Uniwersytet w Białymstoku

Wydział Matematyczno-Fizyczny

Instytut Matematyki

Sławomir Nieciecki

Transakcje SQL w Meta-HTML

Praca dyplomowa napisana
pod kierunkiem
dr. Mariusza Żynela

Białystok 2004

Składam serdeczne podziękowania
dr. Mariuszowi Żynelowi
za pomoc w przygotowaniu
i realizacji tego projektu.

Sławomir Nieciecki

Spis treści

Wstęp

Jedną z najbardziej popularnych baz danych jest baza MySQL natomiast intensywnie eksploatowanym oprogramowaniem do budowy dynamicznych stron HTML jest Meta-HTML. Zarówno MySQL jak i Meta-HTML są dostępne jako Open Source. Daje to użytkownikom możliwość dokonywania samodzielnych ulepszeń i poprawek. Ponieważ Meta-HTML nie jest zbyt populany wśród autorów stron przez to nie jest aktywnie rozwijany. Spowodowało to, że nie wszystkie możliwości najnowszej wersji MySQL są dostępne w Meta-HTML. Wśród nich znajdują się transakcje – bardzo wygodne narzędzie zapewniające integralność danych.

Celem mojej pracy jest dostosowanie oprogramowania Meta-HTML do zmian, które nastąpiły w API serwera baz danych MySQL oraz dodanie funkcji umożliwiających łatwe korzystanie z transakcji.

Oba postawione zadania udało się pomyślnie zrealizować. Wymagały one dobrej orientacji w kodzie źródłowym Meta-HTML, w API bazy MySQL oraz w używanym wraz z komilatorem oprogramowaniem pomocniczym. Z załączonego w mojej pracy opisu wykonanych modyfikacji można odnieść wrażenie, że postawione zadanie było łatwe. Zanim jednak otworzyłem edytor w celu nanoszenia pierwszych poprawek minęło sporo czasu. Wcześniej musiałem zapoznać się z API MySQL, wychwycić zmiany pomiędzy poszczególnymi wersjami oraz nauczyć się bazy MySQL i transakcji. Ta część zadania była o tyle łatwa, że dużo jest literatury książkowej i opisów w inernecie poświęconych MySQL. Dużo gorzej wyglądał sprawa z Meta-HTML. Tutaj w przypadku wątpliwości wynikających z ubogiej dokumentacji należało sięgać bezpośrednio do kodu źródłowego liczącego w sumie 195118 lini, aby wyjaśnić sposób działania Meta-HTML.

Poza wykonaniem zmian w kodzie Meta-HTML napisałem uproszczony sklem internetowy tak aby mieć możliwość testowania. Być może jest to nadmiernie rozbudowany test ale służył mi jako ilustracja praktycznego i sensownego wykorzystania transakcji. Tak więc napisane w języku Meta-HTML skrypty można traktować jako przykład używania transakcji.

Rozdział 1

Nowe rozwiązania w MySQL

1.1 Transakcje

MySQL jest jednym z najpopularniejszych interface'ów języka SQL (Structured Query Language - Strukturalny Język Zapytań). MySQL jest dostępny publicznie od 1996 roku, ale historia jego tworzenia sięga 1979. Obok PostgreSQL, MySQL to jedna z dwóch najpopularniejszych wolno dostępnych relacyjnych baz danych (relational database management system - RDBMS).

MySQL jest obecnie dostępny w licencji Open Source oznacza to dostęp do jego kodu źródłowego, który można bezpłatnie wykorzystywać, zmieniać i redystrybuować, ale w konkretnych przypadkach można również uzyskać licencje komercyjne. MySQL jest dostępny na licencji GNU GENERAL PUBLIC LICENSE.

Aktualnie MySQL jest rozpowszechniany w wersji 4.x. Zmiany pomiędzy wersją 3.x a 4.x polegają na dodaniu transakcji SQL oraz wprowadzeniu nowego formatu tabeli InnoDB.

Kluczowe cechy MySQL:

- jest obsługiwany przez wiele innych programów i języków,
- interfejsy programowe dla: C, C++, Java, Perl, PHP, Meta-HTML,
- integracja bazy z serwerem Apache,
- pozwala na użycie wielu typów danych: liczby całkowite ze/bez znaku w postaci: 1, 2, 3, 4, 8 bajtowej, oraz typy INT, FLOAT, DOUBLE, CHAR, VARCHAR, TEXT, BLOB, DATE, TIME, DATETIME, TIMESTAMP, YEAR, SET i ENUM,
- pełna obsługa: SELECT, WHERE, GROUP BY, ORDER BY i funkcji agregujących,
- stała bądź zmienna długość rekordów,

- tablice haszujące trzymane w pamięci operacyjnej dla polepszenia wydajności,
- obsługa bardzo dużych baz danych (do 50,000,000 rekordów),
- dostęp do bazy danych poprzez TCP/IP, gniazdka UNIX lub nazwane potoki w NT.

MySQL jest dostępny na wielu platformach. W większości przypadków wykorzystywane są wątki wewnętrzne systemów operacyjnych:

- Win9x, NT, 2k, XP,
- Apple MacOS X,
- Sun Solaris 2.5+,
- Linux,
- IBM AIX 4+,
- HP UX 11+,
- w przyszłości QNX i Novell.

