Dostęp do DBMS w Prologu za pomocą API ODBC

Paweł Maślanka

PWSZ w Tarnowie

8 stycznia 2011

Plan prezentacji

Plan prezentacji

Plan prezentacji

Plan prezentacji

Plan prezentacji

Plan prezentacji

Plan prezentacji

Plan prezentacji

Plan prezentacji

Plan prezentacji

Plan prezentacji

ODBC

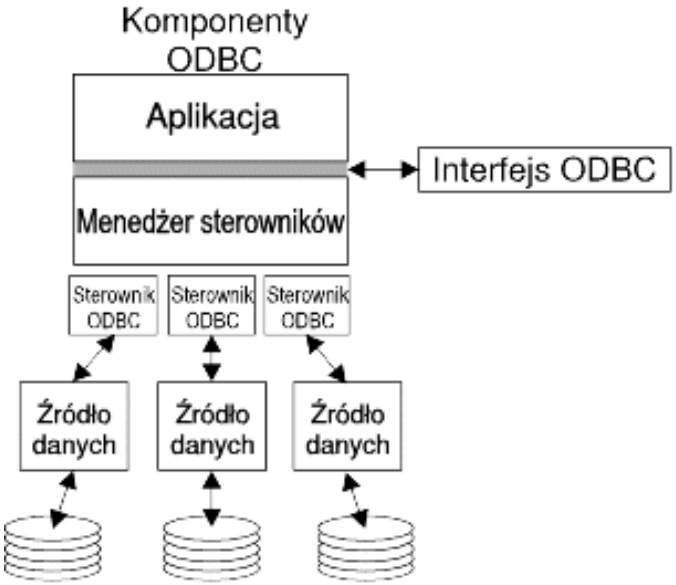
Co to jest ODBC ?

ODBC (ang. Open DataBase Connectivity - otwarte łącze baz danych) - interfejs pozwalający programom łączyć się z systemami zarządzającymi bazami danych (DBMS).
Celem projektantów ODBC było uniezależnienie go od języków programowania, systemów operacyjnych i baz danych. Tak więc ODBC umożliwia każdej aplikacji dostęp do danych, niezależnie od tego, jaki system zarządzania bazą danych (DBMS) przetwarza te dane.

ODBC

Jak to działa ?

ODBC dokonuje tego dzięki pośredniej warstwie, zwanej sterownikiem bazy danych, umieszczonej między aplikacją, a DBMS. Celem tej warstwy jest translacja zapytań wysłanych przez aplikację na komendy specyficzne dla danego DBMS. Dzięki temu aplikacja musi znać tylko składnię API ODBC, a sterownik przekaże zapytanie do DBMS w jego natywnym formacie, zwracając dane w formacie zrozumiałym dla aplikacji.



Aplikacja. Realizuje przetwarzanie i wywołuje funkcje ODBC w celu uruchamiania instrukcji SQL.
Menedżer sterowników. Menedżer sterowników ODBC to interfejs pomiędzy aplikacją ODBC i sterownikiem ODBC.
Przetwarza wywołania funkcji ODBC i przesyła żądania do sterownika.
Sterownik. Przetwarza wywołania funkcji ODBC, wysyła żądania SQL do określonych źródeł danych i zwraca wyniki do aplikacji.
Źródło danych. DBMS.

Co będzie nam potrzebne ?

Aby umożliwić programowi komunikację z bazą danych poprzez interfejs ODBC musimy posiadać zainstalowane niżej wymienione pakiety:

Środowisko SWI-Prolog
: `http://www.swi-prolog.org`

DBMS
: PostgreSQL - `http://www.postgresql.org`

Sterownik ODBC
: psqlODBC - `http://psqlodbc.projects.postgresql.org`

Menedżer sterownika
: unixODBC - `http://www.unixodbc.org`

Konfiguracja sterowników ODBC

Sterowniki ODBC zdefiniowane są w pliku **odbcinst.ini**, przeważnie w katalogu **/etc**
(w celu zweryfikowania lokalizacji wymaganych plików należy do terminala wprowadzić komendę: **odbcinst -j**).
Poniższy przykład przedstawia specyfikację sterownika:

Nazwa sterownika

[PostgreSQL Unicode]

Description = opis sterownika

Description = PostgreSQL ODBC driver (Unicode version)

Driver = ścieżka do sterownika

Driver = /usr/lib/psqlodbcw.so

Do czego potrzebna jest nam nazwa źródła danych (DSN) ?

