Zaawansowane Technologie Bazodanowe

Sprawozdanie z projektu

Przetwarzanie grafów

Łukasz Brewczyński Filip Dziedzic Rafał Studnicki

Informatyka
Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej
Akademia Górniczo-Hutnicza im. St. Staszica w Krakowie

---

1. Wprowadzenie

Celem projektu była implementacja silników przetwarzających graf będący odwzorowaniem systemu sterowania oświetleniem z użyciem trzech różnych technologii bazodanowych: Mnesia, Neo4j oraz MySQL na podstawie implementacji referencyjnej w języku Prolog.
Każdy z zaimplementowanych silników stanowi część wysokopoziomowego agenta sterowania oświetleniem, który jest elementem pośrednim pomiędzy wartościami odczytywanymi z sensorów a sygnałami sterującymi, które wysyłane są do poszczególnych urządzeń świetlnych. Szczegóły implementacyjne dotyczące ww. silników zawarte są w odpowiednich sekcjach sprawozdania.
Agent wysokiego poziomu zaimplementowany został w języku Erlang, a każdy ze wspominanych silników może zostać używany wymiennie w zależności od wybranej architektury systemu, poprzez wybór odpowiedniego modułu w pliku konfiguracyjnym `etc/lighting.cfg`.

2. Środowisko testowe

Wszystkie uruchomienia eksperymentów w ramach projektu zostały wykonane na platformie o następującej specyfikacji:

- System operacyjny: MacOSX 10.9
- Procesor: Intel Core i7 2,9 GHz

- Pamięć RAM: 8 GB

Należy jednak zauważyć, że ze względu na charakterystykę problemu i konieczność sekwencyjnego wykonywania eksperymentów, w trakcie ich wykonywania wykorzystywany mógł być tylko jeden rdzeń procesora.

3. Rozwiązanie referencyjne

3.1 Różnice pomiędzy uruchomieniami implementacji referencyjnej w różnych wersjach SWI-Prolog.

Poniżej zaprezentowano czas uruchomienia eksperymentów przy użyciu SWI-Prolog 5.8.2:

```
194,734 inferences, 2.720 CPU in 2.729 seconds (100% CPU, 71593 Lips)
```

Poniżej natomiast, czas uruchomienia eksperymentów przy użyciu SWI-Prolog 6.4.1:

```
194,737 inferences, 0.097 CPU in 0.098 seconds (99% CPU, 2000318 Lips)
```

Znacząca różnica w czasie wykonywania testów (różnica rzędu 98 ms zestawione z 2729 ms) wynikają z wprowadzonego w wersji 6 SWI-Prolog[1], dynamicznego indeksowania wyrażeń w bazie wiedzy (*Just-in-time indexing*[2]). Dlatego też, w celu dokonania porównania zbliżonych sposobów indeksowania danych zarówno w przypadku środowiska uruchomieniowego SWI-Prolog jak i implementacji silników przetwarzających graf, jako referencyjny czas wykonania testów przyjęto czas ich uruchomienia w SWI-Prolog w wersji 5.8.2.

3.2 Różnice pomiędzy implementacją referencyjną a implementacją silników przetwarzania grafów

Ze względu na konieczność dostosowania implementacji silników do współpracy z wysokopoziomowym agentem systemu sterowania oświetleniem, który zaimplementowany został w języku Erlang, w implementacjach silników można wyszczególnić następujące różnice w porównaniu z implementacją referencyjną w języku Prolog:

- dla implementacji opartych na bazach Neo4j oraz MySQL konieczne było użycie pewnej warstwy komunikacji pomiędzy kodem w Erlangu a bazą

[1] http://swi-prolog.org/versions.html
[2] http://www.swi-prolog.org/pldoc/man?section=jitindex

danych.
W przypadku bazy danych Neo4j był to interfejs REST, do komunikacji z którym użyto biblioteki fusco z kolei w przypadku bazy MySQL, do komunikacji z którą użyto protokołu binarnego, były to biblioteki bank oraz bank_mysql;

- w implementacji referencyjnej zmieniany jest tylko wewnętrzny stan bazy danych, w implementacjach silników z kolei dodatkowo zwracane są informacje o tym, których lamp dotyczy zmiana stanu bazy danych, tak aby agent mógł przesłać agentom niższego poziomu, odpowiadającym bezpośrednio za sterowanie urządzeniami oświetleniowymi, odpowiednie sygnały sterujące;
- stan bazy danych przechowującej graf w implementacji silników przetwarzających graf jest trwały (zachowywany pomiędzy kolejnymi uruchomieniami systemu), w przeciwieństwie do implementacji w SWI-Prolog.

