**INSTYTUT INŻYNIERII I GOSPODARKI WODNEJ**

**POLITECHNIKA KRAKOWSKA im. TADEUSZA KOŚCIUSZKI**

Damian Dziechciarz

# SQLITE JAKO ALTERNATYWA DLA SERWEROWYCH SYSTEMÓW BAZODANOWYCH

praca magisterska

studia dzienne

kierunek studiów: **informatyka**

specjalność: **informatyka stosowana w inżynierii środowiska**

promotor: **dr inż. Robert Szczepanek**

nr pracy: **2180**

KRAKÓW 2008

ul. Warszawska 24, 31-155 Kraków tel/fax (+48 12) 628 20 41 e-mail: sekretariat@iigw.pl internet: www.iigw.pl

Składam serdeczne podziękowania
**Panu dr inż. Robertowi Szczepankowi**
za wskazówki i pomoc udzieloną przy
pisaniu niniejszej pracy.

# Spis treści

# 1.

# Wprowadzenie

Współczesny świat to świat informacji, dlatego w dzisiejszych czasach zachodzi konieczność gromadzenia i przetwarzania dużych ilości danych. Powoduje to szybki rozwój baz danych, które umożliwiają odpowiednie przechowywanie danych oraz sprawny dostęp do nich. Obecnie bazy danych stanowią ważny element złożonych systemów informatycznych. Korzystają z nich wszelkiego rodzaju aplikacje, portale i sklepy internetowe oraz potężne informatyczne systemy przemysłowe i usługowe.

Na rynku dostępnych jest wiele systemów do zarządzania bazami danych zarówno komercyjnych jak np. Oracle oraz darmowych m.in. MySQL i PostgreSQL. Wszystkie one są niezwykle rozbudowanymi aplikacjami opartymi na architekturze typu klient-serwer. Kilka lat temu pojawiła się ciekawa alternatywa dla tych systemów, mianowicie biblioteka SQLite, która implementuje w pełni funkcjonalny system do obsługi baz danych.

Celem niniejszej pracy jest przedstawienie możliwości SQLite'a na tle innych otwartych systemów bazodanowych. Prezentacja SQLite'a sprowadza się do pokazania jego mocnych jak i słabych stron, udostępnianych funkcji oraz porównania jego wydajności z innymi systemami baz danych.

Praca ta składa się z pięciu głównych rozdziałów. W rozdziale pierwszym przybliżone zostały zagadnienia związane z relacyjnymi bazami danych takie jak relacyjny model danych, normalizacja baz czy język SQL. Rozdział drugi stanowi szczegółowy opis strony funkcjonalnej bazy SQLite. Opisany został tam język zapytań do bazy, architektura, typy danych oraz optymalizacja. Trzeci rozdział to przedstawienie interfejsu API dostępu do bazy na przykładzie języka PHP. Z rozdziałem tym związany jest rozdział kolejny, w którym porównana została wydajność SQLite'a z innymi darmowymi bazami przy użyciu skryptów napisanych w języku PHP. Ostatni rozdział obejmuje prezentację rozszerzenia SQLite'a umożliwiającego składowanie danych przestrzennych.

Na końcu pracy przedstawiono wnioski oraz krótkie podsumowanie.

# 2.

# Podstawy systemów bazodanowych

## 2.1. Język SQL

Język SQL jest najpopularniejszym relacyjnym językiem zapytań bazodanowych. Nazwa jest skrótem od Structured Query Language, czyli Strukturalny Język Zapytań.

Został on opracowany w latach siedemdziesiątych ubiegłego wieku przez firmę IBM. Prototyp języka SQL noszący nazwę SEQUEL (*ang. Structured English Query Language*) został zaprezentowany w 1974 roku. W latach 1976-77 powstała poprawiona wersja języka nazwana SEQUEL-2, a następnie zmieniono nazwę na SQL. W 1986 roku Amerykański Instytut Standardów ANSI dokonał pierwszej standaryzacji języka SQL, a rok później został on zaakceptowany przez Międzynarodową Organizację Standaryzacji ISO. Ta wersja standardu była nieoficjalnie nazywana SQL-86. W 1989 roku standard został rozszerzony i nosił nazwę SQL-89. Wersja standardu obowiązująca do dziś powstała w 1992 roku. Została ona przyjęta jako: „International Standard ISO/IEC 9075:1992, Database Language SQL" i określana jest mianem standardu SQL-92. [Dubois, 2005]

## 2.2. Relacyjny model danych

Relacyjny model danych został wprowadzony po raz pierwszy w 1970 roku przez Edgara Franka Codda. Opiera się on na pojęciu matematycznej relacji.

Model relacyjny składa się z trzech podstawowych elementów:

- struktury
- integralności
- manipulacji

[Wikipedia, 2008-10-12]

W relacyjnym modelu danych istnieje tylko jedna struktura danych, nazywana relacją. Relacja prezentowana jest w formie tabeli. Musi ona zawierać nagłówek oraz zawartość. Nagłówek relacji to zbiór atrybutów reprezentowanych przez kolumny, zawierających nazwę oraz typ. Zawartość natomiast jest zbiorem krotek

reprezentowanych w postaci wierszy (krotki ściślej określane są jako zbiór wartości atrybutów).

Integralność to ograniczenia nakładane na bazę przez model relacyjny. Wyróżniamy integralność encji oraz integralność odwołań. Więzy integralności encji zakazują stosowania wartości pustych dla atrybutów tworzących klucz główny relacji. Więzy integralności odwołań definiowane są pomiędzy parami relacji i wykorzystywane są do utrzymania spójności pomiędzy krotkami należącymi do różnych relacji (inaczej mówiąc krotka jednej relacji, która odwołuje się do innej relacji musi zawsze wskazywać na istniejącą krotkę tamtej relacji).

Elementy manipulacyjne to zbiór operatorów relacyjnych zwanych algebrą relacyjną.

Algebra relacyjna stanowi podstawowy zbiór operacji dla modelu relacyjnego. Wynikiem tych operacji jest nowa relacja powstała z jednej lub dwóch relacji już istniejących.

Podstawowymi operatorami algebry relacyjnej są:

- selekcja (ograniczenie)
- rzut (projekcja)
- złączenie

Selekcja wykorzystywana jest do wyznaczenia podzbioru wierszy z relacji, który spełnia podany warunek selekcji. Selekcja pobiera jako argument jedną relację i zwraca w wyniku jedną relacje. Warunek selekcji może stanowić jeden parametr (warunek prosty) lub kilku warunków prostych połączonych koniunkcją lub alternatywą (warunek złożony). Selekcja nazywana jest „poziomą maszyną do cięcia”, ponieważ w wyniku działania ogranicza liczbę wierszy.

Rzut pobiera jako argument jedną relację i zwraca w wyniku również jedną relację, ale z ograniczoną liczbą kolumn. Rzut nazywany jest „pionową maszyną do cięcia”, ponieważ w wyniku działania ogranicza liczbę kolumn.

Złączenie pobiera jako argument dwie relacje i tworzy jedną relacje wynikową złożoną z wybranych krotek z obu relacji. Złączenie może być lewostronne, prawostronne lub obustronne.

Oddzielną grupę operacji stanowią operacje z matematycznej teorii zbiorów. Są to:

- suma – w wyniku której otrzymujemy relację zawierającą wiersze z obu relacji
- część wspólna – w wyniku której otrzymujemy relację zawierającą wiersze wspólne obu relacji
- różnica – w wyniku której otrzymujemy relację zawierającą wiersze należące do pierwszej relacji i nie należące do drugiej
- iloczyn kartezjański – który tworzy relację zawierającą wszystkie możliwe kombinacje wierszy w wejściowych relacji

## 2.3. Normalizacja baz danych

Normalizacja baz danych to proces mający na celu zapewnienie jak największej zwartości relacyjnej bazy danych. Ma on na celu wyeliminowanie takich zjawisk jak nadmiarowość i niespójne zależności.