W MySQL'u istnieje kilka typów tabel, które możemy stworzyć. Poprzez odpowiednie ich dołączenie i dopasowanie do potrzeb, możemy uzyskać zadziwiający wzrost wydajności. Oto lista obecnie istniejących typów:

MyISAM - domyślny typ tabel; szybki, nie wspomaga transakcji.

ISAM - starsza wersja typu MyISAM o mniejszych możliwościach i mniejszej wydajności. Zostanie całkowicie usunięta w MySQL 5.0. Nie wspomaga transakcji.

HEAP - ten typ jest znacznie szybszy od wszystkich innych. Wynika to z tego, iż składowany jest wyłącznie w pamięci RAM. Jeśli MySQL padnie, danych tam zawartych nie będzie można już odzyskać. Również nie wspomaga transakcji; autorzy zalecają go wykorzystywanie w tabelach tymczasowych (temporary tables).

BerkeleyDB (BDB) - za typ ten nie odpowiadają ludzie z ekipy MySQL AB; dlatego też domyślnie obecny jest tylko w pakiecie MySQL-MAX. Wspomaga transakcje, lecz tabele w tym formacie działają najwolniej ze wszystkich.

InnoDB - również wspomaga transakcje, jest jednym z wbudowanych typów. Największą wydajność osiąga na wielo-gigabajtowych bazach z bardzo dużą ilością odwołań w ciągu sekundy. Cechą szczególną ostatnich dwóch typów jest to, iż w przypadku awarii MySQL'a najłatwiej odzyskać z nich dane oraz to iż umożliwiają przeprowadzenie transakcji.

Obsługa transakcji jest bardzo ważna w bazach danych. Transakcja to zbiór operacji które mogą być wykonane jedynie wszystkie lub żadna. Nazwa pochodzi od operacji bankowych - przelew musi jednocześnie zabrać z jednego konta i dodać na drugie. W przypadku niepowodzenia żadna z tych operacji nie powinna mieć miejsca. Jeśli zajdzie tylko jedna skutki byłyby potencjalnie katastrofalne.

Transakcje opisuje zasada ACID - Atomicity, Consistency, Isolation, and Durability:

Atomicity - transakcja może być wykonana tylko w całości albo wcale.

Consistency - stan bazy danych zawsze przedstawia stan przed lub po transakcji. Zapytania składane systemowi w czasie wykonywania transakcji muszą pokazywać sytuacje przed transakcją, nie sytuacje przejściową.

Isolation - transakcja dzieje się niezależnie od innych wykonywanych operacji, w tym od innych transakcji.

Durability - w przypadku krachu całego systemu bazodanowego, np.: w wyniku odcięcia elektryczności, transakcja będzie albo wykonana w całości albo wcale.

Najprostszym przykładem jest przelew pieniędzy. Jeśli wydajemy dyspozycję o przelaniu pieniędzy na czyjeś konto - rozpoczynamy transakcję. Zakończenie transakcji nastepuje dopiero kiedy bank wypełni papierkowa robotę i fizycznie one się tam znajda. Jeśli coś pójdzie nie tak transakcja zostanie wycofana.

W czasie transakcji dane na plikach zmieniane są przez systemy plików z journallingiem. Journalling - księgowanie to w informatyce termin związany z konstrukcją baz danych oraz systemów plików. Przy użyciu journallingu dane nie są od razu zapisywane na dysk, tylko zapisywane w swoistym dzienniku (ang. journal. Dzięki takiemu mechanizmowi działania zmniejsza się prawdopodobieństwo utraty danych - jeśli utrata zasilania nastąpiła w trakcie zapisu - zapis zostanie dokończony po przywróceniu zasilania, jeśli przed - stracimy tylko ostatnio naniesione poprawki, a oryginalny plik pozostanie nietknięty). Gwarantuje to że system plików jest stabilny nawet pomimo krachu systemu operacyjnego.
Sam zapis do plików wykonywany transakcyjnie, to zbyt kosztowne rozwiązanie. Istnieją jednak metody transakcyjnego zapisu danych do systemu plików - najprostsza to (na Uniksach):

- zapisujemy plik tymczasowy w którym znajdują się nowe dane. W przypadku krachu w tej fazie mamy stary plik nienaruszony,
- kasuje się poprzedni plik. Operacja jest atomowa. W przypadku krachu przed skasowaniem mamy oba pliki, w przypadku krachu po kasowaniu ale przed następną fazą mamy nowe dane, choć w złym pliku (należy je później odzyskać kończąc operacje),
- zmienia się nazwę pliku. Operacja jest atomowa. Po tej operacji transakcja została dokończona.

W RDBMS transakcja jest logiczną jednostką zadania składającą się z jednej lub wielu instrukcji SQL. Początkiem transakcji jest pierwsza wykonywalna instrukcja SQL (pierwsza która wystąpiła po zamknięciu poprzedniej transakcji). Transakcje są wykonywane za pomocą dwóch rozkazów: COMMIT i ROLLBACK. Pierwszy z tych rozkazów jest używany do zapisywania w bazie wszystkich dokonanych zmian, drugi do wycofywania zmian uprzednio wprowadzonych, pozostawia bazę danych w stanie przed początkiem transakcji. Przykładem transakcji jest dokonywanie przelewu pomiędzy jednym bankiem, a drugim. Operacja ta wymaga wykonania dwóch rozkazów UPDATE - zmniejszenie stanu konta w banku dokonującym przelew i zwiększenia odpowiedniego stanu konta w banku otrzymującym przelew. Z charakteru operacji przelewu wynika, że muszą być wykonane oba rozkazy albo żaden. Jeśli bowiem wykonano by tylko jeden z nich to pieniądze albo by "znikły" albo się "rozmnożyły".