Powyższe różnice powodują dodatkowy narzut obliczeniowy w trakcie przeprowadzania eksperymentów. Wszystkie czasy zaprezentowane poniżej są czasami eksperymentów wykonanych z perspektywy implementacji agenta w Erlangu, zawierają one więc ww. narzut czasowy.

4. Wymienne moduły przetwarzania grafu

4.1 Konfiguracja odpowiedniego silnika

Konfiguracji wybranego silnika przetwarzania grafu dokonuje się w pliku `etc/lighting.cfg` poprzez modyfikację klucza `backend`.

Moduł wybierany jest w parametrze `module` np. dla silnika opartego na Mnesii:

```
{backend, [{module, lighting_graph_mnesia}]}.
```

Jeżeli moduł potrzebuje pewnych dodatkowych informacji do poprawnego funkcjonowania, również należy podać je w konfiguracji. Na przykład, dla modułu Neo4j należy również podać adres oraz port, na którym baza danych nasłuchuje na zapytania po interfejsie REST:

```
{backend, [{module, lighting_graph_neo4j},
           {host, "localhost"},
           {port, 7474}]}.
```

Dla MySQL z kolei należy podać sterownik aplikacji **bank**, który będzie łączył się z bazą danych, argumenty do jego wystartowania (w przypadku **bank_mysql** są to kolejno: adres bazy danych, port na którym baza nasłuchuje na połączenia, użytkownik, hasło oraz wybierana baza danych):

```
{backend, [{module, lighting_graph_mysql},
           {driver, bank_mysql},
           {args, ["localhost", 3306, "slic", "slic", "slic"]},
           {connectors, 10}]}.
```

Po uruchomieniu aplikacji, odpowiednie operacje na bazie danych mogą być wykonywane poprzez odwołanie się do odpowiednich funkcji modułu `lighting_graph`. Wywołanie funkcji zostanie delegowane do skonfigurowanego modułu.

4.2 Funkcje eksportowane z modułów silnika

Funkcje, które powinny zostać wyeksportowane z modułów, które implementują logikę silnika przetwarzania grafu zostały zdefiniowane w *behaviourze* w pliku źródłowym `lighting_graph.erl`.
Są to:

- `start() -> ok.`
  Funkcja wywoływana przy starcie danego modułu, zawsze zwraca `ok`.
- `init() -> ok.`
  Funkcja inicjalizująca stan grafu, odpowiednik predykatu `init/0` w referencyjnym rozwiązaniu w Prologu. Zawsze zwraca `ok`.
- `v(Type::atom(), Id::integer()) -> {Type::atom(),Id::integer()}.`
  Funkcja wstawiająca wierzchołek typu `Type` o identyfikatorze `Id` do bazy danych. W przypadku powodzenia zwraca krotkę odpowiadającą temu wierzchołkowi.
- `e(Type1::atom(), Id1::integer(), Type2::atom(), Id2::integer()) -> {Type1::atom(),Id1::integer(),Type2::atom(),Id2::integer()}.`
  Funkcja wstawiająca krawędź między wierzchołkiem typu `Type1` o identyfikatorze `Id1` a wierzchołkiem typu `Type2` o identyfikatorze `Id2` bez etykiety. W przypadku powodzenia zwraca krotkę odpowiadającą tej krawędzi.
- `e(Type1::atom(), Id1::integer(), Type2::atom(), Id2::integer(), Label::atom()) -> {Type1::atom(),Id1::integer(), Type2::atom(),Id2::integer(),Label::atom()}.`
  Funkcja wstawiająca krawędź między wierzchołkiem typu `Type1` o identyfikatorze `Id1` a wierzchołkiem typu `Type2` o identyfikatorze `Id2` o etykiecie `Label`. W przypadku powodzenia zwraca krotkę odpowiadającą tej krawędzi.
- `set(Old::tuple(),New::tuple()) -> New::tuple().`
  Funkcja ustawiająca wartość krotki odpowiadającej odpowiedniemu predykatowi ustawiającego etykietę wierzchołka, przeznaczona do zmiany wartości sensorów. W przypadku powodzenia zwracana jest ustawiona wartość.

