opisem implementacji.

Tak więc zapytanie `SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.id` jest logicznie wykonywane jako `SELECT * FROM t1 CROSS JOIN t2 WHERE t1.id = t2.id`.

Złączenie `SELECT * FROM t1 LEFT JOIN t2 ON t1.id = t2.id` jest logicznie wykonywane jako `SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.id UNION SELECT * FROM t1 WHERE NOT EXISTS (SELECT 1 FROM t2 WHERE t1.id = t2.id)`. Powyższe zapytanie nie jest poprawne, gdyż zestawy kolumn są niezgodne – należy je interpretować jako wybranie najpierw wierszy spełniających dopasowanie, a następnie dodanie wierszy z tabeli `t1` nie mających dopasowania.

Złączenia *RIGHT JOIN* oraz *FULL JOIN* są wykonywane analogicznie.

*NATURAL JOIN* jest operacją specyficzną dla silnika *SQLxD*, teoretyczna podstawa implementacji klauzuli została dokładnie opisana w pracy magisterskiej, na której się opieraliśmy. Do reprezentowania hierarchii przodek-potomek wykorzystujemy identyfikatory *GUID* przypisane węzłom dokumentu *XML*, a następnie przekazywane w relacjach jako *GuidCell*.

### *Grouping*

Pakiet *Grouping* zawiera klasy odpowiadające za implementację klauzuli `GROUP BY wyrażenie` z języka *SQL*.

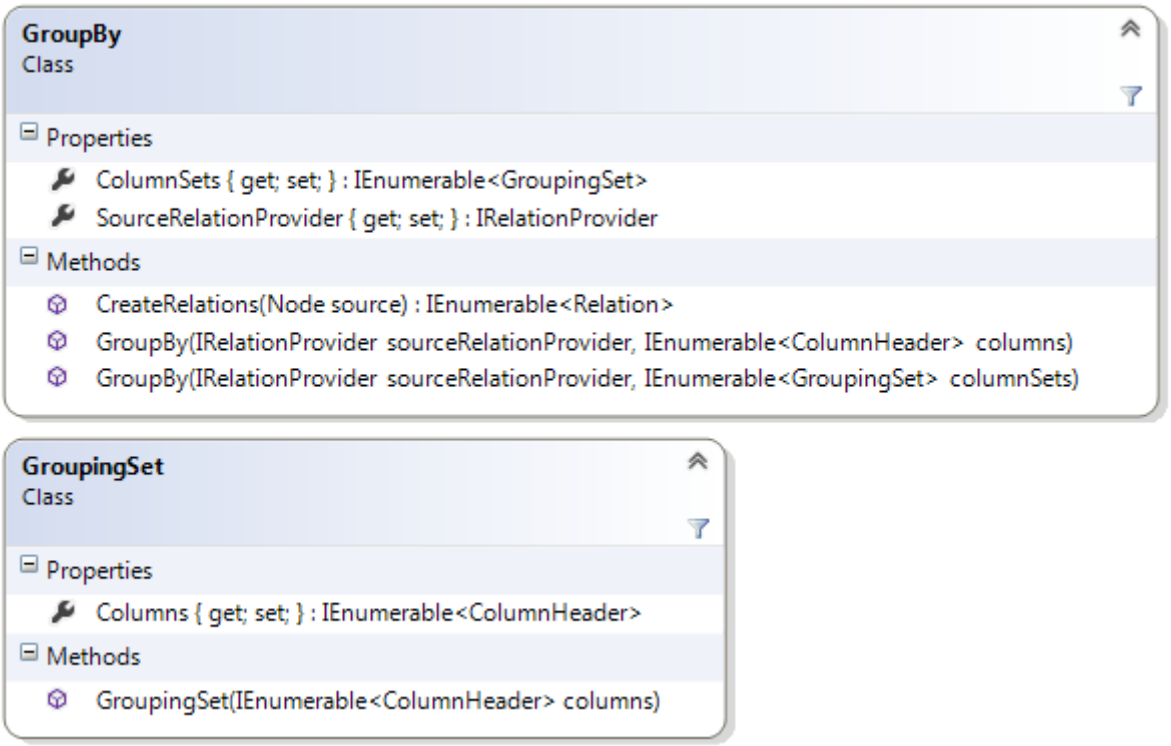
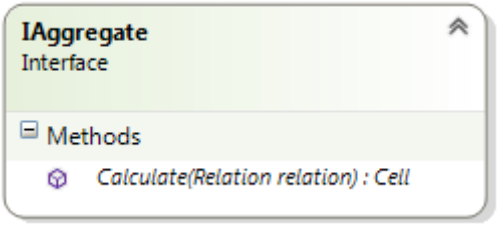



**Diagram klasy *GroupBy* i klasy *GroupingSet***

### *Aggregates*

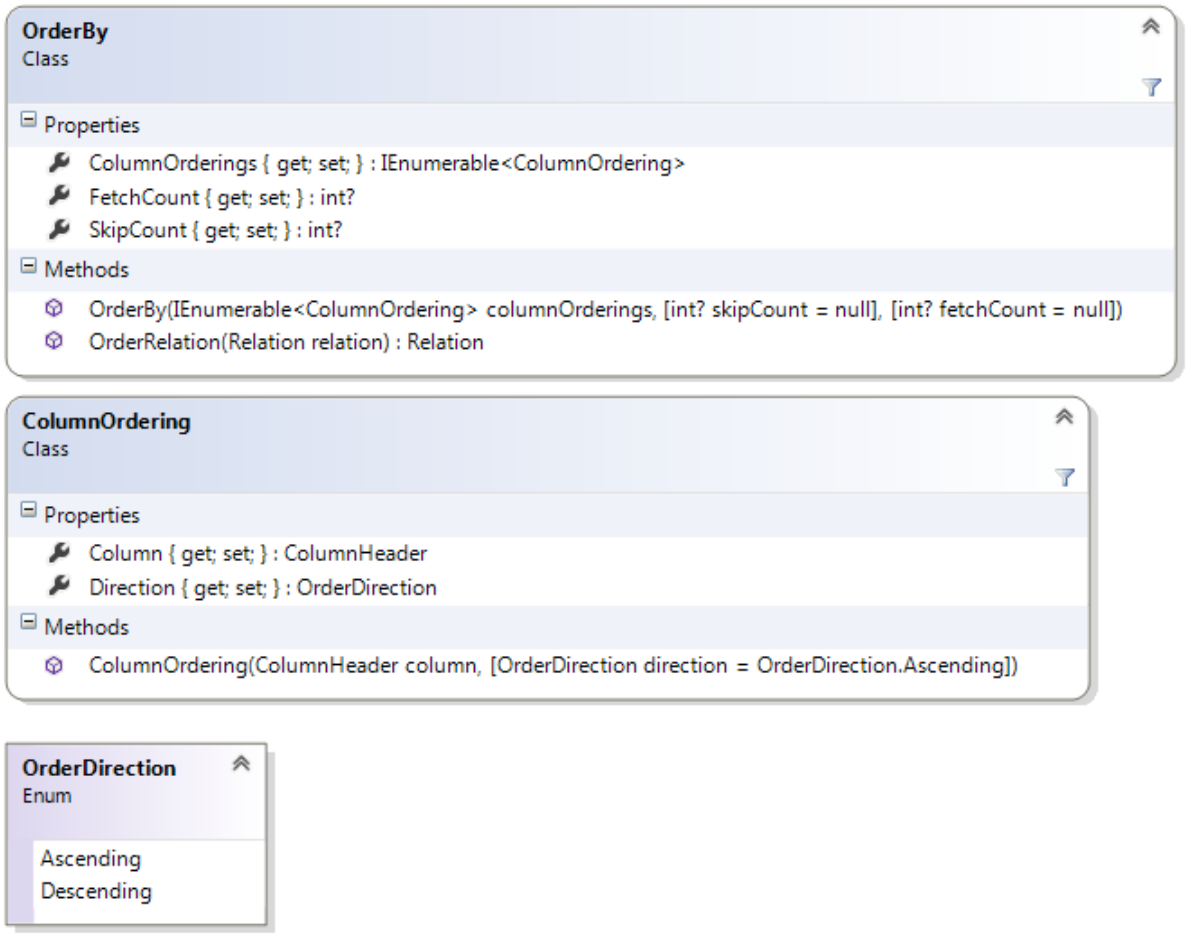
Pakiet *Aggregates* zawiera klasy odpowiadające za implementację różnych funkcji agregujących znanych z języka *SQL*. Funkcje agregujące wykonywane są na całych relacjach, więc ich interfejs oraz implementacja, pomimo zbieżności nazw, znacznie się różnią od zwykłych funkcji opisanych wcześniej. Obecnie zaimplementowane są agregaty: *MIN, MAX, SUM, COUNT* oraz *AVG*, ich działanie jest zgodne z agregatami dostępnymi w typowych silnikach bazodanowych.