Nadmiarowość występuje wtedy, gdy w bazie wielokrotnie powtarzają się te same informacje. Powtarzające się dane niepotrzebnie zajmują miejsce na dysku i są przyczyną powstawania problemów z obsługą zapytań.

Niespójna zależność oznacza brak możliwości dotarcia do określonej informacji i spowodowana jest błędnym zaplanowaniem relacji w strukturze bazy danych.

Proces normalizacji polega na sprowadzeniu struktury bazy do tzw. postaci normalnych. Obecnie wyróżnić można pięć poziomów postaci normalnych, jednak trzecia postać normalna uważana jest za najwyższy poziom wymagany przez większość aplikacji. [Kent, 1983]

**Pierwsza postać normalna**

Pierwsza postać normalna wymaga, żeby wszystkie pola w tabelach były atrybutami elementarnymi tzn. niedającymi podzielić się na mniejsze części. Aby sprowadzić bazę do pierwszej postaci normalnej należy:

- w poszczególnych tabelach wyeliminować powtarzające się grupy
- dla każdego zestawu danych pokrewnych utworzyć oddzielną tabelę
- dla każdego zestawu danych pokrewnych określić klucz podstawowy

**Druga postać normalna**

Druga postać normalna wymaga, żeby wszystkie atrybuty były w pełni funkcjonalnie zależne od kluczy. W celu sprowadzenia bazy do drugiej postaci normalnej należy:

- utworzyć oddzielne tabele dla zestawów wartości, odnoszących się do wielu rekordów
- ustalić powiązania tabel za pomocą klucza obcego

**Trzecia postać normalna**

Trzecia postać normalna wymaga, aby każdy atrybut wtórny był tylko bezpośrednio zależny od klucza.. W celu sprowadzenia bazy do trzeciej postaci normalnej należy:

- wyeliminować pola, które nie zależą od klucza

Jeśli zawartość grupy pól odnosi się do więcej niż jednego rekordu tabeli, należy rozważyć umieszczenie tych pól w oddzielnej tabeli.

## 2.4. Popularne otwarte systemy bazodanowe

### 2.4.1. MySQL

MySQL jest najpopularniejszym otwartym systemem bazodanowym. Rozwijany jest od 1995 roku przez szwedzką firmę MySQL AB (w 2008 roku kupioną przez Sun Microsystems). Udostępniany jest na licencji GNU GPL oraz licencji komercyjnej (dla aplikacji komercyjnych). Pisany przede wszystkim z myślą o szybkości działania. Oparty jest na architekturze klient-serwer.

Cechy systemu MySQL 5.0:

- Transakcje ACID - TAK (tylko dla tabel InnoDB)
- Wyzwalacze – TAK, kursory – TAK, procedury składowane – TAK
- Języki proceduralne – TAK (zdefiniowane przez ANSI SQL 2003)

- Różne typy tabel – TAK (opisane poniżej m.in. MyISAM, InnoDB)
- Indeksy R-tree dla danych przestrzennych – TAK (tylko dla tabel MyISAM)
- Maksymalna wielkość tabeli – od 2 GB (Win32) do 16 TB (Solaris)
- Maksymalny rozmiar rekordu – 64 KB
- Maksymalny rozmiar przechowywanego obiektu binarnego (BLOB) – 4 GB
- Wspierane systemy operacyjne: Windows, Linux, Mac OS, BSD, Solaris, Symbian

MySQL oferuje różne typy tabel (mechanizmy składowania). Są to m.in.:

- MyISAM – domyślny typ tabel, nie obsługuję transakcji, bardzo szybki, obsługuje wyszukiwanie pełnotekstowe
- InnoDB – typ tabel obsługujący transakcje oraz poziomy izolacji, co powoduje, że jest bardziej niezawodny od MyISAM, ale przez to wolniejszy
- MEMORY – najszybszy typ tabel, przechowujący dane w pamięci operacyjnej, a nie na dysku, po wyłączeniu komputera dane są tracone
- MERGE – tabela tego typu jest połączeniem kilku tabel MyISAM o identycznej strukturze
- FEDERATED – umożliwia tworzenie rozproszonych baz danych
- CSV – przechowuje dane w standardowych plikach CSV, nie posiada możliwości indeksowania danych
- ARCHIVE – typ służący do przechowywania dużych ilości danych bez indeksów, dane są skompresowane przy użyciu algorytmu zliż

### 2.4.2. PostgreSQL

PostgreSQL to obok MySQL-a jeden z najpopularniejszych otwartych systemów zarządzania relacyjnymi bazami danych. Jest najbardziej zaawansowanym systemem bazodanowym rozpowszechnianym na wolnej licencji. Często porównywany jest z komercyjnym systemem Oracle. Opracowany został na Uniwersytecie Kalifornijskim w Berkeley i opublikowany pod nazwą Postgres. W miarę rozwoju

i zwiększania funkcjonalności baza otrzymała nazwy Postgres95 i ostatecznie PostgreSQL. Obecnie rozwijana jest przez organizację PostrgreSQL Global Development Group. Udostępniany jest na licencji BSD. Jest systemem opartym na architekturze klient-serwer.

Cechy systemu PostgreSQL:

- Transakcje ACID - TAK
- Wyzwalacze – TAK, kursory – TAK, procedury składowane – TAK
- Języki proceduralne – TAK (PL/pgSQL, PL/Tcl, PL/perl)
- Różne typy tabel – NIE (posiada jeden wbudowany silnik tzw. Postgres Storage System)
- Indeksy R-tree dla danych przestrzennych – TAK
- Maksymalna wielkość tabeli – 32 TB
- Maksymalny rozmiar rekordu – 1.6 TB
- Maksymalny rozmiar przechowywanego obiektu binarnego (BLOB) – 1 GB
- Wspierane systemy operacyjne: Windows, Linux, Mac OS, BSD, Solaris

### 2.4.3. Firebird

Firebird to trzeci najpopularniejszy otwarty system bazodanowy. Rozwijany jest od 2000 roku na bazie kodu źródłowego serwera Interbase 6.0 udostępnionego przez firmę Inprise Corp. Rozpowszechniany jest na licencji InterBase Public License. Firebird zgodny jest ze standardem SQL-92.

Cechy systemu Firebird:

- Transakcje ACID - TAK
- Wyzwalacze – TAK, kursory – TAK, procedury składowane – TAK
- Języki proceduralne – TAK (PSQL)
- Różne typy tabel - NIE
- Indeksy R-tree dla danych przestrzennych – NIE

- Maksymalna wielkość tabeli – 32 TB
- Maksymalny rozmiar rekordu – 65 536 B
- Maksymalny rozmiar przechowywanego obiektu binarnego (BLOB) – 2 GB
- Wspierane systemy operacyjne: Windows, Linux, Mac OS, BSD, Solaris

# 3.

# SQLite – opis systemu

## 3.1. Czym jest SQLite?

SQLite jest biblioteką, która implementuje niezależny, bezserwerowy i bezkonfiguracyjny silnik bazodanowym. Pierwsze wydanie tej biblioteki miało miejsce w 2000 roku. Twórcą SQLite'a jest D. Richard Hipp. Obecnie projekt rozwijany jest przez firmę Hwaci (Hipp, Wyrick & Company, Inc.), której założycielem jest Hipp.