- `rules() -> list().`
  Funkcja zwracająca listę reguł przetwarzania grafu specyficznych dla każdego z silników. W przypadku Mnesii są to zapytania w utworzonym do tego celu języku, Neo4j - zapytania w języku Cypher, MySQL - zapytanie wywołujące odpowiednią procedurę składowaną w bazie danych.
- `match() -> [tuple()].`
  Funkcja dokonująca przetworzenia grafu z wykorzystaniem reguł zwracanych przez funkcję `rules/0`. Wynikiem zwracanym jest lista krotek reprezentujących urządzenia świetlne wraz z nowymi wartościami ich parametrów po przetworzeniu grafu, postaci `{l,Id,Par,on|off}`.
- `match(Rules::list()) -> [tuple()].`
  Jak funkcja `match/0` z tą różnicą, że dopasowanie jest dokonywane z użyciem reguł przekazanych w argumencie. Reguły te są specyficzne dla każdego z zaimplementowanych silników przetwarzania grafu.
- `set_match(Old::tuple(),New::tuple()) -> [tuple()].`
  Złączenie funkcji `set/2` i `match/0`, które wykonywane są w jednej transakcji. Wartość zwracana jest taka sama jwk w przypadku funkcji `match/0`.
- `clear() -> ok.`
  Funkcja czyszcząca wszystkie informacje z bazy danych. W przypadku powodzenia wartością zwaracaną jest `ok`.

5. Mnesia

Implementacja silnika znajduje się w pliku źródłowym `lighting_graph_mnesia.erl`.

5.1 Struktura bazy danych

Baza danych składa się z następujących tabel:

Tabela	Typ	Lokalizacja	Pola	Indeksy
pred	set	disc_copies	key,functor,secondary,tertiary	functor,secondary,tertiary
c	set	disc_copies	id,value	brak
s	bag	ram_copies	id,value	brak

Struktura bazy danych została utworzona na wzór bazy wiedzy w Prologu. W bazie danych przechowywane są krotki będące odzwierciedleniem predykatów dynamicznych w rozwiązaniu referencyjnym (np. predykatowi `v/3` odpowiada krotka `{v,_,_,_}`).

Dodatkowo przechowywanymi informacjami są indeksy, które mogą okazać się przydatne przy operacjach odczytu rekordów z bazy w trakcie dopasowywania

wzorca do grafu. Dla każdego predykatu przechowywane są trzy typu indeksów:

- `functor`, zawierający informację o tym jakiego predykatu dynamicznego reprezentantem jest dana krotka, np. dla krotki `{v,_,_,_}` wartością indeksu jest `{v,3}`;
- `secondary`, zawierający informacje o wierzchołkach początkowych i końcowych danej krawędzi o danej etykiecie, niezależnie od identyfikatora wierzchołka końcowego. W przypadku wierzchołka jest to informacja o typie i identyfikatorze tego wierzchołka niezależnie od jego etykiet;
- `tertiary`, zawierający informacje o wierzchołkach początkowych i końcowych danej krawędzi niezależnie od jej etykiety, niezależnie od identyfikatora wierzchołka końcowego.

W sytuacjach, w których typ predykatu powoduje, że przechowywanie indeksów nie jest konieczne, a operacje ich dodawania i usuwania do tabeli indeksowanych predykatów wprowadzają zbyt duży narzut czasowy, dla tych predykatów stworzono osobne tabele. Dodatkowo, ponieważ predykaty typu `s/2` tworzone są tylko na czas przetwarzania grafu, tabela je zawierająca nie jest trwała a jej rekordy przechowywane są tylko w pamięci RAM.

5.2 Implementacja reguł

Implementację reguł dla tego silnika przetwarzania grafu można znaleźć w pliku źródłowym `lighting_graph_mnesia_rules.erl`.

Aby umożliwić przetwarzanie reguł wprowadzono język przetwarzania grafu oraz logikę przetwarzającą wyrażenia w nim zapisane. Logika przetwarza zapisane reguły i tłumaczy je na operacje wykonywane na bazie danych Mnesia. Implementacja logiki przetwarzania znajduje się w pliku źródłowym `lighting_graph_mnesia.erl`.

Reguła składa się z dwóch części: wzorca dopasowywanego oraz akcji wykonywanej w przypadku pomyślnego dopasowania.