**Interfejs *IAggregate***

### *Ordering*

Pakiet *Ordering* zawiera klasy odpowiadające za implementację klauzuli `ORDER BY kolumna` z języka *SQL*.




Diagramy klas: *OrderBy*, *ColumnOrdering* i *enum OrderDirection*

### *Exceptions*

Pakiet zawiera wyjątki, które mogą zostać rzucone przez algorytmy zawarte w module *QueryLogic*.

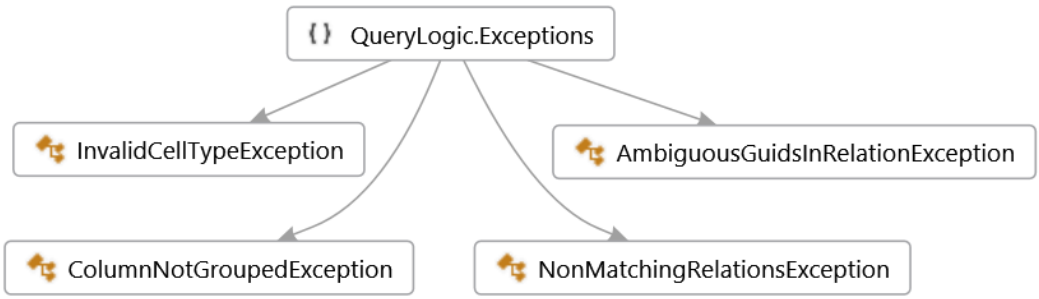



Diagram architektury: *QueryLogic.Exceptions*

## QueryParser

### *Wstęp*

Moduł zawiera logikę służącą do zamiany tekstowej reprezentacji zapytania *SQL* na model używany w projekcie. Parser podczas działania tworzy złożone modele wyrażeń używając klas z modułu

*QueryLogic*. Wyrażenie będące wynikiem działania parsera zapytań może być wykonane na modelu dokumentu *XML* dostarczonym przez parser *XML* opisany wcześniej.

Po przeglądnięciu różnych dostępnych bibliotek do parsowania, okazało się, że *C#* nie posiada biblioteki, która sprostała by naszym wymaganiom. Postanowiliśmy więc, że moduł parsera zapytań zostanie napisany w języku *Python 2.7.3* [11], z użyciem biblioteki *PLY* [12]. Większość naszej implementacji silnika *SQLxD* jest napisana w języku *C#*, jednak z pomocą platformy *.NET* oraz biblioteki *IronPython* [13] udało nam się w prosty sposób połączyć te technologie.

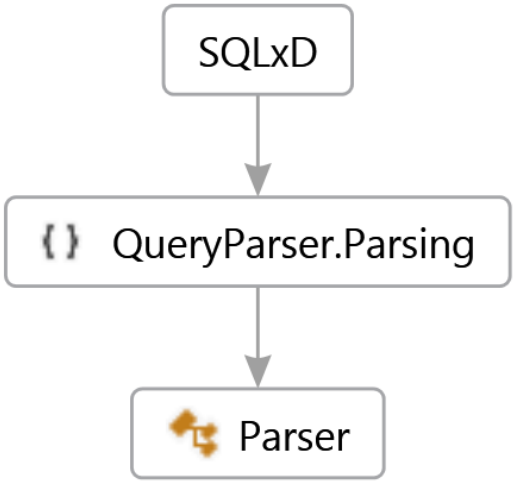



Diagram architektury modułu *QueryParser*

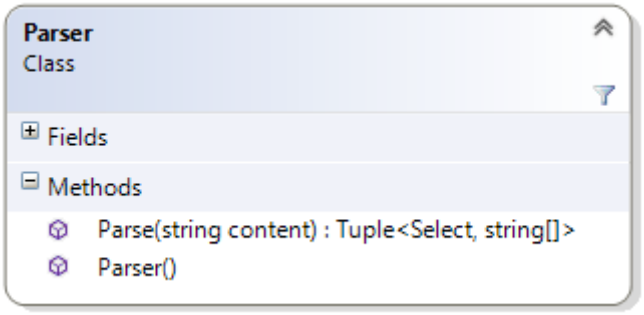



Diagram klasy *Parser*

### *Gramatyka*

Za wyjściową gramatykę przyjęliśmy gramatykę standardu *SQL/92* [1]. Oryginalna gramatyka została dołączona w punkcie Dodatek A. Oryginalna gramatyka SQL 92.

Ze względu na specyficzne cechy silnika *SQLxD*, oryginalna gramatyka musiała zostać zmodyfikowana. Zmiany dotyczyły głównie *NATURAL JOIN*, nadawania aliasów tabel, wybierania kolumn oraz tabel i sortowania.

Gramatyka robocza została dołączona w punkcie Dodatek B. Gramatyka robocza.

# Technologia

Implementacja silnika *SQLxD* wymagała użycie przede wszystkim bibliotek do parsowania: dokumentów *XML*, oraz samej składni języka *SQL*. Dodatkowo używane były biblioteki do testowania. Resztę narzędzi dostarczyła sama platforma *.NET* oraz używane języki programowania.

Wszystkie wykorzystane biblioteki i formaty są udostępnione na zasadzie *Open Source* (ang. „otwarty kod") albo można z nich korzystać za darmo w ramach projektów niekomercyjnych.

## Języki

### C# 5.0

Jednym z wymagań niefunkcjonalnych było utworzenie projektu na platformę *.NET*. Język *C#* jest w tym przypadku naturalnym wyborem jako wydajny, silnie typowany język z dużą biblioteką standardową i wsparciem dla wielu potrzebnych mechanizmów.

### IronPython 2.7

Język *Python* [11] wraz z biblioteką *PLY* [12] został użyty do implementacji parsera języka *SQL*. Motywacją była łatwość użycia, oraz wysoka funkcjonalność dostarczona przez narzędzia.

## Biblioteki

### PLY (Python-Lex-Yacc) 3.4

http://www.dabeaz.com/ply/

1. Biblioteka dostarczająca implementacje narzędzi do parsowania *lex/yacc*
2. Cel: używana do parsowania zapytań *SQL* i tworzenia modelu zapytania używanego w silniku
3. Użyta ze względu na: funkcjonalność, wydajność, znajomość
4. Alternatywa: *Antlr* [14], *GPPG* [15], *Irony* [16]
5. Licencja: *BSD*

### RhinoMocks

http://www.hibernatingrhinos.com/oss/rhino-mocks

1. Framework do tworzenia obiektów atrap do testów jednostkowych
2. Cel: implementacja testów
3. Użyta ze względu na: funkcjonalność, znajomość
4. Alternatywa: *Moq* [17], *Typemock Isolator* [18]
5. Licencja: *BSD*

### NUnit

http://www.nunit.org/

1. Framework do testów jednostkowych dla platformy *.NET*
2. Cel: implementacja testów
3. Użyta ze względu na: funkcjonalność, znajomość
4. Alternatywa: *xUnit.net* [19], *MSTest*
5. Licencja: *BSD-style* (zmodyfikowana licencja *zlib*)

## Formaty danych

### XML .xml

1. Zastosowanie: format przechowywania danych w bazie danych *SQLxD*
2. Wymagania: *.NET Framework 4.5*

# Testy

Zaczynając projekt ustaliliśmy, że w celu zapewnienia wysokiej jakości produktu oraz dokumentacji, każdą dodaną funkcjonalność pokryjemy testami jednostkowymi. Natomiast nie ustalaliśmy czasu pisania testów, część projektu była pisana z użyciem *Test Driven Development*, część testów zostało napisane po implementacji właściwej funkcjonalności.

Dodatkowo w celu przetestowania całości biblioteki dodane zostały testy integracyjne, zarówno do samego modułu *QueryLogic*, jak i po dołączeniu parsera zapytań języka *SQL*.

Do tej pory wszystkie testy zostały zautomatyzowane i mogą być uruchomione z poziomu *Visual Studio* (przy użyciu mechanizmu dostarczonego przez dodatek *ReSharper*) lub przez *NUnit-Runner*.