Kod SQLite'a rozpowszechniany jest na publicznej licencji i baza może być za darmo używana w każdym celu zarówno prywatnym jak i komercyjnym. SQLite można obecnie znaleźć w wielu aplikacjach, włączając w to projekty o dużym znaczeniu.

SQLite jest wbudowanym silnikiem bazodanowym. W przeciwieństwie do większości SQL-owych baz danych, SQLite nie korzysta z oddzielnego procesu serwera pracującego w tle.

SQLite odczytuje i zapisuje dane bezpośrednio do zwykłego pliku. Kompletna SQL-owa baza danych łącznie ze złożonymi tabelami, indeksami, wyzwalaczami i widokami, które obsługuje, przechowywana jest w pojedynczym pliku. Plik, w którym zapisana jest baza może działać na różnych platformach, można go swobodnie kopiować pomiędzy systemami 32 i 64-bitowymi. Te cechy sprawiają, że SQLite jest ciekawym wyborem jako format plikowy dla aplikacji.

SQLite jest spójną biblioteką, która łącznie ze wszystkimi możliwymi właściwościami zajmuje nie więcej niż 250 KB (w zależności od ustawień kompilacji), a jeżeli zostaną pominięte opcjonalne cechy to wielkości biblioteki maleje do 180 KB. Wymaga także niewielkiej ilości pamięci operacyjnej. Powoduje to, że SQLite jest popularnym wyborem dla urządzeń z ograniczeniami pamięci jak np. telefony komórkowe czy odtwarzacze MP3.

SQLite posiada reputację bardzo niezawodnej bazy danych. Prawie 75% kodu zostało poddane szczegółowym testom i całkowitej weryfikacji [Sqlite.org]. SQLite odpowiednio obsługuje niepowodzenia alokacji pamięci oraz błędy zapisu i odczytu z dysku. Wszystko to zostało zweryfikowane przez zautomatyzowane testy symulujące błędy systemu. Oczywiście mimo to nadal występują błędy. Ale w odróżnieniu od

podobnych systemów (głównie komercyjnych) wszystkie błędy są publikowane i w miarę szybko naprawiane.

## 3.2. Istotne cechy SQLite'a

- Obsługa transakcji zgodnych ze standardem ACID, czyli transakcje są atomowe, spójne, izolowane i trwałe nawet wtedy, gdy zostaną przerwane przez błąd programu, błąd systemu operacyjnego czy brak napięcia i restart komputera.
- Brak konfiguracji, nie trzeba nic instalować ani konfigurować.
- Zaimplementowany w większości zgodnie ze standardem SQL92 (wyjątki poniżej)
- Kompletna baza danych przechowywana w pojedynczym pliku, który może być przenoszony między różnymi platformami.
- Obsługa baz danych o wielkości rzędu terabajtów oraz gigabajtowych łańcuchów znaków („stringów") i obiektów binarnych (BLOB-ów).
- Zajmuje mało miejsca na dysku poniżej 275 KB.
- Szybszy od popularnych baz danych opartych na modelu klient-serwer dla większości najbardziej popularnych operacji.
- Napisany w ANSI-C, a używany w powiązaniu z innymi językami, szczególnie mocny związek z językiem skryptowym Tcl.
- Działający na wielu platformach: Linux, MacOSX, OS/2, Win32 i WinCE.
- Kod źródłowy rozpowszechniany na licencji Public Domain.

## 3.3. Zgodność ze standardami SQL

SQLite w większości posiada standardową składnie języka SQL. Istnieją jednak pewne cechy, które pomija i pewne nowe, unikalne właściwości, które dodaje.

Cechy standardu SQL-92, które NIE zostały zaimplementowane w SQLite:

- **FOREIGN KEY (klucz zewnętrzny, klucz obcy),** czyli pole, którego wartość odpowiada kluczowi głównemu w innej tabeli

- **Niektóre właściwości wyzwalaczy (triggerów)**

  Wyzwalacze to procedury wykonywane w odpowiedzi na zdarzenia takie jak np. dodanie czy usunięcie rekordu. W SQLite pominięte zostały takie właściwości triggerów jak: FOR EACH STATEMENT (wszystkie wyzwalacze muszą być FOR EACH ROW) i INSTEAD OF na tabelach (INSTEAD OF możliwy tylko na widokach)

- **Niektóre warianty polecenia ALTER TABLE**

  ALTER TABLE, czyli polecenie zmieniające właściwości istniejącej tabeli. W SQLite wspierane są tylko dwa warianty tego polecenia mianowicie zawierające atrybuty RENAME TABLE (zmiana nazwy tabeli) i ADD COLUMN (dodanie kolumny). Pozostałe rodzaje operacji ALTER TABLE, takie jak DROP COLUMN (usunięcie kolumny), czy ADD CONSTRAINT(dodanie ograniczenia) zostały pominięte.

- **Obsługa zagnieżdżonych transakcji**

  W SQLite obecnie możliwe są tylko pojedyncze transakcje.

- **Operacje łączenia prawostronnego RIGHT OUTER JOIN i pełnego FULL OUTER JOIN**

  Zaimplementowane jedynie operacje łączenia lewostronnego LEFT OUTER JOIN

- **Operacje zapisywania do widoków**

  Widoki (VIEWS), czyli wirtualne tabele w SQLite są tylko do odczytu. Nie można wykonywać na nich operacji DELETE, INSERT i UPDATE.

- **Polecenia GRANT i REVOKE**

  Komendy te służą do nadawania i odbierania uprawnień użytkownikom. SQLite zapisuje i odczytuje dane bezpośrednio z pliku, więc prawa dostępu nadawane są dla pliku z poziomu systemu operacyjnego.

## 3.4. Różnice w porównaniu z innymi bazami danych

**Brak konfiguracji**

SQLite nie wymaga instalowania, wystarczy umieścić w odpowiednim miejscu na dysku plik biblioteki. Nie ma procesu serwera, który musiałby być uruchamiany, zatrzymywany czy konfigurowany. Nie potrzeba administratora, który tworzyłby nową bazę i określał prawa dostępu dla użytkowników. SQLite nie używa także plików konfiguracyjnych. Żadne działania nie są potrzebne w celu odzyskania danych po błędzie systemu.

**System bezserwerowy**

W większości przypadków silniki baz danych implementowane są jako oddzielne procesy serwera. Programy chcąc uzyskać dostęp do danych komunikują się z serwerem używając różnego rodzaju komunikacji międzyprocesowej, wysyłając żądanie do serwera i otrzymując w odpowiedzi wyniki. SQLite nie działa w ten sposób. Proces, który chce uzyskać dostęp do bazy czyta bezpośrednio z pliku bazy na dysku. Nie istnieje pośredni proces serwera. Oczywiście istnieją zarówno zalety jak i wady takiego rozwiązania. Główna zaleta braku procesu serwera, to brak potrzeby instalowania, konfiguracji, zarządzania, inicjalizacji. Każdy program, który ma dostęp do plików dyskowych, ma możliwość dostępu do bazy SQLite.

Z drugiej strony silniki bazodanowe używające serwera mogą dostarczać lepszej ochrony przed nieautoryzowanym dostępem do bazy ze strony aplikacji klienckich. Większość silników bazodanowych oparta jest na architekturze klient-serwer. Z pośród tych, które są bezserwerowe, tylko SQLite pozwala na jednoczesny dostęp do tych samych danych w tym samym czasie.