Rozkaz COMMIT służy do zapisywania na stałe wykonywanych uprzednio operacji. Przed wykonanie tego rozkazu żadne zmiany w bazie danych nie są widoczne dla innych użytkowników. W podanym uprzednio przykładzie po wykonaniu dwóch rozkazów UPDATE, należy wykonać COMMIT w celu trwałego zapisania dokonanych zmian. Od tego momentu zmienione stany kont będą widoczne dla innych użytkowników.

Rozkaz ROLLBACK jest odwrotnością COMMIT. Jest to również rozkaz kończący transakcję, jednak powoduje wycofanie wszystkich zmian w bazie od poprzedniego rozkazu COMMIT lub ROLLBACK.
Zamknięcie transakcji następuje również w wyniku:

- wydania jednej z instrukcji związanych z definicją danych (CREATE, DROP, RENAME, ALTER) co powoduje zamknięcie (zastosowanie) bieżącej transakcji (o ile zawiera ona instrukcje modyfikujące dane),
- wykonanie instrukcji definicji danych jako nowej transakcji rozłączenia się klienta z serwerem bazy danych (transakcja zostaje zastosowana anomalnego zerwania sesji użytkownika) transakcja zostaje cofnięta.

Możliwe jest również dzielenie dłuższych transakcji na mniejsze części, poprzez deklarowanie znaczników pośrednich (savepoints). Umożliwia to zazna-

czenie i nazwanie pewnego punktu wewnątrz transakcji a co za tym idzie wycofanie w razie błędu jedynie fragmentu transakcji (tzn. do podanego punktu wewnątrz aktualnie wykonywanej transakcji) i ponowienie próby jego wykonania (np. ze zmienionymi parametrami) bez powtarzania całej transakcji od początku. Częściowe cofnięcie transakcji do znacznika nie zamyka transakcji. Wykonanie COMMIT lub ROLLBACK powoduje skasowanie wszystkich uprzednio zaznaczonych punktów.

Jest dobrym zwyczajem, by każdą transakcje kończyć rozkazem COMMIT lub ROLLBACK. Jeśli nie jest to zrobione, a kończy się skrypt lub blok, to system sam podejmuje decyzję, czy rozpoczętą transakcję wykonać czy wycofać. Może to w pewnych przypadkach prowadzić do rezultatów, które nie były oczekiwane przez osobę tworzącą skrypt.

Transakcje można częściowo zasymulować (w tabelach innych typów) poprzez blokowanie tabel biorących udział w zapytaniach, ale wówczas należy utworzyć własny mechanizm anulowania nieudanej próby wykonania takiej "pseudotransakcji".

1.2 Zmiany w API

Hasło na głównej stronie MySQL AB Company głosi: "MySQL - Najpopularniejsza na świecie opensource'owa baza danych". Nie ma w tych słowach przesady - MySQL-a znajdziemy bowiem w każdej dystrybucji Linuxa, a z witryny projektu możemy pobrać, oprócz kodu źródłowego, binarne wersje bazy dla kilkunastu systemów operacyjnych (m.in. Linux, Solaris, Windows i Mac OS X). Na kolejną wersję MySQL-a musieliśmy czekać dosyć długo. W tym czasie edycja 3.x stała się dobrze przetestowanym i wydajnym programem. Administratorom i programistom brakowało jednak niektórych opcji, znanych m.in. z komercyjnych baz danych - chociażby obsługi transakcji (dostępnych dotychczas tylko w wersji Max), czy zagnieżdżonych zapytań SQL-owych (subqueries).

Głównym celem projektantów MySQL-a było takie przepisanie większości kodu, aby baza była szybsza i łatwiejsza w rozbudowie. Oprócz optymalizacji kodu serwera pod kątem jego wydajności projektanci postarali się również o poprawę bezpieczeństwa i wygody obsługi bazy. Warto tu wymienić bezpośrednią obsługę połączeń SSL przez serwer bazy danych oraz rozbudowanie systemu uprawnień użytkowników przyznawanych za pomocą funkcji GRANT. Zwiększeniu funkcjonalności serwera służą m.in. takie modyfikacje, jak wspomniana wyżej obsługa transakcji w tabelach typu InnoDB (dostępnych standardowo w binarnych wersjach instalacyjnych pakietu), dodanie operatora UNION (łączenie zapytań), dynamicznych zmiennych serwera (polecenie SET), poprawienie mechanizmu wyszukiwania pełnotekstowego czy zastosowanie pamięci podręcznej zapytań. Programiści położyli też spory nacisk na poprawę zgodności bazy ze standardem ANSI SQL 92, obiecują również dą-

żenie do pełnej obsługi ANSI SQL 99. Podobnie jak poprzednia edycja, także MySQL 4.x doskonale porozumiewa się z klientami za pośrednictwem interfejsu ODBC 3.0, co jest cenną cechą, zwłaszcza w przypadku korzystania z serwera NT.