## Testy jednostkowe

Na chwile obecną projekt posiada około 500 testów jednostkowych, do wszystkich 4 modułów projektu. Testy zostały napisane przy użyciu biblioteki *NUnit* [20]. Do izolacji zależności użyta została biblioteka *RhinoMocks* [21] i ręcznie implementowane atrapy.

Przyjęliśmy następującą konwencję:

- Dla każdego modułu dostarczającego funkcjonalność w projekcie dodany został moduł o takiej samej nazwie z końcówką `.Test` zawierający testy jednostkowe do klas modułu
- Nazwa klasy testowej składa się z nazwy klasy testowanej z końcówką `Tests`
- Nazwa metody testowej jest w formacie `Akcja_WarunkiPoczątkowe_SpodziewanyRezultat`
- Klasy testowe mają kwalifikator dostępu `internal`
- Konwencja *Arrange-Act-Assert* (ang. „zbuduj-zadziałaj-zweryfikuj”)

## Testy integracyjne

Po zaimplementowaniu znacznej części bazowej funkcjonalności projektu dodane zostały testy integracyjne, mające na celu sprawdzenie poprawności implementacji silnika w większej skali.

Do testów stworzony został osobny projekt *IntegrationTest*, zawierający testy oraz większą ilość danych testowych. Testy utrzymane są w podobnej konwencji do testów jednostkowych, lecz w przypadku testów integracyjnych nie używamy obiektów – atrap.

# Składnia zapytania SQL

## Podstawy teoretyczne

Jako wyjściową gramatykę przyjęliśmy gramatykę ze standardu *SQL/92* [8], jednak w trakcie implementacji musieliśmy dokonać pewnych modyfikacji, aby dostosować język do specyficznych potrzeb naszego silnika. Dołożyliśmy starań, aby zmiany te były jak najmniejsze, tak aby osoba znająca język *SQL* potrafiła bez problemów korzystać z naszego silnika.

## Intuicyjny opis gramatyki

Możliwe do wykonania zapytanie wygląda następująco:

```
SELECT
    *
    | lista_kolumn_funkcji_agregatow
FROM selektor AS nazwa
{
    [ NATURAL JOIN selektor AS nazwa ]
    [ INNER JOIN selektor AS nazwa ON warunek ]
    [ LEFT [ OUTER ] JOIN selektor AS nazwa ON warunek ]
    [ RIGHT [ OUTER ] JOIN selektor AS nazwa ON warunek ]
    [ FULL [ OUTER ] JOIN warunek AS nazwa ON warunek ]
    [ CROSS JOIN selektor AS nazwa ]
}
[ WHERE warunek ]
[ GROUP BY lista_kolumn ]
[ ORDER BY lista_kolumn [ DESC ] [ SKIP liczba_całkowita ] [ FETCH
liczba_całkowita ] ]
```

Gdzie:

- *lista_kolumn_funkcji_agregatow* oznacza listę kolumn, funkcji, agregatów lub ich złożeń, oddzielonych przecinkami, opcjonalnie posiadających aliasy
- *lista_kolumn* oznacza listę kolumn oddzielonych przecinkami
- *selektor* oznacza mechanizm podawania źródła opisany w Mechanizm selektorów
- *nazwa* oznacza łańcuch znaków składający się z płotka, litery (małej lub dużej), podkreślenia lub cyfry, przy czym nierozpoczynający się cyfrą
- *warunek* oznacza złożenie dowolny warunek prosty lub złożony. Warunek prosty oznacza porównania <, >, <=, >=, =, <> oraz predykat LIKE, warunek złożony oznacza koniunkcję lub alternatywę warunków prostych lub złożonych
- *liczba_całkowita* oznacza dowolną liczbę całkowitą
- Nawias kwadratowy oznacza opcjonalne jednokrotne wystąpienie
- Nawias klamrowy oznacza opcjonalne wielokrotne wystąpienie
- Kreska pionowa oznacza alternatywę.

Opis ten należy rozumieć następująco:

- Zapytanie *SELECT* może pobrać wszystkie kolumny (gwiazdka) lub podaną listę kolumn, funkcji i agregatów
- Zapytanie musi mieć źródło (klauzula *FROM*)
- Zapytanie może zawierać złączenia naturalne, wewnętrzne, zewnętrzne (lewostronne, prawostronne, obustronne) i iloczyn kartezjański. Mogą być one podane dowolną liczbę razy i w dowolnej kolejności.
- Zapytanie może posiadać warunki

- Zapytanie może posiadać klauzulę grupującą według podanej listy kolumn
- Zapytanie może posiadać klauzulę sortującą według podanej listy kolumn
- Sortowanie może sortować w kolejności rosnącej lub malejącej
- Sortowanie może spowodować pominięcie pierwszych wierszy wyniku
- Sortowanie może spowodować pobranie jedynie pierwszych wierszy wyniku

## Różnice w stosunku do gramatyki SQL/92

### Różnice w złączeniu naturalnym

Bazowa gramatyka pozwala na wykonanie złączenia `NATURAL INNER JOIN`, w naszym silniku złączenie naturalne jest zupełnie innym typem złączenia, dlatego możliwe jest wykonanie `NATURAL JOIN`.

### Nadawanie aliasów źródłom

W bazowej gramatyce nadawanie aliasów tabelom źródłowym (używanym w klauzuli *FROM* i klauzulach złączeń) jest opcjonalne, w naszej gramatyce jest wymagane.

### Podawanie kolumny

Bazowa gramatyka nie wymaga, aby poprzedzać nazwę kolumny nazwą schematu (źródła), jeżeli nie prowadzi to do niejasności. W naszej gramatyce jest to wymagane.

### Podawanie źródła

Bazowa gramatyka dopuszcza jako źródło podanie nazwy tabeli lub nazwy tabeli wraz z nazwą schematu, w naszej gramatyce możliwe jest podawanie całej ścieżki do węzła oraz używanie dwóch znaków specjalnych: pytajnika (oznaczającego jeden poziom) i gwiazdki (oznaczającej dowolną liczbę poziomów).

### Zmiany w sortowaniu

Nasza gramatyka dopuszcza konstrukcję `[ SKIP liczba_całkowita ] [ FETCH liczba_całkowita ]`

## Opis implementacji

Szczegóły implementacyjne zostaną omówione na przykładowym zapytaniu zawierającym wszystkie elementy języka, dzięki czemu widoczny będzie przepływ danych i sposób budowania logicznej reprezentacji kwerendy *SQL*.

### Dane

W przykładzie zostanie wykorzystany dokument, którego pełna treść znajduje się w Dodatek C. Przykładowy dokument XML.

### Zapytanie

Wykorzystane zapytanie prezentuje się następująco:

```
SELECT
    p.postac.#imie AS imie,
    p.postac.#id,
    s.statystyki.#atak,
    k.kraj.#nazwa,
    m.miasto.#nazwa,
    r.rzecz.#nazwa,
    a.artefakt.#nazwa,
    c.cel.#nazwa,
    SUBSTR(YEAR(m.miasto.#data), 2, 2),
    MONTH(m.miasto.#data),
    DAY(m.miasto.#data),
    HOUR(m.miasto.#data),
    MINUTE(m.miasto.#data),
    SECOND(m.miasto.#data),
    TRIM(COUNT(p.postac.#imie)) AS liczbaImion,
    MAX(s.statystyki.#atak) AS maksymalnyAtak,
    MIN(s.statystyki.#atak) AS minimalnyAtak,
    AVG(s.statystyki.#atak) AS sredniAtak,
    SUM(s.statystyki.#atak) AS sumaAtaku,
FROM
    dane.?.postac AS p
NATURAL JOIN
    dane.*.statystyki AS s
INNER JOIN
    dane.kraje.kraj AS k
    ON p.postac.#id = k.kraj.#idPostaci
LEFT OUTER JOIN
    dane.miasta.miasto AS m
    ON m.miasto.#idPostaci = p.postac.#id
RIGHT OUTER JOIN
    dane.rzeczy.rzecz AS r
    ON r.rzecz.#idPostaci = p.postac.#id OR p.postac.#id IS NULL
FULL OUTER JOIN
    dane.artefakty.artefakt AS a
    ON a.artefakt.#idPostaci = p.postac.#id OR p.postac.#id IS NULL
CROSS JOIN
    dane.cele.cel AS c
WHERE
    (p.postac.#imie like '%' AND p.postac.#id > 0) OR (p.postac.#id IS
NULL)
GROUP BY
    p.postac.#imie,
    p.postac.#id,
    s.statystyki.#atak,
    k.kraj.#nazwa,
```

```
    m.miasto.#nazwa,
    r.rzecz.#nazwa,
    a.artefakt.#nazwa,
    c.cel.#nazwa,
    m.miasto.#data
ORDER BY
    p.postac.#id DESC
SKIP 2 FETCH 3
```

## Opis działania

### Parsowanie dokumentu

Pierwszym krokiem jest parsowanie dokumentu *XML*. Odpowiada za to klasa *XmlDocumentParser* dostarczająca metodę *Parse* przyjmującą treść dokumentu. W wyniku parsowania zwracany jest obiekt klasy *Node*, który reprezentuje główny węzeł dokumentu *XML*.