**Pojedynczy plik z bazą**

Baza SQLite jest zwykłym pojedynczym plikiem, który może znajdować się w dowolnym miejscu na dysku. Jeżeli SQLite może przeczytać ten plik to może przeczytać każdą informacje zapisaną w bazie, a jeżeli plik i katalog, w którym się znajduje mają ustawione prawa do zapisu wtedy SQLite może zmienić każdą daną w bazie. Plik z bazą może być łatwo kopiowany na pamięci przenośne oraz przesyłany mailem.

Inne silniki bazodanowe z reguły zapisują dane w wielu plikach. Często te pliku muszą znajdować się w standardowych lokalizacjach, aby serwer bazy danych miał do nich dostęp. Sprawia to, że dane są lepiej zabezpieczone, ale jednocześnie istnieje trudniejszy dostęp do nich oraz mogą pojawić się problemy z przenoszeniem bazy.

**Stabilny i działający na różnych platformach plik bazy**

Plik z bazą może być przenoszony bez problemu z jednej platformy sprzętowej na inną, nie ma znaczenia czy jest to architektura 32 czy 64 bitowa. Wszystkie maszyny używają tego samego formatu pliku, w którym przechowywana jest baza. SQLite działa na platformach z rodziny Linux, MacOSX, OS/2, Win32 i WinCE.

**Odmienne od innych baz określanie typów danych tzw. manifest typing**

Większość SQL-owych baz danych używa typów statycznych. Typy danych są związane z każdą kolumną w tabeli i tylko dane takiego typu mogą być składowane w określonej kolumnie. SQLite rozluźnia nieco te restrykcje. Typ danych należy do wartości zapisanej w kolumnie, a nie do kolumny, w której ta wartość jest składowana. Umożliwia to wstawianie wartości dowolnego typu w dowolnej kolumnie bez względu na wcześniej zadeklarowany typ owej kolumny. Jest jednak kilka wyjątków od tej reguły: kolumna określona jako INTEGER PRIMARY KEY może przechowywać dane tylko typu INTEGER.

Ze względu na to, że większość innych baz SQL-owych używa statycznego określania typów, niektórzy użytkownicy odbierają jako błąd sposób określania typów w SQLite. Jednak autorzy SQLite twierdzą, że jest to cecha świadomie zaprojektowana, która w praktyce udowadnia, że SQLite jest bardziej niezawodny i prostszy

w użytkowaniu, szczególnie wtedy, gdy używany jest w połączeniu z takimi językami jak Tcl czy Pyton.

**Rekordy zmiennej długości**

Większość znanych baz danych alokuje zmienną ilość przestrzeni dyskowej dla każdego wiersza tabeli. Jeżeli kolumna jest typu np. VARCHAR(100), wtedy silnik bazodanowy zaalokuje 100 bajtów przestrzeni dyskowej bez względu na to jak duża ilość informacji jest aktualnie przechowywana w danej kolumnie.

SQLite natomiast używa tylko wielkości przestrzeni dyskowej aktualnie potrzebnej do składowania informacji zawartej w danym wierszu. Jeśli przechowywany jest jeden znak w kolumnie VARCHAR(100) wtedy tylko jeden bajt przestrzeni dyskowej jest zajęty. Użycie rekordów o zmiennej długości powoduje zmniejszenie wielkości bazy, pozwala także na opisane we wcześniejszym punkcie stosowanie różnych typów danych w pojedynczej kolumnie.

**Rozszerzenia języka SQL**

SQLite dostarcza kilku usprawnień do języka SQL, których nie znajdziemy w innych systemach bazodanowych. SQLite dostarcza członów składniowych takich jak REPLACE i ON_CONFLICT, które dają kontrolę nad rozwiązywaniem wymuszonych konfliktów. SQLite wspiera także komendy ATTACH i DETACH, które pozwalają na użycie kilku niezależnych baz danych w jednym zapytaniu. SQLite definiuje również programowy interfejs (API), który pozwala użytkownikom na dodawanie nowych SQL-owych funkcji.

## 3.5. Zastosowanie SQLite'a

### 3.5.1. Rodzaje zastosowań

**Format plikowy dla aplikacji**

SQLite używany jest z dużym powodzeniem jako dyskowy format plikowy dla aplikacji desktopowych takich jak: narzędzia do analiz finansowych, pakiety CAD, programy do prowadzenia archiwum itd. Tradycyjna operacja znajdująca się w większości aplikacji, czyli Plik/Otwórz" wykonywana jest przez funkcję sqlite3_open() i komendę BEGIN TRANSACTION w celu uzyskania wyłącznego dostępu do zawartości.

Operację „Plik/Zapisz" wykonuje komenda COMMIT.

Tymczasowe wyzwalacze mogą być dodane do bazy, aby zapisać wszystkie zmiany w tymczasowej tablicy logów „undo/redo". Zmiany te mogą być cofnięte, kiedy użytkownik uruchomi polecenie Undo lub Redo znajdujące się w aplikacji.

**Wbudowane urządzenia i aplikacje**

Ponieważ baza SQLite wymaga niewielkich zabiegów administracyjnych jest dobrym wyborem dla urządzeń i usług, które musza działać bez nadzoru i wsparcia użytkownika. SQLite jest dobrym rozwiązaniem dla telefonów komórkowych, palmtopów i innych urządzeń podobnego typu.

**Strony internetowe**

SQLite świetnie pracuje jako silnik bazodanowym dla stron internetowych o małym lub średnim ruchu odwiedzających, czyli ok. 99,9% wszystkich stron. Liczba odwiedzin, z jaką radzi sobie SQLite zależy oczywiście na jak „ciężkich" stronach jest użyty. Generalnie mówiąc, każda strona, która ma mniej niż 100 tysięcy odwiedzin dziennie powinna dobrze współpracować z SQLitem. Oczywiście nie jest to sztywna górna granica, bo SQLite był testowany na stronach z 10-krotnie większą liczba odwiedzin.

**Wewnętrzna lub tymczasowa baza danych**

Dla programów mających dużo danych, które musza być posortowane w różnorodny sposób, łatwiej i szybciej jest załadować dane do wewnętrznej bazy i używać zapytań z klauzulami JOIN i ORDER BY do wyciągnięcia danych w potrzebnej formie i porządku niż próbować zakodować te same operacje ręcznie. Użycie bazy w ten sposób daje programom także większą elastyczność, ponieważ nowe kolumny i indeksy mogą być dodane bez potrzeby ponownego kodowania każdego zapytania.

**Narzędzie do analizy danych**

Doświadczony użytkownik SQL-a może stosować w SQLite linię poleceń do analizy danych. Surowe dane mogą być zaimportowane z pliku CSV, następnie mogą być pocięte i rozbite w celu generowania szerokiej gamy podsumowujących raportów. Możliwe wykorzystanie zawiera analizę logów stron internetowych, analizę statystyk sportowych, analizy wyników eksperymentalnych.
To samo można oczywiście zrobić używając komercyjnych baz danych, ale zaletą użycia SQLite'a w tym przypadku jest to, że SQLite nie wymaga konfiguracji, a wynikowa baza danych jest pojedynczym plikiem.