Aby używać źródła danych, należy utworzyć nazwę źródła danych (DSN).
DSN zawiera informacje potrzebne sterownikowi w celu uzyskania dostępu do DBMS. Można podać jedną z następujących nazw DSN:

- **DSN użytkownika:** Te źródła danych są źródłami lokalnymi dla komputera i są dostępne tylko dla użytkownika, który je utworzył. Informacje te przechowywane są w rejestrze(Windows).
- **DSN systemu:** Te źródła danych są źródłami lokalnymi dla komputera, nie są przeznaczone dla użytkownika. System lub użytkownik z odpowiednimi uprawnieniami może używać źródła danych skonfigurowanego w DSN systemu. Informacje te przechowywane są w rejestrze.
- **DSN pliku:** Są to źródła danych oparte na plikach, które mogą być współużytkowane przez wszystkich użytkowników z zainstalowanymi takimi samymi sterownikami, umożliwiającymi im dostęp do bazy danych. Te źródła danych nie muszą być dedykowane dla użytkownika ani lokalne dla komputera.

Konfiguracja DSN

W celu konfiguracji nazwy źródła danych wykorzystamy DSN pliku, edytując plik **odbc.ini** w katalogu systemowym **/etc**.
Poniższy przykład przedstawia specyfikację DSN:

Nazwa źródła danych ODBC
[PostgreSQLProlog]

Description = *opis źródła danych*
Description = Baza PostgreSQL dla programów w Prologu

Driver = *nazwa sterownika*
Driver = PostgreSQL Unicode

Database = *nazwa bazy danych*
Database = prolog

Servername = *adres serwera*
Servername = localhost

Username = *nazwa użytkownika*
UserName = postgres

Password = *hasło*
Password = postgress

Port = *numer portu*
Port = 5432

Sprawdzenie zdefiniowanego połączenia można wykonać za pomocą poniższego polecenia:

isql PostgreSQLProlog

Argument **PostgreSQLProlog** to nazwa źródła danych z pliku **odbc.ini**

Interfejs ODBC dla Prologa dostarcza predykatów do zarządzania połączeniem, wykonywania zapytań SQL oraz odwzorowania typów danych.

Łączenie z bazą danych

odbc_connect(+DSN, -Connection, +Options)

Predykat ten tworzy nowe połączenie ODBC do źródła danych **DSN** (Data Source Name), a następnie zwraca uchwyt do tego połączenia poprzez ukonkretnienie nim zmiennej **Connection**. Opcje dla połączenia definiuje się za pomocą listy i przekazuje się ją w argumencie **Options**. Możliwe jest użycie następujących opcji:

- **user(User)** - nazwa użytkownika
- **password(Password)** - hasło
- **alias(AliasName)** - AliasName służy jako alias do zmiennej **Connection**
- **open(OpenMode)** - przyjmuje dwie wartości: once, multiple. Opcja ta definiuje zachowanie przy próbie ponownej próby połączenia do bazy danych przy użyciu tej samej nazwy źródła danych. Gdy wartością jest **once** wówczas zwrócony pozostaje uchwyt do istniejącego połączenia, w przeciwnym razie otwierane jest kolejne połączenie do tego samego źródła danych.

Przykład

```
open_connection :-
  odbc_connect('PostgreSQLProlog', Connection, [user(postgres),
                        password(postgres), alias(psql), open(once)]).
```

Łączenie z bazą danych

odbc_driver_connect(+DriverString, -Connection, +Options)

Predykat ten pozwala na połączenie z bazą przy użyciu wywołania funkcji **SQLDriverConnect()**. **DriverString** przekazany jest bez sprawdzenia. Opcje **Options** nie powinny zawierać atrybutów **user** i **password**.