Klasa *XmlDocumentParser* analizuje kolejne węzły dokumentu *XML*, dla każdego z nich tworząc odpowiedni obiekt klasy *Node*. Do reprezentacji dokumentu wykorzystywana jest klasa *XDocument*.

### Parsowanie zapytania

Za parsowanie zapytania odpowiada klasa *Parser*, która dostarcza metodę *Parse* przyjmującą treść zapytania. Metoda ta zwraca parę składającą się z obiektu klasy *Select* oraz tablicy napisów – pierwszy obiekt reprezentuje właściwy model logiczny zapytania, tablica napisów przechowuje ewentualne błędy napotkane w trakcie parsowania zapytania.

Klasa *Select* jest sercem całego zapytania – zawiera instancje klas *GroupBy*, *OrderBy*, listę *IRowTransformer* oraz *IAggregate*, Klasa Select zawiera również metodę *CreateRelation*, która zwraca obiekt klasy *Relation* – jest to obiekt zawierający rezultat zapytania (obiekty klasy *Row*, które zawierają obiekty klasy *Cell*).

Samo parsowanie zapytania opiera się o bibliotekę *PLY* [12], która dostarcza skanera oraz parsera *LALR* gramatyki SQL/92 [8] (zmodyfikowanej do naszych potrzeb).

### Utworzone zapytanie

Po sparsowaniu zapytania uzyskany obiekt klasy *Select* zawiera wszystkie informacje niezbędne do wyłuskania danych z modelu – wykonanie zapytania rozpoczyna się w momencie wywołania metody *CreateRelation* z odpowiednim parametrem, który jest obiektem klasy *Node*.

Wykonanie zapytania odbywa się w kilku etapach:

Dla zapytania z agregatami:

1. Utworzenie obiektów klasy *Relation* dla każdej relacji utworzonej przy pomocy *GroupBy.*
2. Dodanie pojedynczych wartości skalarnych dla agregatów.
3. Dodanie pojedynczych wartości skalarnych dla kolumn nieagregowanych.
4. Zwrócenie wynikowej relacji.

Dla zapytania bez agregatów:

1. Utworzenie obiektów klasy *Relation* dla każdej relacji utworzonej przy pomocy *GroupBy*.
2. Dodanie pojedynczych wartości skalarnych dla kolumn.
3. Zwrócenie wynikowej relacji.

### Wykonanie GroupBy

Utworzony obiekt klasy *GroupBy* przedstawia stos poszczególnych transformacji relacji wejściowej – każdy z obiektów stosu implementuje interfejs *IRelationProvider*, dzięki czemu posiada metodę *CreateRelation* zwracającą relację. *GroupBy* zawiera również listę kolumn, po których powinien zgrupować uzyskaną relację. Przy wywoływaniu metody *CreateRelations* obiektu klasy *GroupBy* zachodzą dwie czynności:

1. *GroupBy* tworzy relację przy użyciu obiektu *IRelationProvider*.
2. *GroupBy* grupuje relację na podstawie listy kolumn i zwraca kolekcję obiektów klasy *Relation* (każdy obiekt reprezentuje wiersze dla innej krotki kolumn grupujących).

### Mechanizm złączeń

Jak już zostało wspomniane wcześniej, złączenia opierają się na połączeniu logiki *CrossJoin* oraz *Where*, więc de facto wyłuskiwane dane przemierzą następującą drogę:

1. `From dane.?.postac` – przy użyciu mechanizmu selektorów zostaną wybrane węzły *postac*.
2. `From dane.*.statystyki` – przy użyciu mechanizmu selektorów zostaną wybrane węzły *statystyki*.
3. *CrossJoin* danych z punktów 1 oraz 2 – zostanie utworzony iloczyn kartezjański wszystkich wierszy.
4. *Where* – nastąpi wybranie jedynie wierszy spełniających warunki złączenia naturalnego (szczegóły implementacyjne tego rodzaju złączenia są opisane w dalszej części).
5. `From dane.kraje.kraj` – przy użyciu mechanizmu selektorów zostaną wybrane węzły *kraj*.
6. *CrossJoin* danych z punktów 4 oraz 5 – zostanie utworzony iloczyn kartezjański wszystkich wierszy.
7. `Where p.postac.#id = k.kraj.#idPostaci` – zostaną wybrane jedynie wiersze, w których wartości *#id* oraz *#idPostaci* są równe.
8. `From dane.miasta.miasto` – przy użyciu mechanizmu selektorów zostaną wybrane węzły *miasto*.
9. *CrossJoin* danych z punktów 7 oraz 8 – zostanie utworzony iloczyn kartezjański wszystkich wierszy.
10. `Where m.miasto.#idPostaci = p.postac.#id` – zostaną wybrane jedynie wiersze spełniające warunek.
11. Zostaną dodane wiersze, dla których nie zostało znalezione dopasowanie (aby uzyskać logicznie wykonanie złączenia zewnętrznego).
12. `From dane.rzeczy.rzecz` –przy użyciu mechanizmu selektorów zostaną wybrane węzły *rzecz*.
13. *CrossJoin* danych z punktów 11 oraz 12 – zostanie utworzony iloczyn kartezjański wszystkich wierszy.
14. `Where r.rzecz.#idPostaci = p.postac.#id OR p.postac.#id IS NULL` – zostaną wybrane jedynie wiersze spełniające warunek.
15. Zostaną dodane wiersze, dla których nie zostało znalezione dopasowanie (w celu uzyskania złączenia zewnętrznego).
16. `From dane.artefakty.artefakt` – przy użyciu mechanizmu selektorów zostaną wybrane węzły *artefakt*.
17. *CrossJoin* danych z punktów 15 i 16 – zostanie utworzony iloczyn kartezjański wszystkich wierszy.
18. `Where a.artefakt.#idPostaci = p.postac.#id OR p.postac.#id IS NULL` – zostaną wybrane jedynie wiersze spełniające warunek.
19. Zostaną dodane wiersze, dla których nie znaleziono dopasowania.

20. `From dane.cele.cel` – przy użyciu mechanizmu selektorów zostaną wybrane węzły cel.
21. *CrossJoin* danych z punktów 19 i 20.
22. `Where (p.postac.#imie like '%' AND p.postac.#id > 0) OR (p.postac.#id IS NULL)` – zostaną wybrane jedynie wiersze spełniające podane warunki.

W tym momencie dostępny jest już obiekt klasy *Relation* zawierający dokładnie te dane, które spełniły wszystkie warunki złączeń i filtrowania.

### Mechanizm selektorów

Logika klasy *From* wykorzystuje mechanizm selektorów do wybierania węzłów. Klasa ta posiada metodę *CreateRelation* przyjmującą parametr typu *Node* – węzeł główny dokumentu, z którego zostaną wybrane węzły. Selektory opierają się o następujące elementy budulcowe:

- *TopLevelSelector* – próbuje dopasować węzeł główny dokumentu
- *NodeSelector* – próbuje dopasować węzeł potomny po nazwie
- *LevelSelector* – dopasowuje dowolny węzeł na jednym poziomie zagnieżdżenia
- *AnySelector* – dopasowuje dowolny węzeł na dowolnym poziomie zagnieżdżenia
- *ChainedSelector* – łączy dwa selektory

Przykładowo dla ścieżki `dane.miasta.miasto` zostanie utworzony następujący stos selektorów:

```
ChainedSelector (TopLevelSelector [dane], ChainedSelector (NodeSelector [miasta], NodeSelector [miasto]))
```

Podczas gdy dla zapytania `dane.*.statystyki` zostanie utworzony następujący stos:

```
ChainedSelector (TopLevelSelector [dane], ChainedSelector (AnySelector, NodeSelector [statystyki]))
```

Ostatecznie każdy selektor zwraca kolekcję węzłów, które znajdują się na podanej ścieżce (może to być kolekcja pusta).