Przykładową regułę, będącą odpowiednikiem reguły *1a1b* w implementacji referencyjnej, zaprezentowano poniżej:

```
R1 = [{v,k,'_',false,'A'},
      {e,'A',s,'_','B'},
      {'not',v,'B',off},
      {e,'B',c,'_','_','C'}],
A1 = [{'B', fun({s,Id}) -> assert({s,Id,off}) end}],
{R1, A1}.
```

Zapis wzorca jest analogiczny do zapisu w Prologu z dokładnością do sposobu zapisu poszczególnych warunków (zapis przy użyciu krotek zamiast funktorów

i operatorów). Dodatkową różnicą jest wiązanie zmiennych do całych wierzchołków, a nie tylko identyfikatorów, wiązanie występuje w regule po dopasowaniu konkretnego wierzchołka.
W celu dopasowania dowolnej wartości w danym polu (*wildcard*), w regule należy użyć atomu `'_'`.

Akcja wykonywana po dopasowaniu definiowana jest jako lista związanych wcześniej zmiennych oraz lambdy, której argumentami są wartości tych zmiennych. Jeżeli funkcja przyjmuje jeden argument, w akcji może być zdefiniowana tylko jedna zmienna, w przeciwnym wypadku powinna być to lista zmiennych.

Mnesia zawiera wbudowany mechanizm transakcyjny pozwalający na zachowanie spójności danych gdy na jednej tabeli operacje wykonują różne procesy (mogą znajdować się one na różnych węzłach). Użycie systemu transakcyjnego, ze względu na dużą liczbę operacji odczytu z bazy danych, do implementacji silnika przetwarzania grafu okazuje się być ok. 20 razy wolniejsze od użycia operacji z jego pominięciem. Dlatego też w implementacji wybrano to drugie podejście. Jego wadą jest konieczność własnoręcznej implementacji zarządzania dostępem do silnika przetwarzającego graf. Jest to możliwe do wykonania przy użyciu Erlangowych *behaviourów* `gen_server` oraz `gen_leader`[3].

5.3 Optymalizacja zapytań

Wszystkie operacje wykonywane na bazie danych Mnesia wykonywane są na tej samej instancji maszyny wirtualnej Erlanga (BEAM) i poprzez bezpośrednie wywoływanie funkcji z odpowiednich modułów, nie zaś poprzez specjalny język zapytań. Dzięki temu możliwe jest profilowanie wykonywanych operacji pod kątem wydajnościowym przy użyciu narzędzi uzywających wbudowanego systemu śledzenia wywołań funkcji w Erlangu, jak np. EEP. Narzędzie to generuje czasowy ślad wywołań funkcji w danym okresie czasu, który może zostać analizowany przy użyciu narzędzia qcachegrind.

6. Neo4j

Implementacja silnika znajduje się w pliku źródłowym `lighting_graph_neo4j.erl`.

6.1 Struktura bazy danych

Ze względu na to, że grafowa baza danych Neo4j została zaprojektowana do przechowywania grafów, w zasadzie nie ma konieczności przygotowywania żadnej organizacji danych w bazie.

Wyjątek stanowią indeksy, które mogą być dwojakiego typu: **schema indexes** or **legacy indexes**.

[3] https://github.com/lehoff/gen_leader

Pierwszy z tych indeksów może istnieć wyłącznie dla wierzchołków w grafie i musi zostać utworzony jawnie dla etykiety danego rodzaju wierzchołka.
Drugi z nich z kolei może istnieć zarówno dla wierzchołków jak i krawędzi w grafie oraz może być tworzony automatycznie dla danego rodzaju etykiety niezależnie od typu wierzchołka lub krawędzi.

W przypadku **legacy indexes** ręczne tworzenie indeksów jest jednak problematyczne, gdyż aktualnie możliwe jest to tylko przy pomocy konsoli bazy Neo4j lub interfejsu w Javie. Tworzenie **schema indexes** możliwe jest poprzez język zapytań Cypher. Oprócz tego, **legacy indexes** mogą być użyte zaraz po słowie kluczowym `START`, znajdującym początkowy węzeł lub krawędź do dalszego przetwarzania w zapytaniach. **Schema indexes** z kolei mogą być użyte po słowie kluczowym `MATCH` co bardziej pasuje do charakteru zapytań utworzonych z reguł przetwarzania grafu.