Rozłączenie z bazą danych

odbc_disconnect(+Connection)

Predykat ten zamyka aktywne połączenie, przekazane w uchwycie **Connection**.
Niszczy również aliasy dla tego uchwytu lub, jeżeli nie ma aliasów, uniemożliwia dalsze używanie uchwytu **Connection**.

Przykład

```
close_connection :-
  odbc_disconnect(psql).
```

Ustawianie opcji połączenia

odbc_set_connection(+Connection, +Option)

Predykat ten pozwala na ustawienie właściwości dla istniejącego połączenia **Connection** za pomocą parametru **Option**.
Opcje możliwe do zdefiniowania to:

- **access_mode(Mode)** - możliwe wartości **Mode**: read, update. Jeżeli wartością jest **read**, to informujemy sterownik, że chcemy uzyskać dostęp do bazy danych w trybie do odczytu. Jeżeli ustawimy tryb dostępu na **update**(domyślnie), to informujemy sterownik, że możemy dokonywać aktualizacji stanu bazy danych.
- **auto_commit(bool)** - jeżeli wartością jest **true**(domyślnie), wówczas każda kwerenda aktualizująca stan bazy danych jest zatwierdzana natychmiast. Jeżeli wartością jest **false**, wówczas każda kwerenda aktualizująca stan bazy danych rozpoczyna transakcję, która może zostać zatwierdzona lub wycofana.
- **silent(Bool)** - jeżeli wartością jest **true**, wówczas wyrażenie zwraca **SQL_SUCCESS_WITH_INFO** w razie powodzenia bez wyświetlania informacji.
- **null(NullSpecifier)** - definiuje jak reprezentowana ma być wartość **NULL**. Domyślnie **NULL** jest reprezentowany przez atom *null*.

Przykład

```
query_for_null :-
  odbc_query(psql, 'SELECT nazwa, opis FROM kompozycje WHERE nazwa =
        \'Bukiet 10\'',
           row(Nazwa, Opis)),
  write('Nazwa ' - Nazwa), nl,
        write('Opis ' - Opis).
```

```
set_connection :-
  odbc_set_connection(psql, null(_)).
```

Ustalanie aktywnych połączeń

odbc_current_connection(?Connection, ?DSN)

Za pomocą powyższego predykatu możemy zidentyfikować aktywne połączenia. Poprzez zmienną **Connection** ustalamy identyfikator(alias) połączenia, a dzięki zmiennej **DSN** otrzymujemy nazwę źródła danych skojarzoną z połączeniem.

Przykład

```
connections :-
  odbc_current_connection(Connection, DSN),
  write(Connection - DSN).
```

Pobranie danych na temat połączenia

odbc_get_connection(+Connection, ?Property)

Predykat ten definiuje zapytanie o właściwości połączenia **Connection**.
Property to term o formacie **atrybut(wartość)**. Predykat ten pozwala na odczyt następujących właściwości połączenia:

- **database_name(Atom)** - nazwa bazy danych związana z połączeniem.
- **dbms_name(Name)** - nazwa silnika bazy danych.
- **dbms_version(Atom)** - wersja silnika bazy danych.
- **driver_name(Name)** - nazwa sterownika dostarczającego interfejs pomiędzy ODBC, a bazą danych.
- **driver_odbc_version(Atom)** - wersja ODBC wspierana przez sterownik.
- **driver_version(Atom)** - wersja sterownika.
- **active_statements(Integer)** - maksymalna liczba wykonywanych zapytań, które mogą być aktywne w tym samym czasie dla danego połączenia. Zwraca 0, jeżeli nie określono limitu.