### Budowanie początkowej relacji

Dokładny mechanizm budowania wierszy z danego węzła dokumentu został opisany przez Jakuba Wyrostka w jego pracy magisterskiej [7].

### Mechanizm wykonywania złączenia naturalnego

Złączenie naturalne w języku *SQL* zazwyczaj jest implementowane jako złączenie na wszystkich kolumnach o zgodnych nazwach. W naszej implementacji, zgodnie z opisem Jakuba Wyrostka, złączenie naturalne działa na zupełnie innej zasadzie – oznacza ono złączenie na podstawie relacji przodek-potomek. Oznacza to, że dwa wiersze zostaną dopasowane tylko wtedy, jeżeli zostały utworzone z węzłów, z których jeden był przodkiem drugiego.

Ponieważ w momencie utworzenia wiersza tracimy bezpośrednie połączenie z węzłem, z którego wiersz powstał, musimy w jakiś sposób pamiętać, jakie jest jego pochodzenie. Aby to zrobić, każdy z węzłów dokumentu otrzymuje unikalny identyfikator (*GUID*) powiązany ze ścieżką od głównego węzła dokumentu do danego węzła. Następnie przy tworzeniu wiersza dodajemy mu identyfikatory węzła źródłowego i przodków tego węzła. Identyfikatory te są niewidoczne dla użytkownika, ale są przywiązane do wiersza przez cały czas jego życia.

Przy wykonywaniu złączenia naturalnego dwóch wierszy pozostaje jedynie znalezienie części wspólnej identyfikatorów obu wierszy (będą to identyfikatory wspólnych przodków), sprawdzenie,

czy na pewno zachodzi relacja przodek-potomek, a następnie sprawdzenie, czy odpowiednie identyfikatory mają identyczne wartości.

### Grupowanie

Mechanizm grupowania opiera się o funkcję *GroupBy* dostarczoną przez bibliotekę *LINQ*. Dla każdej nowej krotki wartości zbioru kolumn grupujących tworzona jest osobna relacja (aby możliwe było potem łatwe agregowanie wyników).

### Mechanizm pobierania wartości z kolumn

Mechanizm pobierania wartości z kolumn opiera się o klasę *GetCellRowExpression* – nazwa wskazuje, że jest to wyrażenie dla wiersza (*RowExpression*) pobierające pojedynczą wartość skalarną (*GetCell*). Klasa ta przekazuje wiersz do odpowiedniej implementacji *ICellExpression*, która pobiera wartość źródłową z potrzebnej kolumny, a następnie modyfikuje ją i zwraca nową wartość.

Podstawową implementacją *ICellExpression* jest *GetOriginalCellCellExpression*, która jest identycznością – pobiera wartość kolumny i zwraca ją niezmienioną.

Pozostałe implementacje służą do modyfikowania wartości – przykładowo *GetDatePartCellExpression* pobiera z komórki tylko fragment daty i przekazuje go dalej, *GetSubstringCellExpression* zwraca podciąg wartości komórki, a *CatenateStringCellExpression* służy do łączenia łańcuchów znakowych.

Kolejnym istotnym elementem budulcowym jest *ChainedCellExpression*, który działa podobnie do *ChainedSelector* z mechanizmu selektorów – wykonuje najpierw logikę z jednego wyrażenia *CellExpression*, a wynik przekazuje do drugiego.

### Mechanizm obliczania agregatów

Każdy agregat jest zbudowany z trzech elementów: zewnętrznych wyrażeń *ICellExpression*, wewnętrznych wyrażeń *ICellExpression* oraz logiki samego agregata.

Logika agregata służy do wyliczenia właściwej wartości – przykładowo oblicza maksimum dla agregata *MAX* lub sumę dla agregata *SUM*.

Jednocześnie agregat nie zawsze pracuje na oryginalnej wartości komórki – przykładowo do obliczenia sumy długości wartości potrzebne jest wyrażenie `SUM(LEN(kolumna))` – aby uzyskać długość wartości, używane są wewnętrzne wyrażenia *ICellExpression*.

Ponieważ możliwe jest również wykonanie transformacji wartości zwróconej przez agregat (przykładowo obliczenie długości lub pobranie tylko części daty), potrzebne są zewnętrzne wyrażenia *ICellExpression*.

Ostatecznie agregat wykonuje następujące działania:

1. Dla każdego wiersza wywołuje logikę wewnętrznych *ICellExpression* (może to być zwykłe pobranie wartości kolumny przy użyciu *GetOriginalCellCellExpression* lub coś bardziej skomplikowanego z użyciem *ChainedCellExpression*).
2. Dla wszystkich wartości wykonuje logikę właściwą dla agregata (na przykład obliczenie sumy).
3. Dla obliczonej wartości wykonuje logikę zewnętrznych *ICellExpression*.
4. Zwraca wynik.

### Mechanizm sortowania i ograniczania liczby wynikowych wierszy

Następnym krokiem jest scalenie wynikowych relacji w jedną (poprzez wykonanie sumy zbiorów wierszy z uwzględnieniem powtórzeń) i posortowanie wyniku. Sortowanie jest wykonywane przy użyciu metody *OrderBy* lub *OrderByDescending* z biblioteki *LINQ*.

W tym momencie pozostaje już tylko ograniczenie wynikowej liczby wierszy, co jest zaimplementowane przy użyciu metod *Skip* oraz *Take* z biblioteki *LINQ*. Po tym kroku otrzymujemy ostateczny wynik zapytania.

# Opracowanie problemów

## Semantyka węzłów o tych samych nazwach

### Opis problemu

Nasz silnik jest zdolny operować na dowolnych, poprawnych składniowo danych *XML*. Problem pojawia się, gdy węzeł o tej samej nazwie wystąpi w dwóch miejscach, w każdym oznaczając coś innego – przykładowo węzeł `Adres` może oznaczać adres domowy (miejscowość, kod pocztowy, ulica, numer domu) lub adres elektroniczny (e-mail). Analizując jedynie nazwę nie jesteśmy w stanie określić, czy dwa węzły o tych samych nazwach oznaczają to samo, ponadto użytkownik musiałby móc sprecyzować, które węzły go interesują (przykładowo przy zapytaniu `FROM *.Adres` nie wiemy, o jakie adresy chodzi).

### Proponowane rozwiązania i przyjęte do implementacji

Rozważaliśmy dwa rozwiązania: rozróżnianie węzłów lub traktowanie wszystkich jednakowo. Pierwsze rozwiązanie jest z oczywistych względów bardzo skomplikowane i wykracza poza ramy tematyczne naszej pracy. Aby zastosować je poprawnie, musielibyśmy móc analizować węzły pod kątem semantycznym, ponadto należałoby zmienić składnię zapytań, aby użytkownik mógł odwoływać się do tych węzłów, do których chce.

Dlatego też przyjęliśmy rozwiązanie drugie – traktowanie wszystkich węzłów jednakowo. W przypadku braku danych (np. nazwy ulicy dla adresu e-mail), traktujemy te wartości jako `NULL`. Nie zaburza to naszym zdaniem logiki działania zapytań, a jednocześnie przekłada odpowiedzialność za interpretację dostarczonych i uzyskanych na użytkownika.

## Sposób implementacji złączenia naturalnego

### Opis problemu

Złączenia naturalne zazwyczaj oznaczają złączenia na podstawie kolumn o odpowiadających sobie nazwach. W naszym silniku ten rodzaj złączenia ma zupełnie inne znaczenie – dopasowuje wiersze na podstawie relacji przodek-potomek ich węzłów źródłowych. Aby możliwe było wykonanie tego rodzaju złączenia, musieliśmy opracować sposób przechowywania informacji o „pochodzeniu" wiersza względem dokumentu *XML*.

### Proponowane rozwiązania i przyjęte do implementacji

Analizowaliśmy dwa rozwiązania – pierwsze polegało na przechowywaniu informacji o węźle źródłowym dla wiersza (czyli referencji do instancji typu *Node*), drugie rozwiązanie polegało na rozróżnieniu węzłów poprzez unikalne identyfikatory.

Pierwsze rozwiązanie nasuwa się samo, jednak posiada pewne wady, które sprawiły, że zrezygnowaliśmy z jego implementacji. Przechowywanie referencji do węzłów źródłowych sprawia, że musimy przy wykonywaniu złączenia sprawdzać relację przodek-potomek poprzez analizę grafu dokumentu *XML*. Wprawdzie możliwe byłoby zastosowanie tutaj pewnych optymalizacji (przykładowo jednokrotne zagregowanie rodziców i umieszczenie takiej informacji w każdym węźle), jednak nie byliśmy przekonani do tego rozwiązania.