### 3.5.2. Produkty, które korzystają z bazy SQLite

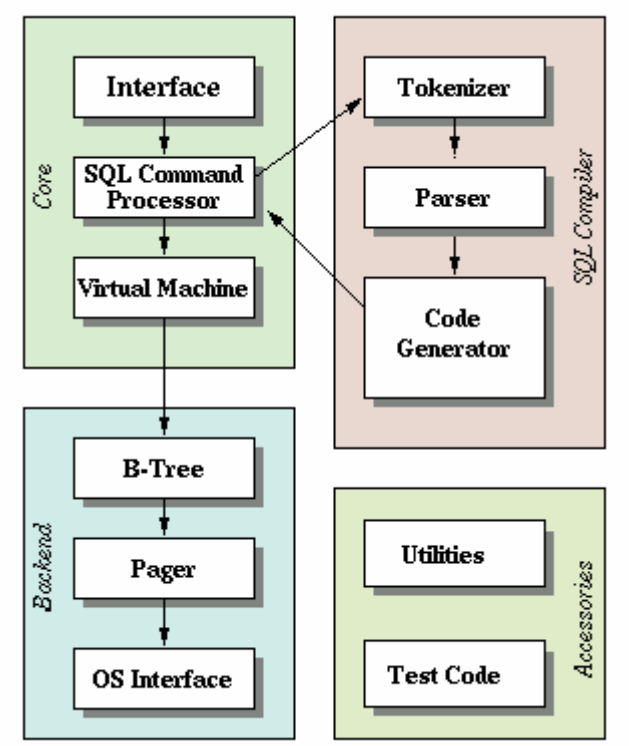
Popularność SQLite'a rośnie z miesiąca na miesiąc i coraz więcej znanych korporacji zaczyna wykorzystywać go w swoich produktach i projektach. Są to m.in.: [sqlite.org]

- Adobe, który używa SQLite'a jako formatu plikowego dla aplikacji z rodziny Photoshop Lightroom oraz Acrobat Readera.
- Apple wykorzystuje SQLite'a w wielu aplikacjach związanych z Mac OS-X, m.in. w Apple Mail, Safari i Aperture. Niepotwierdzone informacje mówią także o wykorzystaniu SQLite w iPhone i iPod.

- Mozilla Firefox powoli zastępuje swój format plikowy bazą SQLite, którą obecnie wykorzystuje do przechowywania profili użytkowników.
- Philips wykorzystuje SQLite w odtwarzaczach Mp3 do przechowywania metadanych na temat utworów muzycznych.
- PHP posiada wbudowaną bazę SQLite.
- Sun w systemie Solaris 10 używa bazę SQLite jako formatu składowania zastępując SQLitem tradycyjny unixowy plik /etc/inittab.
- Google wykorzystuje SQLite'a w Google Gears, a także w platformie dla telefonów komórkowych Android.
- Symbian, będący powszechnie stosowanym systemem operacyjnym w telefonach komórkowych również korzysta z SQLite'a

## 3.6. Architektura

Jak już wspomniano we wcześniejszych rozdziałach, SQLite nie jest typowym systemem bazodanowym, a jedynie biblioteką, która implementuje bezserwerowy mechanizm umożliwiający składowanie danych oraz dostęp do nich przy użyciu komend SQL-a. Zatem przyjrzyjmy się dokładniej jak wygląda ten mechanizm od środka i co dzieje się w czasie pomiędzy wpisaniem komendy SQL-owej, a umieszczeniem danych w pliku bazy. Na rysunku 1 przedstawione zostały kluczowe komponenty SQLite'a oraz relacje miedzy nimi.




Rysunek 1. Diagram blokowy prezentujący architekturę SQLite'a.
[źródło: sqlite.org]

Jak widać na diagramie, w architekturze SQLite'a można wyszczególnić cztery główne bloki: rdzeń (Core), kompilator języka SQL (SQL Compiler), mechanizm składujący (Backend) oraz dodatki (Accessories).

Rdzeń odpowiada za obsługę zapytań oraz dostęp do danych zapisanych w pliku bazy. Można w nim wyróżnić takie elementy jak: interfejs (Interface), mechanizm obsługujący zapytania SQL-owe (SQL Command Procesor) oraz maszynę wirtualną (Virtual Machine).

Kompilator języka SQL przetwarza polecenia SQL-owe na kod maszynowy. Składa się z analizatora leksykalnego (tokenizer), analizatora składniowego (parser) i generatora kodu.

Mechanizm składujący ma za zadanie utrzymanie odpowiedniej struktury danych przechowywanych w bazie. Wykorzystuje on do tego celu tzw. B-drzewa.

A teraz nieco bardziej szczegółowo przyjrzyjmy się przetwarzaniu zapytań i operacjom wykonywanym przez silnik SQLite'a.

Użytkownik uzyskuje dostęp do bazy dzięki interfejsowi. Interfejs umożliwia interakcję między bazą, a użytkownikiem dostarczając zbioru procedur i funkcji obsługujących operacje wejścia/wyjścia. Interfejs SQLite'a pozwala na dostęp do bazy przy pomocy wielu języków programowania m.in. C/C++, Tcl, PHP, Java, Ruby, Pyton, Perl. SQLite posiada również interfejs powłokowy.

Zapytania SQL-owe wprowadzane przy użyciu interfejsu w postaci łańcucha znaków przesyłane są do kompilatora języka SQL. Składnia zapytania SQL-owego przekształcana jest w analizatorze leksykalnym na symbole zwane jednostkami leksykalnymi, które wysyłane są do analizatora składniowego (generatora parserów). SQLite używa generatora parserów o nazwie Lemon opartego na wstępującej metodzie analizy składniowej LARL. Następnie struktura powstała w wyniku działania parsera trafia do generatora kodu, gdzie tworzony jest kod programu dla maszyny wirtualnej.

Kod ten zwracany jest do mechanizmu obsługującego zapytania SQL i stamtąd wysyłany do maszyny wirtualnej, na której zostaje wykonany. Maszyna wirtualna posiada stos, który używany jest do przejściowego składowania danych.

Baza danych utrzymywana jest na dysku przy użyciu struktury danych zwanej B-drzewem. Oddzielne B-drzewo tworzone jest dla każdej tabeli i każdego indeksu. Wszystkie B-drzewa przechowywane są w tym samym pliku.

SQLite posiada także mechanizm trzymania stron B-drzew w pamięci podręcznej tzw. page cache. Wykorzystywany jest on przy transakcjach, gdzie dane zapisywane są do pamięci podręcznej aż do momentu potwierdzenia transakcji (commit).

W celu zapewnienia przenośności między systemami operacyjnymi typu POSIX i Win32, SQLite używa abstrakcyjnej warstwy do komunikowania się z systemem operacyjnym. Systemy operacyjne, z którymi współpracuje SQLite to Windows, Unix, MacOS.

Wśród dodatków, jakie zawiera biblioteka SQLite, a które nie są bezpośrednio związane z obsługą zapytań SQL i składowaniem danych, znajdują się: procedury alokujące pamięć, funkcje porównujące łańcuchy znaków, tabele hash-y, własna implementacja printf() oraz generatora liczb losowych, a także kody testów.

## 3.7. Typy danych

**Klasy składowania**

W SQLite 2.8 wszystkie wartości zarówno tekstowe jak i liczbowe składowane były w bazie jako tekst ASCII. SQLite 3.0 wprowadza możliwość składowania liczb całkowitych (INTEGER) i zmiennoprzecinkowych (REAL) w bardziej kompaktowej formie oraz zdolność przechowywania obiektów binarnych (BLOB-ów).

Każda z wartości przechowywanych w bazie należy do jednej z poniższych klas składowania:

- NULL
- INTEGER – wartość jest liczbą całkowitą (ze znakiem) zapisaną na 1, 2, 3, 4, 6 lub 8 bajtach w zależności od rozmiaru wartości
- REAL – wartość jest liczbą zmiennoprzecinkową przechowywana jako 8-bajtowa liczba rzeczywista
- TEXT – wartość jest tekstowym łańcuchem znaków przechowywanym przy użyciu kodowania UTF-8, UTF-16BE lub UTF-16-LE
- BLOB – wartość jest binarnym obiektem danych, składowanym bez żadnych modyfikacji, czyli tak jak otrzymaliśmy ten obiekt na wejściu

SQLite sam rozpoznaje typ wartości znajdującej się w kolumnie i zapisuje ją w bazie jako obiekt jednej z powyższych klas. Wartość znajdująca się w określonej kolumnie może być innego typu niż zdefiniowany przy tworzeniu tabeli typ kolumny. Typ kolumny ma znaczenie przy porównywaniu i sortowaniu danych, bo właśnie na podstawie typu kolumny wybierana jest metoda porównywania czy sortowania.