Przykład

```
property_connection :-
  odbc_get_connection(psql,database_name(DBName)),
  odbc_get_connection(psql,dbms_name(DBMSName)),
  odbc_get_connection(psql,dbms_version(DBMSVer)),
  odbc_get_connection(psql,driver_name(DriverName)),
  odbc_get_connection(psql,driver_version(DriverVer)),
  odbc_get_connection(psql,driver_odbc_version(ODBCVer)),
  odbc_get_connection(psql,active_statements(ActiveStmts)),
  write('Nazwa bazy danych '          - DBName),      nl,
  write('Silnik bazy danych '         - DBMSName),    nl,
  write('Wersja silnika '             - DBMSVer),     nl,
  write('Nazwa sterownika '           - DriverName),  nl,
  write('Wersja sterownika '          - DriverVer),   nl,
  write('Wersja ODBC '                - ODBCVer),     nl,
  write('Liczba aktywnych kwerend '- ActiveStmts), nl.
```

Opis zdefiniowanych DSN

odbc_data_source(?DSN, ?Description)

Predykat ten zwraca opis do wszystkich zdefiniowanych źródeł danych w systemie.
DSN to nazwa źródła danych tak samo jak w predykacie **odbc_connect/3**.
Description zawiera opis dla danego źródła danych.

Przykład

```
data_sources :-
  odbc_data_source(DSN, Description),
  write(DSN + Description).
```

API ODBC pozwala nam na obsługę zapytań w formie tradycyjnej, jako tekst zapytania SQL oraz w formie zapytania sparametryzowanego. Pierwszą formę stosuje się dla zapytań rzadko wykonywanych (np. tworzenie tabeli). Zapytanie sparametryzowane pozwala interfejsowi ODBC i sterownikowi bazy danych na wstępną kompilację zapytania i przechowywanie jego zoptymalizowanego kodu, dzięki czemu uzyskujemy lepszą wydajność dla powtarzanych zapytań. Ponadto pozwalają one na przekazanie do zapytania parametrów.

odbc_query(+Connection, +SQL, -RowOrAffected, +Options)

Powyższy predykat wykonuje zapytanie **SQL** do bazy danych reprezentowanej przez uchwyt **Connection**. **SQL** to każde prawidłowe zapytanie. Zapytanie **SQL** może być zdefiniowane jako atom, łańcuch znaków czy też jako term w formacie **Format** - **Argumenty**, który jest konwertowany przy użyciu predykatu **format/2**.
Jeżeli zapytanie jest wyrażeniem **SELECT**, wówczas zbiór wyników zwracany jest w **RowOrAffected**. Domyślnie rekordy zwracane są jeden za drugim w procesie nawracania jako termy funktora **row/arg**{**Arity**}, gdzie **Arity** oznacza numer kolumny w zbiorze wyników. Dla pozostałych zapytań **RowOrAffected** zwraca **affected(Rows)**, gdzie **Rows** reprezentuje liczbę zwróconych wierszy przez zapytanie.
Parametr **Options** definiuje następujące opcje:

- **null(NullSpecifier)** - definiujemy format reprezentacji wartości **NULL**, podobnie jak w predykacie **odbc_set_connection/2**
- **types(ListOfTypes)** - określa do jakiego typu powinny zostać skonwertowane zwrócone wartości poszczególnych kolumn
- **source(Bool)** - jeżeli wartością jest **true**(domyślnie **false**) wówczas zwrócone wyniki zawierają także nazwę tabeli oraz kolumny. Każdy wiersz zwracany jest w postaci: *column(NazwaTabeli, NazwaKolumn, Wartość)*
- **findall(Template, row(Column, ...))** - zwraca wyniki w formie listy

Przykłady

Zapytanie zwracające pojedyncze wiersze

```
odbc_query(psql, 'SELECT nazwa FROM Kompozycje', row(Nazwa)).
```

Zapytanie modyfikujące rekord w bazie

```
odbc_query(psql, 'UPDATE Kompozycje SET cena = 8
        WHERE stan = 2', affected(Rows)).
```