Rozwiązanie drugie opiera się na unikalnych identyfikatorach. Każdy z węzłów dostaje *GUID*, a przy tworzeniu wiersza na podstawie węzła przechowujemy w wierszu zarówno *GUID* źródła, jak i identyfikatory wszystkich przodków węzła źródłowego. Dzięki temu sprawdzenie zachodzenia relacji przodek-potomek sprowadza się do wyznaczenia przecięcia zbioru identyfikatorów, ustalenia, czy

potencjalnie zachodzi relacja przodek-potomek, a następnie sprawdzenie, czy odpowiednie identyfikatory mają te same wartości. To rozwiązanie przyjęliśmy do implementacji.

## Nadawanie aliasów kolumnom

### Opis problemu

Przy wykonywaniu zapytania z uwzględnieniem agregatów, każda z kolumn wyniku musi posiadać jakąś nazwę. W przypadku „zwykłych" kolumn jest to sprawa prosta – nazwa jest taka sama, jak alias tabeli źródłowej oraz nazwa kolumny. W przypadku wykonywanego agregata nie jest to już takie jednoznaczne (szczególnie gdy stosowane są również funkcje). Pojawia się też kwestia występowania w relacji wielu kolumn o tej samej nazwie.

### Proponowane rozwiązania i przyjęte do implementacji

W tym przypadku również rozważaliśmy dwa rozwiązania – pierwsze polegało na zezwoleniu na wiele kolumn o tej samej nazwie, drugie wymuszało unikalność nazw dla każdej kolumny. Do implementacji przyjęliśmy pierwsze rozwiązanie.

Rozwiązanie z zezwoleniem na wiele kolumn o tej samej nazwie jest stosowane w wielu bazach danych – przykładowo *MSSQL 2008*. Wprawdzie wprowadza to niejednoznaczność przy wykonywaniu zapytania z tabelą z duplikatami kolumn, jednak rozwiązanie tego problemu jest stosunkowo proste – użytkownik może jawnie podać alias dla kolumny, co kończy problem niejednoznaczności. Jednocześnie w tym podejściu kolumna mająca wartość będącą agregatem innej kolumny lub funkcji może mieć dowolną nazwę (w naszym przypadku jest to nazwa pusta), co dopuszcza standard *SQL/92* [1].

Rozwiązanie drugie jest o tyle problematyczne, że o niejednoznaczność w nazwach kolumn jest bardzo łatwo w typowym zapytaniu *SQL*. Rozwiązanie polegałoby na wymuszeniu na użytkowniku stosowania aliasów dla każdej problematycznej kolumny, z czego ostatecznie zrezygnowaliśmy.

# Propozycje rozwoju

## Implementacja języka SQL

### Podzapytania

Bardzo przydatnym mechanizmem języka *SQL* są podzapytania – zarówno skorelowane, jak i nieskorelowane. Pozwalają one na uproszczenie budowy niektórych zapytań, co w codziennej pracy ma niebagatelne znaczenie.

### CTE – Common Table Expressions

Kolejnym przydatnym mechanizmem, o który warto byłoby wzbogacić silnik *SQLxD*, są *Common Table Expressions* – pozwalają one na zwiększenie wydajności wykonywanych zapytań, a także upraszczają ich budowę przez uniknięcie wielu powtórzeń.

### HAVING

*Having* w działaniu przypomina klauzulę *Where*, jednak w wykonaniu różni się istotnym faktem - *Where* jest wykonywany przed projekcją danych (czyli nie ma dostępu do wyników agregatów), *Having* zaś jest wykonywany po, więc możliwe jest dodawanie warunków uwzględniających już przeliczone agregaty.

### DISTINCT

*Distinct* pozwala na pozbycie się zdublowanych rekordów z utworzonej relacji, co jest często niezbędne do poprawnego (w sensie pragmatycznym) wykonania zapytania (przykładowo znalezienie liczby różnych wartości w kolumnie).

### CASE

*Case* pozwala na wykonywanie wyrażeń warunkowych w projekcji, dzięki czemu możliwe jest budowanie jeszcze bardziej skomplikowanych zapytań.

### OVER

*Over* pozwala na wykonywanie pewnych obliczeń na podstawie zawężonego zbioru danych. Przykładowo agregaty można liczyć tylko na kolumnach nie będących w zbiorach *Group By*, co w wielu sytuacjach wymusza pewne niekomfortowe złączenia, aby wyliczyć interesujące nas wartości. *Over* pozwala na uproszczenie wielu zapytań i zwiększenie siły języka przez tworzenie zapytań bardziej intuicyjnie.

### CROS APPLY, OUTER APPLY

Operatory *Cross Apply* oraz *Outer Apply* pozwalają na wywoływanie dla każdego wiersza funkcji zwracających tabelę, a następnie wykonują złączenie (odpowiednio *INNER* lub *LEFT OUTER*), co pozwala na wykonywanie zapytań, które przy użyciu zwykłych złączeń byłyby niewykonalne. Dzięki tym elementom siła języka wzrosłaby jeszcze bardziej.

### Sumy, przecięcia, różnice zbiorów

Często należy zsumować zbiory, obliczyć ich przecięcie lub różnicę, dodanie możliwości wykonywania takich operacji znacząco zwiększyłoby siłę ekspresji języka.

## Generowanie dokumentu XML z wyników zapytania

Bardzo ciekawym kierunkiem rozwoju silnika jest tworzenie dokumentu *XML* z uzyskanego wyniku zapytania. Pozwoliłoby to na utworzenie narzędzia, które pozwalałoby na transformację dokumentów *XML* w stosunkowo łatwy sposób.

## Wydajność

W trakcie prac nad silnikiem nie skupialiśmy się na wydajności, nie przeprowadzaliśmy również testów szybkości działania aplikacji. W wielu miejscach stosowaliśmy konstrukcje obiektowe, które mogą stanowić zbyt duży narzut czasowy w generowaniu wyniku zapytania. Niezwykle interesującą kwestią wydaje się przetestowanie działania silnika pod kątem właśnie szybkości działania i zajętości pamięci operacyjnej.

# Dodatek A. Oryginalna gramatyka SQL 92

Poniżej znajduje się wybrany przez nas podzbiór gramatyki SQL/92 [8]

<scalar subquery> ::= <subquery>
<subquery> ::= <left paren> <query expression> <right paren>
<query expression> ::= <non-join query expression> | <joined table>
<non-join query expression> ::=
<non-join query term>
| <query expression> UNION [ ALL ] [ <corresponding spec> ] <query term>
| <query expression> EXCEPT [ ALL ] [ <corresponding spec> ] <query term>
<non-join query term> ::=
<non-join query primary>
| <query term> INTERSECT [ ALL ] [ <corresponding spec> ] <query primary>
<non-join query primary> ::= <simple table> | <left paren> <non-join query expression> <right paren>
<simple table> ::=
<query specification>
| <table value constructor>
| <explicit table>
<query specification> ::=
SELECT [ <set quantifier> ] <select list> <table expression>
<select list> ::=
<asterisk>
| <select sublist> [ { <comma> <select sublist> }... ]
<select sublist> ::= <derived column> | <qualifier> <period> <asterisk>
<derived column> ::= <value expression> [ <as clause> ]
<as clause> ::= [ AS ] <column name>
<table expression> ::=
<from clause>
[ <where clause> ]
[ <group by clause> ]
[ <having clause> ]
<from clause> ::= FROM <table reference> [ { <comma> <table reference> }... ] Note that <correlation specification> does not appear in the ISO/IEC grammar. The notation is written out longhand several times, instead.
<table reference> ::=
<table name> [ <correlation specification> ]
| <derived table> <correlation specification>
| <joined table>
<correlation specification> ::=
[ AS ] <correlation name> [ <left paren> <derived column list> <right paren> ]
<derived column list> ::= <column name list>
<derived table> ::= <table subquery>
<table subquery> ::= <subquery>
<joined table> ::=
<cross join>
| <qualified join>
| <left paren> <joined table> <right paren>

```
<cross join>   ::=
      <table reference>  CROSS  JOIN <table reference>
<qualified join>   ::=
      <table reference> [  NATURAL  ] [ <join type> ]  JOIN <table reference> [ <join specification> ]
<join type>   ::=
       INNER
    | <outer join type> [  OUTER  ]
    |  UNION
<outer join type>   ::=   LEFT  |  RIGHT  |  FULL
<join specification>   ::=  <join condition> | <named columns join>
<join condition>   ::=   ON <search condition>
<named columns join>   ::=   USING <left paren> <join column list> <right paren>
<join column list>   ::=  <column name list>
<where clause>   ::=   WHERE <search condition>
<group by clause>   ::=   GROUP  BY <grouping column reference list>
<grouping column reference list>   ::=
      <grouping column reference> [ { <comma> <grouping column reference> }... ]
<grouping column reference>   ::=  <column reference> [ <collate clause> ]
<collate clause>   ::=   COLLATE <collation name>
<collation name>   ::=  <qualified name>
<having clause>   ::=   HAVING <search condition>
<table value constructor>   ::=   VALUES <table value constructor list>
<table value constructor list>    ::=   <row value constructor> [ { <comma> <row value constructor>
}... ]
<explicit table>   ::=   TABLE <table name>
<query term>   ::=  <non-join query term> | <joined table>
<corresponding spec>   ::=   CORRESPONDING  [  BY <left paren> <corresponding column list> <right
paren> ]
<corresponding column list>   ::=  <column name list>
<query primary>   ::=  <non-join query primary> | <joined table>
<case expression>   ::=  <case abbreviation> | <case specification>
<case abbreviation>   ::=
       NULLIF <left paren> <value expression> <comma> <value expression> <right paren>
   |     COALESCE <left paren> <value expression> { <comma> <value expression> }... <right paren>
<case specification>   ::=  <simple case> | <searched case>
<simple case>   ::=
       CASE <case operand>
         <simple when clause> ...
         [ <else clause> ]
       END
<case operand>   ::=  <value expression>
<simple when clause>   ::=   WHEN <when operand>  THEN <result>
<when operand>   ::=  <value expression>
<result>   ::=  <result expression> |  NULL
<result expression>   ::=  <value expression>
<else clause>   ::=   ELSE <result>
<searched case>   ::=
       CASE
```