MySQL 4.x jest interesującą propozycją dla większości projektantów baz danych. Choć brak jej niektórych przydatnych właściwości (np. obsługi SQL-owej funkcji VIEW[1]), to dokonane poprawki (głównie obsługa transakcji w podstawowej wersji bazy) bardzo zbliżają ten serwer do produktów "pierwszoligowych" - zwłaszcza że program ma być w założeniach intensywnie rozwijany, nawet po powstaniu wersji finalnej. Jeśli programistom MySQL AB uda się przy tym zachować legendarną szybkość napędu bazy, zyska ona z pewnością wielu nowych zwolenników. Celowi temu służy biblioteka libmysqlclient.so, pozwalająca w prosty sposób włączać bazę danych do własnych projektów. Nie do pogardzenia są również takie cechy MySQL-a, jak doskonała integracja bazy z serwerem Apache i banalna wręcz instalacja pakietu. Opisywany produkt jest więc doskonałym potwierdzeniem tezy o zbawiennym wpływie "uwolnienia" kodu programu na jego jakość i popularność.

W poniższej tabeli zostały zgromadzone funkcje z API MySQL 3.x, wykorzystywane w Meta-HTML oraz ich odpowiedniki w API MySQL 4.x.

Funkcje w MySQL 3.x	Funkcje w MySQL 4.x
`mysql_connect()`	`mysql_real_connect()`
`mysql_drop_db()`	`mysql_query(DROP DATABASE)`
`mysql_create_db()`	`mysql_query(CREATE DATABASE)`

Dokładne opisy wyżej wymienionych funkcji znajdują się w Dodatku A, na stronie 21.

W Meta-HTML spośród trzech wyżej wymienionych funkcji używana jest tylko funkcja `mysql_connect()`. Zgodnie z sugestiami autorów MySQL należy tę funkcję zastąpić parą odwołań: `mysql_init()` i `mysql_real_connect()`. Zasadnicza różnica pomiędzy `mysql_connect()` a `mysql_real_connect()` polega na sposobie alokacji pamięci dla obiektu `MYSQL`, w którym przechowywane są parametry połączenia z bazą danych. O ile w wersji MySQL 3.x funkcja `mysql_connect()` jednocześnie alokuje pamięć dla obiektu `MYSQL`, inicjalizuje go oraz nawiązuje połączenie z bazą danych, to w wersji MySQL 4.0 alokowanie pamięci i inicjalizacja obiektu `MYSQL` wykonywana jest za pomocą `mysq_init`. Mówiąc krótko złożona funkcja `mysql_connect()` została rozbita na dwie prostsze funkcje, co zwiększa elastyczność API bazy MySQL. Raz zaalokowany obiekt możemy wykorzystać wielokrotnie do różnych połączeń bez potrzeby zwalniania i rezerwacji pomięci za każdym razem.

[1]Funkcja jest dostępna w aktualnej wersji MAX, natomiast w standardowej bazie MySQL planowane jest jej dodanie w wersji 5.0.

Rozdział 2

Zmiany w kodzie Meta-HTML

2.1 Zmiany w Meta-HTML wymuszone zmianami w MySQL

Zasadnicze zmiany API MySQL między wersją 3.x a 4.x polagały na usunięciu funkcji `mysql_connect()`, `mysql_drop_db()`, `mysql_create_db()` oraz dodaniu kilku nowych. Spośród trzech zaniechanych funkcji w Meta-HTML wykorzystywana jest tylko pierwsza z nich: `mysql_connect()`. Tak więc zmiany w Meta-HTML wymuszone zmianami API MySQL dotyczą tylko tej jednej funkcji. Funkcja ta odpowiada za inicjalizację połączenia z bazą danych.

Moim zadaniem było więc dostosowanie istniejącego interfejsu pomiędzy Meta-HTMLa bazą danych MySQL. Aktualizacja polega na użyciu funkcji `mysql_real_connect()` w miejsce `mysql_connect()`.

Praktycznie cały interfejs pomiędzy Meta-HTML a MySQL umieszczony jest w pliku `mysqlfuncs.c` i tam właśnie należało dokonać modyfikacji.