Zapytanie konwertujące typ wartości kolumny

```
odbc_query(psql, 'SELECT nazwa, stan FROM Kompozycje',
            row(Nazwa, Stan), [types([default, float])]).
```

odbc_query(+Connection, +SQL, -RowOrAffected)

Taki sam jak predykat **odbc_query/4** bez użycia opcji.

odbc_query(+Connection, +SQL)

Taki sam jak predykat **odbc_query/3**, ale używany do zapytań takich form SQL, które nie powinny zwracać wyniku jako wierszy (wszystkie wyrażenia za wyjątkiem zapytania SELECT). Predykat wypisuje diagnostyczny komunikat, jeżeli zapytanie zwróci wynik.

Przykład

Zapytanie zwracające wyniki w postaci listy przy użyciu predykatów **findall/3** i **odbc_query/3**

```
findall(Nazwa, odbc_query(psql, 'SELECT nazwa FROM Kompozycje',
        row(Nazwa)), ListaNazw).
```

Zapytania sparametryzowane posiadają znak **?** w miejscu, gdzie pojawić się ma parametr. Zalecane jest stosowanie zapytań sparametryzowanych, gdyż są one optymalizowane przez silnik ODBC i przy wielokrotnym użyciu znacząco podwyższają wydajność zapytań w przeciwieństwie do zapytań nieparametryzowanych.

Przygotowanie zapytania

odbc_prepare(+Connection, +SQL, +Parameters, -Statement, +Options)

- **Connection** - uchwyt do połączenia
- **SQL** - zapytanie SQL, które zawiera **?** w miejscu, gdzie ma pojawić się parametr
- **Parameters** - lista z typami parametrów
- **Statement** - zmienna, która będzie ukonkretniona tym zapytaniem
- **Options** - opcje do wykonywanego zapytania

Parameters:

- **default** - ODBC sam pobierze typ parametru z bazy danych
- **TypSQL** - typ używany przez SQL np. *char*, *varchar*, *integer*

Options:

- **fetch(FetchType)** - ustawienie FetchType, na *auto* (domyślne) powoduje że wyniki będą pobieranie przy nawracaniu, tak jak to ma miejsce w zapytaniach niesparametryzowanych, ustawienie FetchType, na *fetch* powoduje że wyniki zapytania będą mogły być pobrane za pomocą predykatu **odbc_fetch/3**.

odbc_prepare(+Connection, +SQL, +Parameters, -Statement)

Predykat taki sam, jak **odbc_prepare/5** bez możliwości definiowania opcji

Wykonanie zapytania

odbc_execute(+Statement, +ParameterValues, -RowOrAffected)

- **Statement** - zmienna, która została ukonkretniona przygotowanym zapytaniem
- **ParameterValues** - lista parametrów, które mają zastąpić znak **?** w zapytaniu
- **RowOrAffected** - zwracane wyniki zapytania

Predykat może rzucić wyjątek *type_error* jeśli typy podane w **ParameterValues** nie będą się zgadzały z tymi podanymi podczas przygotowania zapytania

odbc_execute(+Statement, +ParameterValues)

Podobny do predykatu **odbc_query/2**. Używany do prostych zapytań, gdzie nie interesują nas wyniki zwrócone przez zapytanie.

Zwolnienie zasobu

odbc_free_statement(+Statement)

Predykat ten zwalnia przygotowany wcześniej zasób **Statement**, utworzony przez predykat **odbc_prepare**.