**Automatyczne przydzielanie klas składowania przez SQLite**

Wartości przechowywane są jako typ:

- TEXT, jeżeli w zapytaniu SQL-a ujęte są w pojedynczy lub podwójny cudzysłów
- INTEGER, jeżeli w zapytaniu występują jako liczby nieujęte w cudzysłów i nieposiadające separatora lub wykładnika

- REAL, jeżeli są liczbą nieujętą w cudzysłów oraz posiadają separator w postaci kropki lub wykładnik
- BLOB, jeżeli zostały dostarczone za pomocą funkcji interfejsu API sqlite3_bind_blob()

**Typy kolumn**

W SQLite typy danych związane są z wartością znajdującą się w kolumnie, a nie z kolumną (odwrotnie niż w innych systemach bazodanowych). Każda kolumna, z wyjątkiem kolumny określonej jako INTEGER PRIMARY KEY, może przechowywać dane dowolnego typu. Kolumna określona jako klucz główny musi zawierać 32-bitową liczbę całkowitą. Wszelkie próby wstawienia wartości innego typu do takiej kolumny spowodują błąd.

W celu zwiększenia kompatybilności między bazą SQLite, a innymi bazami danych zaleca się składowanie danych odpowiedniego typu dopasowanego do typu kolumny.

SQLite definiuje następujące typy kolumn:

- TEXT
- NUMERIC
- INTEGER
- REAL
- NONE

**Konwersja typów**

W kolumnie określonej jako TEXT zgodnie z zaleceniami powinno się przechowywać wartości odpowiadające klasom składowania: TEXT, BLOB i NULL. W kolumnie NUMERIC można przechowywać dane wszystkich pięciu klas składowania, przy czym, gdy próbujemy wstawić tam wartość tekstową, to SQLite przed zapisaniem jej, próbuje zamienić ją na typ INTEGER lub REAL. Jest to tzw. konwersja typów. Jeśli konwersja się powiedzie to wartość zapisywana jest w klasie składowania INTEGER lub REAL. Jeśli natomiast nie uda się przeprowadzić konwersji to wartość zapisywana jest jako

TEXT. Podobne zmiany typów zachodzą przy próbie wstawienia „stringa" do kolumn typu INTEGER i REAL.

W celu kompatybilności z innymi bazami danych, SQLite dokonuje także konwersji nazw typów występujących w innych bazach na nazwy typów z SQLite'a. Jeżeli przy tworzeniu tabeli podamy typ niewystępujący w SQLite, a zdefiniowany w innych silnikach bazodanowych to możemy mieć pewność, że nie spowoduje to błędu.

Konwersje nazw typów:

- INT zamieniany jest na INTEGER
- CHAR i VARCHAR zamieniane są na TEXT
- FLOAT i DOUBLE zamieniane są na REAL
- Jeśli kolumna nie posiada typu lub posiada typ BLOB to zamieniany jest on na NONE
- Pozostałe typy zamieniane są na NUMERIC

**Porównywanie i sortowanie danych w zależności od typu**

Przy porównywaniu i sortowaniu danych, kolumny lub wyrażenia mogą należeć do jednej z dwóch grup, mogą być numeryczne lub tekstowe. Typ kolumny określa, do której grupy zostanie zaliczona kolumna i jaka metoda porównywania lub sortowania będzie użyta. Jeżeli dane są tekstowe to do porównywania ich używana jest standardowa funkcja języka C *memcmp()* lub *strcmp()*. Funkcje te porównują dwa łańcuchy znaków bajt po bajcie i zwracają pierwszą napotkaną różnice.

## 3.8. Język zapytań

Językiem zapytań do bazy SQLite jest standardowy język SQL zdefiniowany w standardzie SQL-92.

### 3.8.1. Tworzenie tabel

Tabele tworzymy za pomocą polecenia CREATE TABLE.

Składnia: *CREATE TABLE nazwa_tabeli (definicje kolumn);*

Definicje kolumn zawierają: nazwę kolumny, typ oraz ograniczenia takie jak:

- NOT NULL - oznacza, że pole musi posiadać wartość różną od NULL
- PRIMARY KEY - definiuje pole tabeli jako klucz główny, pole to musi być typu INTEGER
- UNIQUE - oznacza pole tabeli jako unikatowe, czyli wartości w kolumnie nie mogą się powtarzać
- CHECK - wymaga wartości określonych w podanych warunkach
- DEFAULT - definiuje wartość domyślna, która zostanie wstawiona w dane pole tabeli
- COLLATE - określa, jaką funkcję użyć do porównywania wartości tekstowych

### 3.8.2. Wstawianie, aktualizowanie i usuwanie danych

W celu wstawienia danych do tabeli korzystamy z polecenia INSERT. Polecenie to dodaje nowy wiersz w istniejącej tabeli.

Składnia: *INSERT INTO nazwa_tabeli (lista_kolumn) VALUES (wartosci);*

W powyższym poleceniu nazwy kolumn i wartości oddzielamy przecinkami. Lista kolumn jest opcjonalna, jeśli ją pominiemy to liczba wartości musi odpowiadać liczbie kolumn w tabeli.

Istnieje także możliwość aktualizacji danych znajdujących się w tabeli. Służy do tego polecenie UPDATE.
Składnia: *UPDATE nazwa_tabeli SET nazwa_kolumny = wartosc;*

Rekordy z tabeli możemy usunąć poleceniem DELETE.
Składnia: *DELETE FROM nazwa_tabeli;*
Jeśli nie chcemy usunąć wszystkich rekordów z tabeli to używamy klauzuli WHERE z wyrażeniem ograniczającym.

### 3.8.3. Pobieranie danych

Dane pobieramy z tabeli przy pomocy polecenia SELECT. Wynikiem działania zapytania SELECT jest zero lub więcej wierszy tabeli.

Składnia: *SELECT lista_kolumn FROM nazwa_tabeli;*

Użycie znaku * zamiast listy kolumn spowoduje pobranie danych ze wszystkich kolumny z określonej tabeli.

W zapytaniu SELECT dostarcza także różnych klauzuli umożliwiających zawężanie przeszukiwania tabeli, czy grupowanie i sortowanie wyników. Są to klauzule:

- WHERE – ogranicza liczbę wierszy zwracanych w wyniku zapytania przez zastosowanie operatorów przypisania i porównania
- GROUP BY – scala rekordy o tych samych wartościach w kolumnach umieszczonych przy tym wyrażeniu
- ORDER BY – klauzula ta służy do sortowania rekordów wg kolumn umieszczonych po tej klauzuli
- LIMIT – określa górną granice zwracanych rezultatów

Z poleceniem SELECT związane są także złączenia (JOIN). Wykorzystuje się je, gdy jako wynik chcemy uzyskać dane z kilku tabel.