<searched when clause> ...
[ <else clause> ]
END
<searched when clause> ::= WHEN <search condition> THEN <result>
<cast specification> ::= CAST <left paren> <cast operand> AS <cast target> <right paren>
<cast operand> ::= <value expression> | NULL
<cast target> ::= <domain name> | <data type>
<numeric value function> ::= <position expression> | <extract expression> | <length expression>
<position expression> ::=
POSITION <left paren> <character value expression> IN <character value expression> <right paren>
<character value expression> ::= <concatenation> | <character factor>
<concatenation> ::= <character value expression> <concatenation operator> <character factor>
<character factor> ::= <character primary> [ <collate clause> ]
<character primary> ::= <value expression primary> | <string value function>
<string value function> ::= <character value function> | <bit value function>
<character value function> ::=
<character substring function>
| <fold>
| <form-of-use conversion>
| <character translation>
| <trim function>
<character substring function> ::=
SUBSTRING <left paren> <character value expression> FROM <start position> [ FOR <string length> ] <right paren>
<start position> ::= <numeric value expression>
<string length> ::= <numeric value expression>
<fold> ::= { UPPER | LOWER } <left paren> <character value expression> <right paren>
<form-of-use conversion> ::=
CONVERT <left paren> <character value expression> USING <form-of-use conversion name> <right paren>
<form-of-use conversion name> ::= <qualified name>
<character translation> ::=
TRANSLATE <left paren> <character value expression> USING <translation name> <right paren>
<translation name> ::= <qualified name>
<trim function> ::= TRIM <left paren> <trim operands> <right paren>
<trim operands> ::= [ [ <trim specification> ] [ <trim character> ] FROM ] <trim source>
<trim specification> ::= LEADING | TRAILING | BOTH
<trim character> ::= <character value expression>
<trim source> ::= <character value expression>
<bit value function> ::= <bit substring function>
<bit substring function> ::=
SUBSTRING <left paren> <bit value expression> FROM <start position> [ FOR <string length> ] <right paren>
<bit value expression> ::= <bit concatenation> | <bit factor>
<bit concatenation> ::= <bit value expression> <concatenation operator> <bit factor>

<bit factor> ::= <bit primary>
<bit primary> ::= <value expression primary> | <string value function>
<extract expression> ::= EXTRACT <left paren> <extract field> FROM <extract source> <right paren>
<extract field> ::= <datetime field> | <time zone field>
<datetime field> ::= <non-second datetime field> | SECOND
<time zone field> ::= TIMEZONE_HOUR | TIMEZONE_MINUTE
<extract source> ::= <datetime value expression> | <interval value expression>
<datetime value expression> ::=
<datetime term>
| <interval value expression> <plus sign> <datetime term>
| <datetime value expression> <plus sign> <interval term>
| <datetime value expression> <minus sign> <interval term>
<interval term> ::=
<interval factor>
| <interval term 2> <asterisk> <factor>
| <interval term 2> <solidus> <factor>
| <term> <asterisk> <interval factor>
<interval factor> ::= [ <sign> ] <interval primary>
<interval primary> ::= <value expression primary> [ <interval qualifier> ]
<interval term 2> ::= <interval term>
<interval value expression> ::=
<interval term>
| <interval value expression 1> <plus sign> <interval term 1>
| <interval value expression 1> <minus sign> <interval term 1>
| <left paren> <datetime value expression> <minus sign> <datetime term> <right paren> <interval qualifier>
<interval value expression 1> ::= <interval value expression>
<interval term 1> ::= <interval term>
<datetime term> ::= <datetime factor>
<datetime factor> ::= <datetime primary> [ <time zone> ]
<datetime primary> ::= <value expression primary> | <datetime value function>
<time zone> ::= AT <time zone specifier>
<time zone specifier> ::= LOCAL | TIME ZONE <interval value expression>
<length expression> ::= <char length expression> | <octet length expression> | <bit length expression>
<char length expression> ::= { CHAR_LENGTH | CHARACTER_LENGTH } <left paren> <string value expression> <right paren>
<string value expression> ::= <character value expression> | <bit value expression>
<octet length expression> ::= OCTET_LENGTH <left paren> <string value expression> <right paren>
<bit length expression> ::= BIT_LENGTH <left paren> <string value expression> <right paren>
<null specification> ::= NULL
<default specification> ::= DEFAULT
<row value constructor list> ::= <row value constructor element> [ { <comma> <row value constructor element> } ... ]
<row subquery> ::= <subquery>
<comp op> ::=

<equals operator>
| <not equals operator>
| <less than operator>
| <greater than operator>
| <less than or equals operator>
| <greater than or equals operator>

<between predicate> ::=
<row value constructor> [ NOT ] BETWEEN <row value constructor> AND <row value constructor>

<in predicate> ::= <row value constructor> [ NOT ] IN <in predicate value>

<in predicate value> ::= <table subquery> | <left paren> <in value list> <right paren>

<in value list> ::= <value expression> { <comma> <value expression> } ...

<like predicate> ::= <match value> [ NOT ] LIKE <pattern> [ ESCAPE <escape character> ]

<match value> ::= <character value expression>

<pattern> ::= <character value expression>

<escape character> ::= <character value expression>

<null predicate> ::= IS [ NOT ] NULL

<quantified comparison predicate> ::= <row value constructor> <comp op> <quantifier> <table subquery>

<quantifier> ::= <all> | <some>

<all> ::= ALL

<some> ::= SOME | ANY

<exists predicate> ::= EXISTS <table subquery>

<unique predicate> ::= UNIQUE <table subquery>

<match predicate> ::= <row value constructor> MATCH [ UNIQUE ] [ PARTIAL | FULL ] <table subquery>

<overlaps predicate> ::= <row value constructor 1> OVERLAPS <row value constructor 2>