Przykład

Przygotowanie zapytania

```
odbc_prepare(psql, 'SELECT nazwa FROM kompozycje
                    WHERE stan > ?', [integer], Statement).
```

Wykonanie zapytania

```
odbc_execute($Statement, [5], Rows).
```

Zwolnienie zasobu

```
odbc_free_statement($Statement).
```

Innym sposobem otrzymania wyników z zapytania jest użycie predykatu **odbc_fetch/3**.

odbc_fetch(+Statement, -Row, +Option)

Predykat ten pobiera pojedynczy rekord ze zbioru wyników zapytania identyfikowanego przez **Statement**. Zapytanie **Statement** musi być utworzone przez predykat **odbc_prepare/5** przy użyciu opcji **fetch(fetch)** i wykonane za pomocą predykatu **odbc_execute/2**. **Row** jest unifikowany pobranym rekordem lub atomem **end_of_file** po pobraniu wszystkich danych.
Opcja **Option** jest jedną z poniższych:

- **next** - pobranie następnego wiersza
- **prior** - pobranie poprzedniego wiersza
- **first** - pobranie pierwszego wiersza
- **last** - pobranie ostatniego wiersza
- **absolute(Offset)** - pobranie wiersza o numerze *Offset*
- **relative(Offset)** - pobranie określonego wiersza względem bieżącej pozycji *Offset* *relative(1)* oznacza to samo co *next()*
- **bookmark(Offset)** - zarezerwowane dla przyszłych implementacji

Przykład

```
fetch(Options) :-
  odbc_set_connection(psql, cursor_type(static)),
  odbc_prepare(psql, 'SELECT nazwa FROM kompozycje',
                     [], Statement, [fetch(fetch)]),
  odbc_execute(Statement, []),
  fetch(Statement, Options).

fetch(Statement, Options) :-
  odbc_fetch(Statement, row(Row), Options),
  (Row == end_of_file -> true; writeln(Row),
    fetch(Statement, Options)).
```

ODBC może działać w dwóch trybach transakcji.
W domyślnym trybie wszystkie modyfikacje są zatwierdzane natychmiast po wykonaniu zapytania.
Używając predykatu **odbc_set_connection/2** można zmienić tryb, w którym ODBC przełącza się w tryb transakcji i każde zapytanie, które jest wykonywane uruchamia nową transakcję.
Transakcje te mogą zostać zakończone przez użycie predykatu **odbc_end_transaction/2**.

Kończenie transakcji

odbc_end_transaction(+Connection, +Action)

Powyższy predykat kończy bieżąco otwartą transakcję, jeżeli istnieje.

- **Connection** - uchwyt do połączenia
- **Action** - podejmowana akcja dla transakcji
  commit - zatwierdzenie transakcji
  rollback - porzucenie transakcji

Przykład

Wyłączenie automatycznego zatwierdzania transakcji

```
odbc_set_connection(psql, auto_commit(false)).
```

Zatwierdzenie bieżącej transakcji

```
odbc_end_transaction(psql, 'commit').
```

Słownik Bazy Danych (Data Dictionary)

Słownik Bazy Danych (ang. Data Dictionary) zawiera zestaw opisów definiujących wszystkie wchodzące w skład bazy danych tabele (ang. Tables), a także definicje indeksów (ang. Indexes), sekwencji (ang. Sequences) i wyzwalaczy (ang. Triggers).
W Słowniku Bazy Danych definiowane są także warunki walidacyjne, reguły integralności bazy danych i ograniczenia w dostępie do danych.
Jednym z narzędzi zabezpieczenia integralności bazy danych są wyzwalacze, czyli specyficzne procedury definiowane przez projektanta bazy i uruchamiane automatycznie w momencie wykonywania określonych akcji (np. dodanie nowego rekordu, modyfikacja zawartości pola itp.).

Dostęp do słownika bazy danych

odbc_current_table(+Connection, -Table)

Powyższy predykat zwraca podczas nawracania nazwy wszystkich tabel przechowywanych w bazie danych identyfikowanej przez **Connection**.

- **Connection** - uchwyt do połączenia
- **Table** - zmienna, która zostanie ukonkretniona wszystkimi nazwami tabel

Przykład

Pobranie nazw tabel składowanych w bazie

```
odbc_current_table(psql, Tables).
```

odbc_current_table(+Connection, ?Table, ?Property)

Powyższy predykat pobiera informacje o tabeli.