Ze względu na sposób łączenia dzieli się operacje złączeń na trzy grupy:

- złączenia zewnętrzne (OUTER) – służą do ograniczania w zbiorze wynikowym wierszy z jednej tabeli, podczas gdy wiersze z drugiej nie zostają ograniczone
- złączenia wewnętrzne (INNER) – opierają się na ograniczeniu iloczyny kartezjańskiego dwóch tabel w oparciu o pewne relacje kolumn z tych tabel
- złączenia krzyżowe (CROSS) – odpowiadają iloczynowi kartezjańskiemu wykonanemu na wierszach z obu tabel

Złączenia zewnętrzne zwykle występują w trzech odmianach tj. lewostronne, prawostronne oraz pełne. SQLite obsługuje jedynie złączenia lewostronne, czyli takie, w których do wyniku złączenia wprowadzone zostają wszystkie wiersze z tabeli stojącej po lewej stronie złączenia oraz pasujące do ograniczenia wiersze z drugiej tabeli.

### 3.8.4. Pozostałe operacje na tabelach

Polecenie ALTER TABLE pozwala użytkownikowi na zmianę nazwy tabeli oraz dodanie oraz dodanie nowej kolumny do już istniejącej tabeli.

Składnia: *ALTER TABLE nazwa_tabeli opcje;*

Opcje: RENAME TO nowa_nazwa_tabeli (zmiana nazwy tabeli)
ADD nazwa_kolumny (dodanie kolumny)

W porównaniu do innych baz, w SQLite nie ma możliwości zmiany nazwy czy usuwania kolumn oraz dodawania i usuwania ograniczeń z kolumn. W przypadku dodania nowej kolumny jest ona wstawiana na końcu listy kolumn już istniejących.

Do usuwania tabel służy polecenie DROP TABLE.
Składania: *DROP TABLE [IF EXISTS] nazwa_tabeli;*
Gdy usuwamy tabele, usunięte zostają również wszystkie indeksy i wyzwalacze związane z tą tabelą.

### 3.8.5. Łączenie baz

W SQLite istnieje możliwość przyłączenia dodatkowych plików z bazami do istniejącego połączenia z bazą. Służy do tego polecenie ATTACH DATABASE.
Składnia: *ATTACH DATABASE nazwa_pliku_bazy AS nazwa_bazy;*

Przyłączone pliki można odłączyć używając polecenia DETACH DATABASE.

### 3.8.6. Indeksy

Indeksy (zwane równie kluczami) to struktury przyspieszające wyszukiwanie danych w bazie. Podczas przeszukiwania bazy bez zastosowania indeksów, SQLite musi przeszukać każdą tabele wiersz po wierszu w celu dopasowania wartości. Natomiast, gdy polom wyszukiwania nadamy indeksy, SQLite będzie mógł ominąć ten czasochłonny proces, w taki sposób, że odwoła się do specjalnie stworzonego pola, dzięki któremu odnalezienie lokalizacji interesujących nas danych będzie szybsze.

Jeżeli z góry wiadomo, że dane w konkretnym polu tabeli będą niepowtarzające się, czyli każda wartość wystąpi tylko w jednym rekordzie, to wtedy pole to możemy oznaczyć jako UNIQUE (dokonujemy tego w trakcie tworzenia tabeli). Spowoduje to utworzenie indeksu, który będzie zapobiegał duplikowaniu wartości w danej kolumnie tabeli. Indeks UNIQUE wymusza, przed wprowadzeniem danych do tabeli, każdorazowe sprawdzenie kolumny pod kątem istnienia już danej wartości.

Istnieje również możliwość dodawania indeksów do już istniejących tabel, dokonujemy tego z pomocą polecenia CREATE INDEX.

Składnia: *CREATE INDEX nazwa_indeksu ON nazwa_tabeli (nazwa_kolumny);*

Indeks usuwamy poleceniem: *DROP INDEX nazwa_indeksu;*

**Klucz główny (PRIMARY KEY)**

Klucz główny jest specjalnym rodzajem indeksu. Jeżeli jakieś pole w tabeli oznaczymy jako PRIMARY KEY to staje się ono unikatowym identyfikatorem całego wiersza. Pole mające status klucza głównego musi przechowywać całkowite wartości liczbowe (INTEGER), dlatego przy tworzeniu tabeli musimy je określić jako INTEGER PRIMARY KEY.

Jeżeli przy wstawianiu danych w miejsce wartości dla pola określonego jako klucz główny wpiszemy NULL to SQLite sam wypełnia takie pole wpisując wartość 1 dla pierwszego wiersza, 2 dla drugiego itd. Działa to wtedy analogicznie jak auto_increment w MySQL.

### 3.8.7. Wyzwalacze

Wyzwalacz (trigger) to procedura wykonywana automatycznie jako reakcja na pewne zdarzenie w tabeli. Do tworzenia wyzwalaczy służy polecenie CREATE TRIGGER.

Składnia: *CREATE TRIGGER nazwa_wyzwalacza [BEFORE | AFTER] zdarzenie ON nazwa_tabeli BEGIN akcja_wyzwalacza END;*

W SQLite istnieją dwa typy wyzwalaczy: BEFORE – wykonywane przed instrukcją generującą zdarzenie i AFTER – wykonywane po instrukcji generującej zdarzenie. Brak jest wyzwalacza INSTEAD OF występującego w niektórych bazach i wykonywanego zamiast instrukcji generującej zdarzenie.

Wykonanie wyzwalacza następuje w reakcji na następujące zdarzenia:

- dopisanie nowego rekordu do bazy w wyniku wykonania instrukcji INSERT

- zmianę zawartości rekordu (UPDATE)
- usunięcie rekordu (DELETE)

Wyzwalacz możemy usunąć poleceniem:

*DROP TRIGGER nazwa_wyzwalacza;*

Wyzwalacze usuwane są automatycznie, gdy usuniemy tabelę, z którą są powiązane.

### 3.8.8. Widoki

Widok (perspektywa) to logiczna struktura, wirtualna tabela tworzona w locie na podstawie wyniku zapytania SELECT, umożliwiająca dostęp do podzbioru kolumn i wierszy powstałego po wykonaniu owego zapytania. Z widoku można pobierać dane tak jak ze zwykłej tabeli.

Widoki tworzymy poleceniem CREATE VIEW.

Składnia: *CREATE VIEW nazwa_widoku AS zapytanie_select;*

Widok usuwamy poleceniem *DROP VIEW nazwa_widoku;*

### 3.8.9. Funkcje stosowane w zapytaniach

SQLite umożliwia stosowanie w zapytaniach funkcji zarówno wbudowanych jak i zdefiniowanych przez użytkownika. Funkcje wbudowane można podzielić na funkcje agregujące, funkcje daty i czasu oraz pozostałe funkcje.

**Funkcje agregujące**

Funkcje agregujące to funkcje zwracające rezultat z wartości wielu rekordów. Są to na przykład funkcje obliczające sumę czy średnią ze zbioru wierszy.

SQLite udostępnia następujące funkcje agregujące:

**avg(*X*)** – zwraca wartość średnią ze wszystkich rekordów nie zawierających wartości pustej (NULL) z wyrażenia X.

**count(*X*) i count(*)** – pierwsza forma zwraca liczbę rekordów, w których wyrażenie X nie ma wartości NULL, druga - liczbę wszystkich rekordów łącznie z zawierającymi wartości puste.

**group_concat(*X*) i group_concat(*X*,*Y*)** – wynikiem działania tych funkcji jest ciąg znaków powstały w wyniku złączenia niepustych wartości wyrażenia X, parametr Y jest separatorem, jeśli go pominiemy to domyślnie za separator przyjmowany jest przecinek.