Poniżej znajduje się informacja z konsoli Neo4j o utworzonych i aktywnych **schema indexes**.

```
neo4j-sh (?)$ schema
Indexes
  ON :c(id)       ONLINE
  ON :c(engaged)  ONLINE
  ON :d(detected) ONLINE
  ON :d(id)       ONLINE
  ON :h(detected) ONLINE
  ON :h(id)       ONLINE
  ON :k(detected) ONLINE
  ON :k(id)       ONLINE
  ON :p(detected) ONLINE
  ON :p(id)       ONLINE
  ON :s(low)      ONLINE
  ON :s(off)      ONLINE
  ON :s(high)     ONLINE
  ON :s(id)       ONLINE
```

6.2 Implementacja reguł

Reguły zaimplementowane w języku zapytań Cypher[4] można znaleźć w pliku źródłowym `lighting_graph_neo4j_rules.erl`. Wyjaśnienie znaczenia słów kluczowych w zapytaniach można znaleźć w dokumentacji języka Cypher.

6.3 Optymalizacja zapytań

Konsola bazy danych Neo4j udostępnia narzędzie do profilowania zapytań w języku Cypher.

[4] http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html

W trakcie implementacji zapytań z regułami przy profilowaniu można było dojść do dwóch wniosków:

1. Próba dopasowania od razu całego wzorca do reguł może wiązać się z bardzo dużą liczbą zbędnych odczytów z bazy danych. Jest to spowodowane tym, że indeksy używane są tylko do znalezienia startowych węzłów (wierzchołków), kolejne pasujące krawędzie i wierzchołki dopasowywane są na zasadzie przechodzenia grafu. Najpierw wyciągane są wszystkie potencjalne dopasowania a dopiero później dokonywane jest filtrowanie predykatów zawartych po słowie kluczowym `WHERE`.
   Rozwiązanie tego problemu możliwe jest przy użyciu słowa kluczowego `WITH` z języka Cypher i dopasowywanie wzorca fragmentami w zapytaniu. Wpływa to znacznie na długość i czytelność zapytania, jednak jak można zauważyć na poniższych przykładach uruchomienia profilowania zapytań, wiąże się to ze znacznym zyskiem wydajnościowym (por. liczbę *__db_hits* w obu zapytaniach).

```
neo4j-sh (?)$ PROFILE MATCH (d:d)-[:dir_day]->(s:s)<-[]-(h:h),
              (k:k)-[]->(s)-[:high]->(c:c)
              USING INDEX d:d(detected)
              WHERE d.detected = false AND s.off=false AND h.id = 1
              AND h.detected = "day" AND k.detected = true
              WITH DISTINCT s SET s.off = true;
+-------------------+
| No data returned. |
+-------------------+
Properties set: 100

EmptyResult(_rows=0, _db_hits=0)
UpdateGraph(commands=["PropertySetAction(Product(s,off,true),true)"],
            _rows=100, _db_hits=100)
  EagerAggregation(keys=["s"], aggregates=[], _rows=100, _db_hits=0)
    Filter(pred="(((((((Product(s,off(4),true) == NOT(true)
                  AND Product(h,id(0),true) == Literal(1))
                  AND Product(h,detected(5),true) == Literal(day))
                  AND Product(k,detected(5),true) == true)
                  AND hasLabel(h:h(4)))
                  AND hasLabel(k:k(3)))
                  AND hasLabel(c:c(0)))
                  AND hasLabel(s:s(1)))
                  AND hasLabel(s:s(1)))",
           _rows=600, _db_hits=2400)
      PatternMatch(g="(s)-['  UNNAMED53']-(c),(d)-['  UNNAMED11']-(s),
                      (h)-['  UNNAMED29']-(s),(k)-['  UNNAMED45']-(s)",
                   _rows=600, _db_hits=79877)
        SchemaIndex(identifier="d", _db_hits=0, _rows=800, label="d",
```

```
                    query="NOT(true)", property="detected")

neo4j-sh (?)$ PROFILE MATCH (d:d)-[:dir_day]->(s:s)
              WHERE d.detected = false AND s.off=false
              WITH DISTINCT s
              MATCH (h:h)-[]->(s)
              WHERE h.id = 1 AND h.detected = "day"
              WITH DISTINCT s
              MATCH (k:k)-[]->(s)-[:high]->(c:c)
              WHERE k.detected = true
              SET s.off = true;
+--------------------+
| No data returned. |
+--------------------+
Properties set: 100

EmptyResult(_rows=0, _db_hits=0)
UpdateGraph(commands=["PropertySetAction(Product(s,off,true),true)"],
            _rows=100, _db_hits=100)
  Eager(_rows=100, _db_hits=0)
    Filter(pred="((Product(k,detected(5),true) == true
                AND hasLabel(k:k(3)))
                AND hasLabel(c:c(0)))",
           _rows=100, _db_hits=100)
      PatternMatch(g="(k)-['  UNNAMED172']-(s),(s)-['  UNNAMED180']-(c)",
                   _rows=100, _db_hits=900)
        EagerAggregation(keys=["s"], aggregates=[], _rows=100, _db_hits=0)
          Filter(pred="((Product(h,id(0),true) == Literal(1)
                      AND Product(h,detected(5),true) == Literal(day))
                      AND hasLabel(h:h(4)))",
                 _rows=100, _db_hits=200)
            PatternMatch(g="(h)-['  UNNAMED98']-(s)", _rows=100, _db_hits=1102)
              EagerAggregation(keys=["s"], aggregates=[], _rows=100, _db_hits=0)
                Filter(pred="(Product(s,off(4),true) == NOT(true)
                            AND hasLabel(s:s(1)))",
                       _rows=600, _db_hits=600)
                  PatternMatch(g="(d)-['  UNNAMED11']-(s)",
                               _rows=600, _db_hits=600)
                    SchemaIndex(identifier="d", _db_hits=0, _rows=800, label="d",
                                query="NOT(true)", property="detected");
```