<row value constructor 1> ::= <row value constructor>

<row value constructor 2> ::= <row value constructor>

<truth value> ::= TRUE | FALSE | UNKNOWN

# Dodatek B. Gramatyka robocza

<query specification> ::= SELECT [ <set quantifier> ] <select list> <table expression>
<set quantifier> ::= DISTINCT | ALL
<select list> ::= <asterisk>
| <select sublist> [ { <comma> <select sublist> }... ]
<select sublist> ::= <derived column>
<derived column> ::= <value expression> [ <as clause> ]
<value expression> ::= <string value expression>
<string value expression> ::= <character value expression>
<character value expression> ::= <character factor>
<character factor> ::= <character primary>
<character primary> ::= <value expression primary>
<value expression primary> ::=
<unsigned value specification>
| <column reference>
| <set function specification>
<set function specification> ::=
COUNT <left paren> <asterisk> <right paren>
| <general set function>
<general set function> ::=
<set function type> <left paren> <value expression> <right paren>
<set function type> ::= AVG | MAX | MIN | SUM | COUNT
<column reference> ::= <qualifier> <period> <column name> [ <period> <column name> ... ]
<column name> ::= <identifier>
<as clause> ::= [ AS ] <column name>
<unsigned value specification> ::= <unsigned literal>
<unsigned literal> ::= <unsigned numeric literal> | <general literal>
<unsigned numeric literal> ::=
<exact numeric literal>
<exact numeric literal> ::=
<unsigned integer>
<exact numeric literal> ::=
<unsigned integer>
<general literal> ::=
<character string literal>
<character string literal> ::=
QUOTE [ <character representation> ... ] QUOTE
<character representation> ::= <nonquote character>
<identifier> ::= <actual identifier>
<actual identifier> ::= <regular identifier>
<regular identifier> ::= <identifier body>
<identifier body> ::= <identifier start> [ { <underscore> | <identifier part> } ... ]
<identifier start> ::= <letter_or_hash>
<letter_or_hash> ::= <letter> | <hash>
<letter> ::= ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’ | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’
<hash> ::= ‘#’
<underscore> ::= ‘_’
<digit> ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
<identifier part> ::= <identifier start> | <digit>

```
<table expression>   ::=
      <from clause>
      [ <where clause> ]
      [ <group by clause> ]
      [ <order by clause]
<from clause>   ::=   FROM <table reference>
<table reference>   ::=
      <table name>  <correlation specification>
   | <joined table>
<table name>   ::=  <qualified name>
<qualified name>   ::=   <qualified identifier> [ PERIOD <qualified identifier> ...]
<qualified identifier>   ::=  <identifier> | QUESTION_MARK | ASTERISK
<correlation specification>   ::=         AS   <correlation name>
<correlation name>   ::=  <identifier>
<joined table>   ::=
      <cross join>
   | <qualified join>
<cross join>   ::=        <table reference>  CROSS  JOIN <table reference>
<qualified join>   ::=     <table reference>  <join type> JOIN <table reference> [ <join specification> ]
<join type>   ::=
                  NATURAL
          | INNER
   | <outer join type> [  OUTER  ]
<outer join type>   ::=   LEFT  |  RIGHT  |  FULL
<join specification>   ::=  <join condition>
<join condition>   ::=   ON <search condition>
<search condition>   ::=
      <boolean term>
   | <search condition>  OR <boolean term>
<boolean term>   ::=
      <boolean factor>
   | <boolean term>  AND <boolean factor>
<boolean factor>   ::=  [  NOT  ] <boolean test>
<boolean test>   ::=  <boolean primary>
<boolean primary>   ::=  <predicate> | <left paren> <search condition> <right paren>
<predicate>   ::=
      <comparison predicate>
   | <like predicate>
   | <null predicate>
<comparison predicate>   ::=  <row value constructor> <comp op> <row value constructor>
<row value constructor>   ::=
      <row value constructor element>
<like predicate>   ::=  <match value> [  NOT  ]  LIKE <pattern> [  ESCAPE <escape character> ]
<match value>   ::=  <character value expression>
<qualifier>   ::=  <table name> | <correlation name>
<pattern>   ::=  <character value expression>
<escape character>   ::=  <character value expression>
<null predicate>   ::=   IS  [  NOT  ]  NULL
<where clause>   ::=   WHERE <search condition>
<group by clause>   ::=   GROUP  BY <grouping column reference list>
<grouping column reference list>   ::=
```

<grouping column reference> [ { <comma> <grouping column reference> }... ]
<grouping column reference> ::= <column reference>
<order by clause> ::= ORDER BY <ordering column reference list> [ SKIP <unsigned integer> ] [ FETCH <unsigned integer> ]
<ordering column reference list> ::=
<ordering column reference> [ { <comma> <ordering column reference> }... ]
<ordering column reference> ::= <column reference> [ DESC ]
<comp op> ::=
<equals operator>
| <not equals operator>
| <less than operator>
| <greater than operator>
| <less than or equals operator>
| <greater than or equals operator>

## Dodatek C. Przykładowy dokument XML.

```
<dane>
    <postacie>
        <postac imie="Adam" id="1">
            <statystyki atak="25"/>
        </postac>
        <postac imie="Bogdan" id="2">
            <statystyki atak="30"/>
        </postac>
        <postac imie="Cezary" id="3">
            <statystyki atak="35"/>
        </postac>
    </postacie>
    <kraje>
        <kraj nazwa="Austria" idPostaci="1" id="1"/>
        <kraj nazwa="Belgia" idPostaci="2" id="2"/>
        <kraj nazwa="Czechy" idPostaci="3" id="3"/>
    </kraje>
    <miasta>
        <miasto nazwa="Albertów" idPostaci="1" id="1"/>
        <miasto nazwa="Babice" idPostaci="2" id="2"/>
    </miasta>
    <rzeczy>
        <rzecz nazwa="miecz" idPostaci="1" id="1"/>
        <rzecz nazwa="wiekiera" idPostaci="2" id="2"/>
        <rzecz nazwa="oszczep" idPostaci="3" id="3"/>
        <rzecz nazwa="maczuga" idPostaci="4" id="4"/>
    </rzeczy>
    <artefakty>
        <artefakt nazwa="berło króla" idPostaci="1" id="1"/>
        <artefakt nazwa="miecz gnolla" idPostaci="4" id="2"/>
    </artefakty>
    <cele>
        <cel nazwa="wieża"/>
        <cel nazwa="fort"/>
    </cele>
</dane>
```

# Bibliografia


[1] Digital Equipment Corporation, „International Standard ISO/IEC 9075:1992,” [Online]. Available: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[2] W3C, „XML Path Language (XPath) 2.0 (Second Edition),” 14 12 2010. [Online]. Available: http://www.w3.org/TR/xpath20/.

[3] W3C, „XQuery 1.0: An XML Query Language (Second Edition),” 14 12 2010. [Online]. Available: http://www.w3.org/TR/xquery/.

[4] E. F. Codd, A Relational Model of Data for Large Shared Data Banks, 1970.

[5] Microsoft Corporation, „LINQ to XML,” [Online]. Available: http://msdn.microsoft.com/pl-pl/library/bb387098(v=vs.110).aspx.

[6] Microsoft Corporation, „Implementing XML in SQL Server,” [Online]. Available: http://technet.microsoft.com/en-us/library/ms189887(v=sql.105).aspx.

[7] Wyrostek, Jakub, „Przetwarzanie dokumentów XML w oparciu o model quasi-relacyjny wraz z projektem języka zapytań,” 2010. [Online]. Available: http://sqlxd.org/pliki/documents/Praca-dyplomowa-SQLxD.pdf.

[8] „BNF Grammar for ISO/IEC 9075:1992 - Database Language SQL (SQL-92),” [Online]. Available: http://savage.net.au/SQL/sql-92.bnf.html.

[9] M. Dąbrowski i S. Kulpa, Opracowanie wygodnego narzędzia okienkowego do pracy z dokumentami XML przy pomocy SQLxD, 2013.

[10] Microsoft Corporation, „LIKE (Transact-SQL),” [Online]. Available: http://msdn.microsoft.com/en-US/library/ms179859(v=sql.105).aspx.

[11] „Python,” [Online]. Available: http://www.python.org/.

[12] D. Beazley, „PLY (Python Lex-Yacc),” [Online]. Available: http://www.dabeaz.com/ply/.

[13] „IronPython,” [Online]. Available: http://ironpython.net/.

[14] T. Parr, „ANother Tool for Language Recognition,” [Online]. Available: http://www.antlr.org/.

[15] J. G. Wayne Kelly, „Gardens Point Parser Generator,” [Online]. Available: http://gppg.codeplex.com/.

[16] „Irony - .NET Language Implementation Kit,” [Online]. Available: http://irony.codeplex.com/.

[17] „Moq,” [Online]. Available: http://code.google.com/p/moq/.

[18] „Typemock Isolator,” Typemock, [Online]. Available: http://www.typemock.com/.

[19] „xUnit.net,” [Online]. Available: http://xunit.codeplex.com/.

[20] J. T. S. B. Charlie Poole, „NUnit,” [Online]. Available: http://www.nunit.org/.

[21] „Rhino Mocks,” [Online]. Available: http://hibernatingrhinos.com/oss/rhino-mocks.