Za nawiązanie połączenia z bazą danych MySQL w Meta-HTML odpowiada funkcja `gsql_connect()`. Poniżej przedstawiam kod tej funkcji dostosowany do API MySQL 4.x.

```
static void
gsql_connect (char *dsn, Database *db)
{
  MYSQL *sock;
  char *dbhost = dsn_lookup ("host", dsn);
  char *dbname = dsn_lookup ("database", dsn);
  char *user = dsn_lookup ("uid", dsn);
  char *pass = dsn_lookup ("pwd", dsn);

  db->connected = 0;
  db->dbname = dbname;
  db->hostname = dbhost;

  if ((dbhost != (char *) NULL) && (dbname != (char *)NULL))
    {

#if MYSQL_VERSION_ID >= 32200
```

```
    if (!(sock = mysql_init(NULL)))
      {
        BPRINTF_BUFFER *e = bprintf_create_buffer ();
        bprintf (e, "Unable to initialize connection to MySQL ");
        bprintf (e, "(HOST: '%s'; DATABASE: '%s'; UID: '%s'; PWD: '%s')",
                 dbhost, dbname, user ? user : "[not supplied]",
                 pass ? "[supplied]" : "[not supplied]");
        gsql_save_error_message (db, e->buffer);
        bprintf_free_buffer (e);
      }

      sock = mysql_real_connect (sock, dbhost, user, pass, dbname, 0, NULL, 0);

      db->sock = sock;
      if (sock != NULL)
        {
          db->connected = 1;
        }
      else
        {
          BPRINTF_BUFFER *e = bprintf_create_buffer ();
          bprintf (e, "Unable to connect to server/database! ");
          bprintf (e, "(HOST: '%s'; DATABASE: '%s'; UID: '%s'; PWD: '%s')",
                   dbhost, dbname, user ? user : "[not supplied]",
                   pass ? "[supplied]" : "[not supplied]");
          gsql_save_error_message (db, e->buffer);
          bprintf_free_buffer (e);
        }
#else
      sock = mysql_connect (NULL, dbhost, user, pass);

      db->sock = sock;
      if (sock != NULL)
        {
          if (mysql_select_db (sock, dbname) == -1)
            {
              BPRINTF_BUFFER *e = bprintf_create_buffer ();
              bprintf (e, "Unable to connect to database: '%s' on '%s'",
                       dbname, dbhost);
              gsql_save_error_message (db, e->buffer);
              bprintf_free_buffer (e);
            }
          else
            db->connected = 1;
        }
      else
        {
          BPRINTF_BUFFER *e = bprintf_create_buffer ();
          bprintf (e, "Unable to connect to server! ");
          bprintf (e, "(HOST: '%s'; DATABASE: '%s'; UID: '%s'; PWD: '%s')",
                   dbhost, dbname, user ? user : "[not supplied]",
                   pass ? "[supplied]" : "[not supplied]");
          gsql_save_error_message (db, e->buffer);
          bprintf_free_buffer (e);
        }
#endif
    }
  else
    {
      BPRINTF_BUFFER *e = bprintf_create_buffer ();
      bprintf (e, "Unable to connect to server! ");
      bprintf (e, "(HOST: '%s'; DATABASE: '%s'; UID: '%s'; PWD: '%s')",
               dbhost ? dbhost : "[not supplied]",
               dbname ? dbname : "[not supplied]",
               user ? user : "[not supplied]",
               pass ? "[supplied]" : "[not supplied]");
```

```
            gsql_save_error_message (db, e->buffer);
            bprintf_free_buffer (e);
        }

    xfree (user);
    xfree (pass);
}
```

Zasadnicza zmiana polega na zawołaniu nowych funcji `mysql_init()` i `mysql_real_connect()` zamiast `mysql_connect()`. Zastosowaliśmy tutaj kompilację warunkową wykożystując makropolecenia preprocesora: `if/els/endif`. Wykorzystujemy fakt, że w pliku `mysql.h` zdefiniowana jest stała `MYSQL_VERSION_ID` określająca wersję zainstalowanej bazy danych MySQL. Wartość tej stałej (np. 32200) należy rozumieć jako konkatenację numeru głównego wersji (w tym wypadku 3), numeru subwersji (tutaj 22) oraz subsubwersji (tutaj 00). Funkcje `mysql_init()` i `mysql_real_connect()` zostały wprowadzone w wersji 3.22, dlatego też `MYSQL_VERSION_ID` porównujemy z wartością 32200 i o ile numer wersji jest wyższy używamy nowych funkcji, w przeciwnym razie starych.

W tym samym pliku dokonaliśmy jeszcze jedną zmianę związaną z funkcją `mysql_real_connect()`. Podobnie jak wcześniej zastosowaliśmy kompilację warynkową ze sprawdzeniem wersji bazy MySQL. Poniżej prezentujemy kompletną funkcję `pf_host_databases()` z naniesionymi poprawkami.

```
static void
pf_host_databases (PFunArgs)
{
  char *host = mhtml_evaluate_string (get_positional_arg (vars, 0));
  char *resultvar = mhtml_evaluate_string (get_value (vars, "result"));
  MYSQL *sock;

  /* No errors yet! */
  pagefunc_set_variable ("mysql::mysql-error-message[]", "");

  if (empty_string_p (host))
    {
      xfree (host);
      host = strdup ("localhost");
    }

#if MYSQL_VERSION_ID >= 32200
  if ((sock = mysql_init(NULL)) != NULL &&
          (sock = mysql_real_connect (NULL, host, NULL, NULL, NULL, 0, NULL, 0)) != NULL)
#else
  if ((sock = mysql_connect (NULL, host, NULL, NULL)) != NULL)
#endif
    {
      MYSQL_RES *result = mysql_list_dbs (sock, NULL);
      int nrows = (result ? mysql_num_rows (result) : 0);

      if (nrows != 0)
        {
          int count = 0;
          char **dbnames = (char **) xmalloc ((nrows + 1) * sizeof (char *));
          MYSQL_ROW mysql_row;

          /* Loop over rows returned; the db name will be passed in the first
             field of each 'row'.  Add names to the result array.  */
```

```
          while ((mysql_row = mysql_fetch_row (result)) != (MYSQL_ROW)NULL)
            dbnames[count++] = strdup (mysql_row[0]);

          dbnames[count] = (char *) NULL;

          if (!empty_string_p (resultvar))
            {
              symbol_store_array (resultvar, dbnames);
            }
          else
            {
              register int i;

              for (i = 0; dbnames[i] != (char *)NULL; i++)
                {
                  bprintf_insert (page, start, "%s\n", dbnames[i]);
                  start += 1 + strlen (dbnames[i]);
                  free (dbnames[i]);
                }
              free (dbnames);
              *newstart = start;
            }
        }

      if (result != (MYSQL_RES *)NULL) mysql_free_result (result);
      mysql_close (sock);
    }
  else
    {
      gsql_save_error_message
        ((Database *)NULL, "HOST-DATABASES: mySQL Connect Failed");
    }

  xfree (host);
  xfree (resultvar);
}
```

Funkcja `pf_host_databases()` wykorzystywana jest w makrze `SQL::HOST-DATABASES`, które zwraca listę wszystkich baz danych założonych w MySQL na wskazanym komputerze.