2. Planer zapytań niekoniecznie wybiera najbardziej optymalne indeksy do użycia w wykonaniu zapytania, dlatego czasami należy dać mu podpowiedź poprzez użycie słowa kluczowego `USING INDEX`. Efekt jego użycia można sprawdzić poprzez profilowanie nowego zapytania, jak w punkcie 1.

7. MySQL

Implementacja silnika znajduje się w pliku źródłowym `lighting_graph_mysql.erl`.

7.1 Struktura bazy danych

Ze względu na to, że w tym przypadku została wykorzystana relacyjna baza danych, która nie jest przeznaczona do przechowywania danych grafowych należało przystosować strkturę danych do przechowywania elementów grafu w celu optymalizacji szybkości wykonywanych zapytań.

Poszczególne tabele reprezentują krawędzie oraz węzły. Rekordy dla każdego typu krawędzi/węzła umieszczone są w osobnej tabeli. Podział krawędzi wynika z typu węzłów znajdujących się na końcu krawędzi. Struktura odwzorowuje wszystkie informacje dotyczące grafu z referencyjnego rozwiązania zrealizowanego w języku Prolog.

Poniżej znajduje się lista tabel umieszczonych w bazie **slic**.

Tabela	Typ	Atrybuty (typ)	Indeksy
eas	BASE TABLE	f(smallint), t(smallint)	f,t
ecl	BASE TABLE	f(smallint), t(smallint), l(char)	f,t,l
eds	BASE TABLE	f(smallint), t(smallint), l(char)	f,t,l
ehs	BASE TABLE	f(smallint), t(smallint)	f,t
eks	BASE TABLE	f(smallint), t(smallint)	f,t
eps	BASE TABLE	f(smallint), t(smallint), l(char)	f,t,l
esc	BASE TABLE	f(smallint), t(smallint), l(char)	f,t,l
res	BASE TABLE	l_id(int), l_label(char), new_c(tinyint)	brak
va	BASE TABLE	id(smallint)	brak
vc	BASE TABLE	id(smallint), engaded(tinyint)	brak
vd	BASE TABLE	id(smallint), detected(tinyint)	brak
vh	BASE TABLE	id(smallint), day(tinyint)	brak
vk	BASE TABLE	id(smallint), detected(tinyint)	brak
vl	BASE TABLE	id(smallint)	brak
vp	BASE TABLE	id(smallint), detected(tinyint)	brak
vs	BASE TABLE	id(smallint), high(tinyint), low(tinyint), off(tinyint)	brak

7.2 Implementacja reguł

Reguły zostały zaimplementowane za pomocą języka SQL. Przykładowa procedura (będąca odpowiednikiem reguły *1a1b* z implementacji referencyjnej) przedstawia się następująco:

```
CREATE PROCEDURE rule_1()
BEGIN
    UPDATE vs
    SET off = true
    WHERE vs.id in (
        select distinct esc.f from vk
        join eks on vk.id = eks.f
        join esc on eks.t = esc.f
            where vk.detected = false
        ) and vs.off = false;
END;
```

7.3 Optymalizacja zapytań

Silnik MySQL pozwala na analizę zapytań za pomocą polecenia `EXPLAIN`. Analiza poszczególnych zapytań pozwoliła na wybranie atrybutów na które zostały nałożone indeksy.