- **Connection** - uchwyt do połączenia
- **Table** - nazwa tabeli
- **Property** - właściwość tabeli

Property:

- **qualifier(Qualifier)** - nazwa bazy danych, w której znajduje się tabela
- **owner(Owner)** - właściciel
- **comment(Comment)** - komentarz
- **arity(Arity)** - ilość kolumn w tabeli

Przykład

Pobranie informacji o tabeli

```
odbc_current_table(psql, kompozycje, Wlasciwosci).
```

odbc_table_column(+Connection, ?Table, ?Column)

Powyższy predykat pozwala uzyskać nam nazwy kolumn tabeli.

- **Connection** - uchwyt do połączenia
- **Table** - nazwa tabeli, dla której pobierane będą nazwy kolumn
- **Column** - zmienna, która zostanie ukonkretniona wszystkimi nazwami kolumn

Przykład

Uzyskanie nazw kolumn tabeli

```
odbc_table_column(psql, kompozycje, Names).
```

odbc_table_column(+Connection, ?Table, ?Column, ?Facet)

Powyższy predykat pozwala nam uzyskać informacje o kolumnie tabeli.

- **Connection** - uchwyt do połączenia
- **Table** - nazwa tabeli, dla której pobierane będą nazwy kolumn
- **Column** - nazwa kolumny
- **Facet** - właściwość kolumny

Facet:

- **table_qualifier(Qualifier)** - nazwa bazy danych, w której znajduje się tabela kolumny
- **table_owner(Owner)** - właściciel tabeli kolumny
- **table_name(Table)** - nazwa tabeli kolumny
- **data_type(DataType)** - identyfikator typu
- **type_name(TypeName)** - nazwa typu
- **precision(Precision)** - precyzja
- **length(Length)** - długość (w bajtach)
- **scale(Scale)** - ilość miejsc po przecinku
- **nullable(Nullable)** - czy może być NULL
- **type(Type)** - typ reprezentowany przez Prolog

Przykład

Uzyskanie właściwości kolumny

```
odbc_table_column(psql, kompozycje, cena, Property).
```

odbc_type(+Connection, ?TypeSpec, ?Property)

Powyższy predykat pozwala nam na uzyskanie informacji o typach danych obsługiwanych przez źródło danych.

- **Connection** - uchwyt do połączenia
- **TypeSpec** - identyfikator typu lub stała *all_types*, która wylistuje wszystkie typy danych
- **Property** - właściwość typu

Property:

- name(Name) - nazwa typu
- data_type(DataType) - identyfikator typu
- precision(Precision) - precyzja
- literal_prefix(Prefix) - znak specjalny występujący przed znakiem
- literal_suffix(Suffix) - znak specjalny występujący za znakiem
- create_params(CreateParams) - parametry przy definiowaniu, np. przy określaniu precyzji
- nullable(Bool) - czy może przyjąć NULL
- case_sensitive(Bool) - czy rozróżnia wielkość liter
- searchable(Searchable) - czy może występować w kryterium wyszukiwania
- unsigned(Bool) - czy reprezentuje typ bez znaku
- money(Bool) - czy reprezentuje typ monetarny
- auto_increment(Bool) - czy automatycznie zwiększa swoją wartość
- minimum_scale(MinScale) - minimalny rozmiar typu
- maximum_scale(MaxScale) - maksymalny rozmiar typu

Przykład

Uzyskanie informacji o typie *int4*(identyfikator: **4**)

```
odbc_type(psql, 4, Property).
```

Pobranie listy wszystkich typów danych obsługiwanych przez bazę danych

```
odbc_type(psql, all_types, name(Type)).
```

odbc_statistics(?Key)

Powyższy predykat pozwala nam na uzyskani statystyk dotyczących zapytań.