2.2 Implementacja transakcji w Meta-HTML

Używanie transakcji w MySQL wiąże się z trzema poleceniami SQL: START TRANSACTION (BEGIN w nieco starszych wersjach MySQL), COMMIT i ROLLBACK. Polecenie START TRANSACTION (jak również BEGIN) nie posiada w API swego odpowiednika, wywoływane jest ono za pomocą ogólnej funkcji `mysql_query()` (patrz str. 22). Począwszy od wersji 4.1 polecenia COMMIT i ROLLBACK odpowiadają funkcjom `mysql_commit()` i `mysql_rollback()` (patrz str. 21 i 24). W poprzednich wersjach bazy używa się ogólnej funkcji `mysql_query()`.

Ze względu na różnice w implementacji poleceń związanych z transakcjami w API MySQL oraz dla lepszej czytelności kodu źródłowego napisaliśmy trzy osobne funkcje odpowiadające każdemu z poleceń, tworząc w ten sposób elastyczny interfejs pomiędzy API MySQL a implementacją transakcji w

Meta-HTMLPoniżej przedstawione są te właśnie trzy funkcje.

```
static int
gsql_start_transact_internal (Database *db)
{

#if MYSQL_VERSION_ID < 40000
  return (GSQL_ERROR);
#endif

  return gsql_query(db,
#if MYSQL_VERSION_ID >= 40011
                    "START TRANSACTION",
#else
                    "BEGIN",
#endif
                        1);
}


static int
gsql_commit_internal (Database *db)
{
  int result = GSQL_ERROR;

#if MYSQL_VERSION_ID < 40000
  return (GSQL_ERROR);
#endif

#if MYSQL_VERSION_ID >= 41000
  if (mysql_commit(db) == 0)
    result = GSQL_SUCCESS;
  else
    result = GSQL_ERROR;
#else
  result = gsql_query(db, "COMMIT", 1);
#endif

  return (result);
}


static int
gsql_rollback_internal (Database *db)
{
  int result = GSQL_ERROR;

#if MYSQL_VERSION_ID < 40000
  return (GSQL_ERROR);
#endif

#if MYSQL_VERSION_ID >= 41000
  if (mysql_rollback(db) == 0)
    result = GSQL_SUCCESS;
  else
    result = GSQL_ERROR;
#else
  result = gsql_query(db, "ROLLBACK", 1);
#endif

  return (result);
}
```

Przewidziana przez autorów Meta-HTML implementacja transakcji zakłada, że jest jedno makro `SQL::SQL-TRANSACT`, które realizuje wszystkie pole-

cenia START, COMMIT i ROLLBACK na podstawie pobranego parametru. Częściowo to makro zostało zaimlementowane dla baz danych korzystających z interfejsu ODBC. Aby dostosować się do tej implementacji musieliśmy dołączyć do tworzonego przez nas interfejsu odpowiednią funkcję, którą widać poniżej.

```
static int
gsql_transact_internal (Database *db, char *action_arg)
{
  int result = GSQL_ERROR;

  if (!strcasecmp(action_arg, "START"))
    result = gsql_start_transact_internal(db);
  else if (!strcasecmp(action_arg, "COMMIT"))
    result = gsql_commit_internal(db);
  else if (!strcasecmp(action_arg, "ROLLBACK"))
    result = gsql_rollback_internal(db);

  return (result);
}
```

Funkcja ta zapewnia kompatybilność z istniejącym interfejsem. Możliwe wartości argumentu `action_arg` to: START, COMMIT, ROLLBACK.

W przypadku kompilacji Meta-HTML z wersją bazy MySQL starszą niż 4.0 powyższe funkcje zwracają błąd `GSQL_ERROR`.

Aby móc korzystać z transakcji w Meta-HTML musimy zaimplementować odpowiednie makra. Przedstawiamy je poniżej:

```
<defun SQL::SQL-TRANSACT dbvar action>
  ;;; DOC_SECTION (GENERIC-SQL-INTERFACE)
  ;;;
  ;;; Performs specified in <var action> transactions related operation on an
  ;;; SQL database referenced by <var dbvar>.
  ;;; The <var action> can be one of the following: START,
  ;;; COMMIT or ROLLBACK. If unspecified, <var action> defaults to COMMIT.
  ;;; Returns <i>true</i> if operation succeeds, empty string otherwise.
</defun>

<defun SQL::SQL-START-TRANSACT dbvar>
  ;;; DOC_SECTION (GENERIC-SQL-INTERFACE)
  ;;;
  ;;; Begins SQL transaction on an SQL database referenced by <var dbvar>.
  ;;; Returns <i>true</i> if operation succeeds, empty string otherwise.
</defun>

<defun SQL::SQL-COMMIT dbvar>
  ;;; DOC_SECTION (GENERIC-SQL-INTERFACE)
  ;;;
  ;;; Perform COMMIT on an SQL database referenced by <var dbvar>.
  ;;; Returns <i>true</i> if operation succeeds, empty string otherwise.
</defun>

<defun SQL::SQL-ROLLBACK dbvar>
  ;;; DOC_SECTION (GENERIC-SQL-INTERFACE)
  ;;;
  ;;; Perform ROLLBACK on an SQL database referenced by <var dbvar>.
  ;;; Returns <i>true</i> if operation succeeds, empty string otherwise.
</defun>
```