MySQL umożliwia indeksowanie tabel pozwalając na wyszukiwanie poszczególnych rekordów bez przeszukiwania całej tabeli, co znacząco przyspiesza wykonywanie zapytań. Domyślnie indeksy nakładane są na wszystkie klucze główne tabel. W celu przyspieszenia wykonywania zapytań nałożono dodatkowe indeksy na tabele zawierające rekordy przechowujące dane na temat krawędzi grafu. Wszystkie nałożone indeksy są typu B-Tree.

Na tabele nałożono następujące indeksy:

```
CREATE INDEX eps_f ON eps(f);
CREATE INDEX eps_t ON eps(t);
CREATE INDEX eps_l ON eps(l);

CREATE INDEX eas_f ON eas(f);
CREATE INDEX eas_t ON eas(t);

CREATE INDEX eks_f ON eks(f);
CREATE INDEX eks_t ON eks(t);

CREATE INDEX ehs_f ON ehs(f);
CREATE INDEX ehs_t ON ehs(t);
```

```
CREATE INDEX eds_f ON eds(f);
CREATE INDEX eds_t ON eds(t);
CREATE INDEX eds_l ON eds(l);

CREATE INDEX esc_f ON esc(f);
CREATE INDEX esc_t ON esc(t);
CREATE INDEX esc_l ON esc(l);

CREATE INDEX ecl_f ON ecl(f);
CREATE INDEX ecl_t ON ecl(t);
CREATE INDEX ecl_l ON ecl(l);
```

Wykorzystane indeksy pozwoliły na osiągniecie wyników znacząco poprawiające czasy uzyskane w implementacji referencyjnej w języku Prolog.

Opytamalizację wykonywania zapytań wyraźnie widać na poniższym przykładzie. Jest to zapytanie z reguły *5b*.

```
SELECT * FROM vs
    WHERE vs.id in (
        select distinct esc.f from vk
        cross join vh
        cross join eds
        join vd on vd.id = eds.f
        join eks on eks.f = vk.id and eks.t = eds.t
        join ehs on ehs.f = vh.id and ehs.t = eks.t
        join esc on esc.f = ehs.t
            where vk.detected = true and vh.day = false and eds.l = 'dir_night'
                  and vd.detected = false and esc.l = 'high'
        ) and vs.off = false;
```

Wyniki zapytania przy niezindeksowanej bazie:
`100 rows in set (1.61 sec)`

Wynik polecenia `EXPLAIN`:

```
+----+--------------------+-------+--------+---------------+---------+---------+
| id | select_type        | table | type   | possible_keys | key     | key_len |
+----+--------------------+-------+--------+---------------+---------+---------+
|  1 | PRIMARY            | vs    | ALL    | NULL          | NULL    | NULL    |
|  2 | DEPENDENT SUBQUERY | vk    | ALL    | PRIMARY       | NULL    | NULL    |
|  2 | DEPENDENT SUBQUERY | vh    | ALL    | PRIMARY       | NULL    | NULL    |
|  2 | DEPENDENT SUBQUERY | eks   | eq_ref | PRIMARY       | PRIMARY | 4       |
|  2 | DEPENDENT SUBQUERY | ehs   | eq_ref | PRIMARY       | PRIMARY | 4       |
|  2 | DEPENDENT SUBQUERY | esc   | ref    | PRIMARY       | PRIMARY | 2       |
|  2 | DEPENDENT SUBQUERY | vd    | ALL    | PRIMARY       | NULL    | NULL    |
```

```
|  2 | DEPENDENT SUBQUERY | eds   | eq_ref | PRIMARY       | PRIMARY | 4       |
+----+--------------------+-------+--------+---------------+---------+---------+
+-----------------+------+-----------------------------------------+
| ref             | rows | Extra                                   |
+-----------------+------+-----------------------------------------+
| NULL            | 1181 | Using where                             |
| NULL            |    1 | Using where; Using temporary            |
| NULL            |    1 | Using where; Using join buffer          |
| slic.vk.id,func |    1 | Using index                             |
| slic.vh.id,func |    1 | Using where; Using index                |
| func            |    1 | Using where                             |
| NULL            |  925 | Using where; Distinct; Using join buffer |
| slic.vd.id,func |    1 | Using where; Distinct                   |
+-----------------+------+-----------------------------------------+
```