- **Key** - zdefiniowany jako: **statements(Utworzone, Zwolnione)**
  Otrzymanie informacji o liczbie utworzonych i zwolnionych instrukcji przez wszystkie połączenia.
  Zapytania wykonywane za pomocą **odbc_query/[2-3]** zwiększają liczbę utworzonych zapytań oraz zwiększają liczbę zwolnionych zapytań, gdy zostaną wykonane bez względu na to czy zakończyły się one powodzeniem, nie powodzeniem czy nawet rzuciło wyjątkiem.
  Zapytania utworzone za pomocą **odbc_prepare/[4-5]** zwiększają tylko liczbę utworzonych zapytań, a **odbc_free_statement/1** liczbę zwolnionych zapytań

Przykład

Pobranie statystyk zapytań do bazy

```
odbc_statistics(statements( Utworzonych, Zwolnionych )).
```

Ustawienie poziomu debugger'a

odbc_debug(+Level)

Powyższy predykat pozwala nam ustawić poziom pracy debugger'a.

- **Level** - poziom debugger'a, domyślnie: 0
  Na najwyższym poziomie wypisywane są wszystkie informacje, przydatne głównie dla deweloperów

SQL posiada więcej typów niż Prolog, niektóre z nich są identyczne jak w Prologu. Jednak część typów SQL jest reprezentowana w Prologu przez jeden typ dzięki wbudowanej w interfejsie ODBC/Prolog funkcji do przekształcania typów.

Reprezentacja typów SQL w Prologu

- **atom**
  Typ używany domyślnie dla następujących typów SQL: *char*, *varchar*, *longvarchar*, *binary*, *varbinary*, *longvarbinary*, *decimal* i *numeric*.
  Może być używany do wszystkich typów.
- **string**
  Rozszerzony typ string w SWI-Prolog.
  Może być używany do wszystkich typów.
- **codes**
  Lista znaków. Może być używana do wszystkich typów.

Reprezentacja typów SQL w Prologu

- **integer**
  Typ używany domyślnie dla następujących typów SQL: *bit*, *tinyint*, *smallint* i *integer*.
  Należy zwrócić uwagę, że typ *integer* w SWI-Prolog jest wartością 32-bitową ze znakiem, podczas gdy typ SQL zezwala także na typ bez znaku. Typ ten może być używany do typów całkowitych, a także do typu *decimal*, *date* i *timestamp*, który reprezentowany jest według czasu POSIX (sekundy liczone są od 1 Stycznia 1970).
- **double**
  Typ używany domyślnie dla następujących typów SQL: *real*, *float* i *double*.
  Typ ten może być używany do typów całkowitych, *float* i *decimal*, a także do *date* i *timestamp*, który reprezentowany jest jako *timestamp* według POSIX (sekundy liczone są od 1 Stycznia 1970).

Reprezentacja typów SQL w Prologu

- **date**
  Term w Prologu w formacie *date(Rok,Miesiąc,Dzień)* używany jako domyślny typ dla typu SQL *date*
- **time**
  Term w Prologu w formacie *time(Godzina,Minuta,Sekunda)* używany jako domyślny typ dla typu SQL *time*
- **timestamp**
  Term w Prologu w formacie *timestamp(Rok, Miesiąc, Dzień, Godzina, Minuta, Sekunda, Milisekundy))* używany jako domyślny typ dla typu SQL *timestamp*

Źródła informacji:

- http://www.swi-prolog.org/pldoc/package/odbc.html
- http://www.unixodbc.org/
- http://home.agh.edu.pl/~ligeza/wiki/_media/presentations:odbc_latin2.pdf?id=presentations%3Aprolog&cache=cache
- https://ai.ia.agh.edu.pl/wiki/_media/pl:miw:referat.pdf
- http://www.amzi.com/manuals/amzi/ls/lsxodbc.htm
- http://www.easysoft.com/developer/interfaces/odbc/linux.html
- https://ai.ia.agh.edu.pl/wiki/pl:prolog:prolog_lab:prolog_lab_rdbms
- http://publib.boulder.ibm.com/infocenter/idshelp/v10/index.jsp?topic=/com.ibm.odbc.doc/odbc20.htm
- W. F. Clocksin, C. S. Mellish - "Prolog. Programowanie"

Dziękuję za ewentualną uwagę.