Makro `SQL::SQL-TRANSACT` było obecne w kodzie Meta-HTML, jednak dla bazy MySQL zwracało błąd gdyż nie było do końca zaimplementowane. Po-

zostałe trzy makra zostały przez nas dodane. Aby makra te działały w Meta-HTML musimy je zarejestrować i dopisać obsługę zgodnie z przewidzianą w Meta-HTML składnią. Rejestracja odbywa się przez powiązanie identyfikatora makra z konkretną funkcją która je realizuje. Tabela powiązań znajduje się w tablicy `func_table`. Po uzupełnieniu ma ona postać:

```
static PFunDesc func_table[] =
{
  { "MYSQL::WITH-OPEN-DATABASE",          1, 0, pf_with_open_database },
  { "MYSQL::DATABASE-EXEC-QUERY",         0, 0, pf_database_exec_query },
  { "MYSQL::DATABASE-EXEC-SQL",           0, 0, pf_database_exec_sql },
  { "MYSQL::DATABASE-NEXT-RECORD",        0, 0, pf_database_next_record },
  { "MYSQL::DATABASE-SAVE-RECORD",        0, 0, pf_database_save_record },
  { "MYSQL::PACKAGE-TO-TABLE",            0, 0, pf_package_to_table },
  { "MYSQL::DATABASE-DELETE-RECORD",      0, 0, pf_database_delete_record },
  { "MYSQL::DATABASE-LOAD-RECORD",        0, 0, pf_database_load_record },
  { "MYSQL::DATABASE-SAVE-PACKAGE",       0, 0, pf_database_save_package },
  { "MYSQL::NUMBER-OF-ROWS",              0, 0, pf_database_num_rows},
  { "MYSQL::AFFECTED-ROWS",               0, 0, pf_database_affected_rows},
  { "MYSQL::SET-ROW-POSITION",            0, 0, pf_database_set_pos},
  { "MYSQL::DATABASE-QUERY",              0, 0, pf_database_query },
  { "MYSQL::HOST-DATABASES",              0, 0, pf_host_databases },
  { "MYSQL::DATABASE-TABLES",             0, 0, pf_database_tables },
  { "MYSQL::DATABASE-TABLES-INFO",        0, 0, pf_database_tables_info },
  { "MYSQL::DATABASE-COLUMNS",            0, 0, pf_database_columns },
  { "MYSQL::DATABASE-COLUMN-INFO",        0, 0, pf_database_column_info },
  { "MYSQL::DATABASE-COLUMNS-INFO",       0, 0, pf_database_columns_info },
  { "MYSQL::DATABASE-QUERY-INFO",         0, 0, pf_database_query_info },
  { "MYSQL::DATABASE-SET-OPTIONS",        0, 0, pf_database_set_options },
  { "MYSQL::CURSOR-GET-COLUMN",           0, 0, pf_cursor_get_column },
  { "MYSQL::QUERY-GET-COLUMN",            0, 0, pf_query_get_column },
  { "MYSQL::SQL-TRANSACT",                0, 0, pf_sql_transact },
  { "MYSQL::SQL-START-TRANSACT",          0, 0, pf_sql_start_transact },
  { "MYSQL::SQL-COMMIT",                  0, 0, pf_sql_commit },
  { "MYSQL::SQL-ROLLBACK",                0, 0, pf_sql_rollback },
  { (char *)NULL,                         0, 0, (PFunHandler *)NULL }
};
```

Prefiks `MYSQL` w identyfikatorze makra zastępowany jest ogólnym prefiksem `SQL` jeśli używaną bazą danych jest MySQL. Dalej prezentujemy związane z powyższymi makrami funkcje realizujące te makra.

```
/* <sql-transact db [action=COMMIT|ROLLBACK]>

   Perform transaction, to either commit or rollback all operations on
   the current database connection.

   ACTION can be one of COMMIT or ROLLBACK.
   If unspecified, ACTION defaults to COMMIT.  */
static void
pf_sql_transact (PFunArgs)
{
  char *action_arg = mhtml_evaluate_string (get_value (vars, "ACTION"));

  /* No errors yet! */
  gsql_clear_error_message ();

  if (database_environment_level != 0)
    {
      Database *db = get_dbref (vars);
      int status;
```

```
      if ((db != (Database *)NULL) && gsql_database_connected (db))
        {
          char *result;
          status = gsql_transact_internal (db, action_arg);

          if (status == GSQL_SUCCESS)
            {
              result = "true";
              bprintf_insert (page, start, "%s", result);
              *newstart += strlen (result);
            }
          else
            gsql_save_error_message (db, GSQL_DEFAULT_ERRMSG);
        }
    }
  xfree (action_arg);
}


/* <sql-start-transact db>

   Perform start transaction on the current database connection.
*/
static void
pf_sql_start_transact (PFunArgs)
{
  /* No errors yet! */
  gsql_clear_error_message ();

  if (database_environment_level != 0)
    {
      Database *db = get_dbref (vars);
      int status;

      if ((db != (Database *)NULL) && gsql_database_connected (db))
        {
          char *result;
          status = gsql_start_transact_internal (db);

          if (status == GSQL_SUCCESS)
            {
              result = "true";
              bprintf_insert (page, start, "%s", result);
              *newstart += strlen (result);
            }
          else
            gsql_save_error_message (db, GSQL_DEFAULT_ERRMSG);
        }
    }
}


/* <sql-commit db>

   Perform commit on the current database connection.
*/
static void
pf_sql_commit (PFunArgs)
{
  /* No errors yet! */
  gsql_clear_error_message ();

  if (database_environment_level != 0)
    {
      Database *db = get_dbref (vars);
      int status;
```

```
      if ((db != (Database *)NULL) && gsql_database_connected (db))
        {
          char *result;
          status = gsql_commit_internal (db);

          if (status == GSQL_SUCCESS)
            {
              result = "true";
              bprintf_insert (page, start, "%s", result);
              *newstart += strlen (result);
            }
          else
            gsql_save_error_message (db, GSQL_DEFAULT_ERRMSG);
        }
    }
}


/* <sql-rollback db>

   Perform rollback on the current database connection.
*/
static void
pf_sql_rollback (PFunArgs)
{
  /* No errors yet! */
  gsql_clear_error_message ();

  if (database_environment_level != 0)
    {
      Database *db = get_dbref (vars);
      int status;

      if ((db != (Database *)NULL) && gsql_database_connected (db))
        {
          char *result;
          status = gsql_rollback_internal (db);

          if (status == GSQL_SUCCESS)
            {
              result = "true";
              bprintf_insert (page, start, "%s", result);
              *newstart += strlen (result);
            }
          else
            gsql_save_error_message (db, GSQL_DEFAULT_ERRMSG);
        }
    }
}
```