Wyniki zapytania przy zindeksowanej bazie (pomiędzy wykonywaniem zapytań wykonano polecenie `flush tables` w celu usunięcia cache bazy):
`100 rows in set (0.02 sec)`

Wynik polecenia `EXPLAIN`:

```
+----+--------------------+-------+--------+---------------------------+---------+
| id | select_type        | table | type   | possible_keys             | key     |
+----+--------------------+-------+--------+---------------------------+---------+
|  1 | PRIMARY            | vs    | ALL    | NULL                      | NULL    |
|  2 | DEPENDENT SUBQUERY | vk    | ALL    | PRIMARY                   | NULL    |
|  2 | DEPENDENT SUBQUERY | vh    | ALL    | PRIMARY                   | NULL    |
|  2 | DEPENDENT SUBQUERY | esc   | ref    | PRIMARY,esc_f,esc_l       | PRIMARY |
|  2 | DEPENDENT SUBQUERY | eds   | ref    | PRIMARY,eds_f,eds_t,eds_l | eds_t   |
|  2 | DEPENDENT SUBQUERY | vd    | eq_ref | PRIMARY                   | PRIMARY |
|  2 | DEPENDENT SUBQUERY | eks   | eq_ref | PRIMARY,eks_f,eks_t       | PRIMARY |
|  2 | DEPENDENT SUBQUERY | ehs   | eq_ref | PRIMARY,ehs_f,ehs_t       | PRIMARY |
+----+--------------------+-------+--------+---------------------------+---------+
+---------+----------------+------+-----------------------------------+
| key_len | ref            | rows | Extra                             |
+---------+----------------+------+-----------------------------------+
| NULL    | NULL           | 1181 | Using where                       |
| NULL    | NULL           |    1 | Using where; Using temporary      |
| NULL    | NULL           |    1 | Using where; Using join buffer    |
| 2       | func           |    1 | Using where                       |
| 2       | func           |    3 | Using where; Distinct             |
| 2       | slic.eds.f     |    1 | Using where; Distinct             |
| 4       | slic.vk.id,func |    1 | Using where; Using index; Distinct |
| 4       | slic.vh.id,func |    1 | Using where; Using index; Distinct |
+---------+----------------+------+-----------------------------------+
```

8. Czasy wykonania eksperymentów

W poniższej tabeli zestawiono czasy wykonania poszczególnych eksperymentów przy użyciu poszczególnych silników przetwarzania grafu w milisekundach w zestawieniu do implementacji referencyjnej w języku Prolog.

Eksperyment	Mnesia	Neo4j	MySQL	SWI-Prolog 5.8.2
ex0	55	75	27	3
ex1	187	211	176	289
ex2	128	161	164	301
ex3	119	149	165	304
ex4	115	154	154	306
ex5	118	150	185	330
ex6	116	152	163	312
ex7	116	154	176	325
ex8	112	169	158	317
ex9	119	152	168	325
Suma	**1185**	**1527**	**1536**	**2812**
% czasu ref.	**42,14%**	**54,30%**	**54,62%**	**100%**

9. Wnioski

Dokonano implementacji silnika przetwarzania grafu odwzorowującego system sterowania oświetleniem z użyciem trzech technologii bazodanowych:

1. Mnesia (baza danych klucz-wartość)
2. Neo4j (grafowa baza danych)
3. MySQL (relacyjna baza danych)

We wszystkich trzech przypadkach udało się opracować strukturę bazy danych (o mniejszym lub większym stopniu skomplikowania) oraz zapytania, których użycie wiąże się z mniejszym kosztem czasowym niż w przypadku referencyjnej implementacji w języku Prolog w środowisku SWI-Prolog 5.8.2. Użycie SWI-Prolog w wersji 6 lub wyższej pozwala na uzyskanie dużo lepszych czasów (wręcz pomijalnych) dzięki zastosowaniu techniki *just-in-time indexing*.

Każda z trzech implementacji wykonuje odpowiednie operacje w zbliżonym do dwóch pozostałych czasie. Najszybsza z nich jest jednak implementacja w

Mnesii, co jest związane z operacjami na bazie danych znajdującej się w tej samej przestrzeni adresowej co program agenta wyższego rzędu i co za tym idzie, nie jest konieczne użycie żadnych interfejsów (jak REST czy protokół binarny) by skomunikować się z bazą danych.