Jak widać implementacja powyższych funkcji jest bardzo podobna. Każda z nich na początku czyści bufor błędów przy pomocy `gsql_clear_error_message()`, następnie, o ile otwarte jest połączenie z bazą danych wykonywane jest stosowne polecenie SQL. Jeśli nie wystąpił żaden błąd zwracany jest ciąg znaków `true`, w przeciwnym razie zwracany jest pusty ciąg znaków. Jest to realizacja zgodna ze standardem Meta-HTML.

Rozdział 3

Uruchamianie zmodyfikowanego Meta-HTML

Uruchamianie i testowanie zmodyfikowanego Meta-HTML wykonywaliśmy w labolatorium komputerowym instytutu pod systemem operacyjnym Solaris. Zaczeliśmy od kompilowania kodu źródłowego z naniesionymi zmianami i poprawienia błędów na tym poziomie. Następnym krokiem było wstawienie zmienionych programów wykonywalnych i bibliotek w odpowiednie miejsce w systemie. Wstępne testy wykazały, że przy definiowaniu makr w bibliotece Meta-HTML zapomnieliśmy o niezbędnym parametrze jakim jest uchwyt do bazy danych. Należało odpowiednio poprawić definicję i zrekompilować moduł Meta-HTML.

Aby w pełni przetestować nowo dodane makra w Meta-HTML opracowaliśmy bardzo uproszczony sklep internetowy wykorzystujący bazę MySQL i Meta-HTML. W tym celu w bazie MySQL zostały utworzone tabele typu InnoDB obsługujące transakcje, których zawartość przedstawiamy poniżej. Pomysł został zaczerpnięty z [5].
Tabela `produkty` zawiera spis dostępnych towarów wraz z opisem i ceną, które można kupić w naszym sklepie.

```
mysql> select * from produkty;
+------------+----------------+----------------+---------------+
| produkt_id | nazwa_produktu | opis_produktu  | cena_produktu |
+------------+----------------+----------------+---------------+
|          1 | Komputer       | Super Komputer |       8999.99 |
|          2 | Skaner         | Dobry Skaner   |        399.99 |
+------------+----------------+----------------+---------------+
```

Tabela `klienci` zawiera dane osobowe klientów naszego sklepu.

```
mysql> select * from klienci;
+-------------+--------+----------+----------+------+
| kupujacy_id | imie   | nazwisko | nr_tel   | fax  |
+-------------+--------+----------+----------+------+
|           1 | Jan    | Kowalski | 7654657  | NULL |
|           2 | Janina | Kowalska | 7654656  | NULL |
+-------------+--------+----------+----------+------+
```

Tabela `stan_magazynu` zawiera ilość każdego z produktów.

```
mysql> select * from stan_magazynu;
+------------+-------+
| produkt_id | ilosc |
+------------+-------+
|          1 |     3 |
|          2 |    18 |
+------------+-------+
```

Tabela `zamowienia` wiąże kupującego z tabeli `klienci` z konkretnym zamówieniem, to znaczy dla zamówienia nadawany jest unikalny numer. Ponadto podana jest całkowita wartość zamówienia.

```
mysql> select * from zamowienia;
+---------------+-------------+-----------+
| id_zamowienia | kupujacy_id | cala_cena |
+---------------+-------------+-----------+
|             1 |           1 |   9399.98 |
|             2 |           2 |   8999.99 |
+---------------+-------------+-----------+
```

Tabela `zamowienia_produkty` zawiera spis tego co znajdowało się w koszykach kupujących w momenci składania zamówienia. Innymi słowy z tej tabeli uzyskujemy listę zamówionych towarów przez klienta.

```
mysql> select * from zamowienia_produkty;
+----+---------------+------------+---------+-------+
| id | id_zamowienia | produkt_id | cena    | ilosc |
+----+---------------+------------+---------+-------+
|  1 |             1 |          1 | 8999.99 |     1 |
|  2 |             1 |          2 |  399.99 |     1 |
|  3 |             2 |          1 | 8999.99 |     1 |
+----+---------------+------------+---------+-------+
```