**max(*X*)** – zwraca maksymalną wartość ze wszystkich rekordów z wyrażenia X

**min(X)** – zwraca minimalną niepustą wartość ze wszystkich rekordów z wyrażenia X, jeżeli wszystkie wartości są NULL wtedy funkcja zwraca NULL

**sum(*X*) i total(*X*)** – zwracają liczbowe sumy ze wszystkich nie pustych wartości rekordów, jeżeli nie ma wartości niepustych to sum(X) zwraca NULL, natomiast total(X) – 0.0. Wynikiem zwracanym przez total(X) jest zawsze wartość typu FLOAT, natomiast wartością zwracaną przez sum(X) jest INTEGER w przypadku, gdy wszystkie wartości są typu INTEGER i FLOAT, gdy przynajmniej jedna z wartości jest innego typu niż INTEGER lub NULL.

**Funkcje daty i czasu**

SQLite udostępnia pięć funkcji związanych z datą i czasem. Są to:

**date()** – funkcja zwracająca aktualną datę w formacie YYYY-MM-DD

**time()** – funkcja zwracająca aktualny czas w formacie HH:MM:SS

**datetime()** – funkcja zwracająca aktualną datę i czas w formacie YYYY-MM-DD HH:MM:SS

**julianday()** – funkcja zwracająca liczbę dni, które upłynęły od początku tzw. epoki juliańskiej, czyli od 1 stycznia 4713 roku p.n.e.

**strftime()** – funkcja zwracająca datę sformatowaną wg podanego jako argument wzorca:
%d – dzień miesiąca w postaci: 00
%f – sekundy ułamkowe: SS.SSS
%H – godzina: 00-24
%j – dzień roku: 001-366
%J – dzień juliański
%m – miesiąc: 01-12
%M- minuta: 00-59
%s – sekundy od 1970-01-01 tzw. czas unixowy
%S - sekundy: 00-59
%w – dzień tygodnia 0-6, gdzie 0 to niedziela
%W – tydzień roku: 00-53
%Y - rok: 0000-9999
Przykład: *strftime("%Y-%m-%d")* zwraca datę w formacie YYYY-MM-DD

## 3.9. Optymalizacja SQLite'a

Optymalizacja to zakres działań mających na celu poprawę wydajności systemu bazodanowego, przede wszystkim przez zwiększenie szybkości działania i zmniejszenie wykorzystania zasobów komputera.

W przypadku SQLite'a klucz do poprawy wydajności leży przed wszystkim w ograniczeniu i optymalizowaniu operacji zapisu i odczytu danych na dysku.

Na optymalizację bazy SQLite wpływ mają następujące elementy:

- **Odpowiednie ustawienie stałych konfiguracyjnych PRAGMA**

  PRAGMA cache_size – określa maksymalną liczbę stron btree trzymaną w pamięci podręcznej, dostęp do tych stron jest znacznie szybszy, ponieważ nie wymaga operacji dyskowych. Można zwiększyć wartość cache_size, przez co system będzie mógł przechowywać większą liczbę stron w pamięci. Nie będzie to miało wpływu na marnotrawienie zasobów serwera, ponieważ pamięć alokowana jest dopiero w trakcie wykonywania zapytania i tylko w potrzebnej ilości.

  PRAGMA synchronous – ustawienie tej stałej na OFF spowoduje, że SQLite nie będzie czekał, aż operacja zapisu danych na dysk zostanie wykonana w całości, jedynie zleci zapis i zajmie się kolejnymi operacjami, spowoduje to szybsze działanie bazy, ale zmniejszy kontrolę integralności danych

  PRAGMA count_changes – gdy jest włączone (ON), SQLite zlicza rekordy zmodyfikowane przez operacje INSERT, UPDATE, DELETE. Jeżeli niepotrzebne są nam te dane to możemy wyłączyć tę opcję, co nieznacznie przyspieszy wykonywanie tych operacji.

  PRAGMA temp_store – określa sposób przechowywania plików tymczasowych, możliwe są trzy ustawienia: DEFAULT(0), FILE(1), MEMORY(2). Użycie pamięci (2) spowoduje znaczne przyspieszenie wykonywania operacji.

- **Stosowanie transakcji**

  SQLite używa transakcji domyślnie dla każdego zapytania, ma to na celu zwiększenie niezawodności. Jest to jednak bardzo czasochłonne, ponieważ wymaga każdorazowo otwarcia pliku, zapisania danych oraz zamknięcia pliku dla pojedynczego zapytania. Można tego uniknąć grupując szereg zapytań w transakcje korzystając z poleceń BEGIN TRANSACTION i END TRANSACTION (COMMIT).

- **Używanie indeksów**

  Indeksy utrzymują kolejność sortowania kolumny lub zbioru kolumn w tabeli. Umożliwia to pobranie wartości bez potrzeby skanowania całej tabeli, co znacznie przyspiesza operacje wyszukiwania danych. Wstawianie danych do tabeli z indeksami jest nieco wolniejsze niż w przypadku tabeli bez indeksów.

- **Porządkowanie bazy danych**

  Dodawanie i usuwanie rekordów powoduje fragmentację bazy i przez to zwiększenie rozmiaru pliku bazy. Najprostszym sposobem poradzenia sobie z tym problemem jest użycie polecenia VACUUM, które uporządkuje naszą bazę.

- **Bazy działające w pamięci**

  Dostęp do pamięci operacyjnej jest dużo szybszy niż dostęp do pliku dyskowego, dlatego, jeżeli zależy nam na szybkości warto korzystać z baz tworzonych w pamięci operacyjnej. Bazę taką tworzymy tak samo jak zwykłą bazę w pliku, z tym, że zamiast nazwy pliku wpisujemy MEMORY. Baza taka niestety jest bazą tymczasową.

## 3.10. Transakcje

Transakcja to zbiór operacji na bazie, które stanowią w istocie pewną całość i powinny być wykonane wszystkie lub żadna z nich.

SQLite obsługuje transakcje zgodne ze standardem ACID.

Cechy transakcji (standard ACID):

- Atomowość (Atomicity) – transakcja jest niepodzielna, wykonywana jest w całości lub wcale
- Spójność (Consistency) – po wykonaniu transakcji stan bazy będzie spójny
- Izolacja (Isolation) – transakcje, które wykonują się współbieżnie, nie widzą się nawzajem

- Trwałość (Durability) – wykonanie transakcji może być przerwane na chwilę np. w wyniku awarii systemu, a następnie wznowione, zachowując trwałość całej operacji

W SQLite podobnie jak w silniku InnoDB bazy MySQL, wszystkie działania użytkownika przebiegają w obrębie transakcji. Każda instrukcja SQL tworzy pojedynczą transakcje. Warto jednak grupować w transakcje więcej poleceń, ponieważ zwiększa to wydajność bazy. SQLite całą grupę poleceń wykonuje szybciej niż każde polecenie oddzielnie, związane jest to przede wszystkim z mechanizmem dostępu do pliku oraz synchronizacją danych na dysku. W przypadku transakcji zawierającej grupę poleceń plik otwierany i zamykany jest tylko raz, natomiast, jeśli mamy polecenia nie zgrupowane w transakcję to otwieranie i zamykanie pliku następuje dla każdego polecenia z osobna i dla każdego polecenia następuje synchronizacja danych, która jest procesem bardzo powolnym.

Ręcznie transakcje wywołujemy za pomocą polecenia BEGIN i zatwierdzamy poleceniem COMMIT (lub END TRANSACTION). Pomiędzy tymi poleceniami umieszczamy grupę zapytań. Transakcja wykona się, jeżeli wykonają się wszystkie zapytania.

Każdy etap transakcji jest zapisywany w dzienniku logów, dzięki czemu w razie awarii, można odtworzyć stan bazy sprzed transakcji, która nie została zamknięta.

Transakcję można również zamknąć unieważniając zmiany wprowadzone przez wszystkie zapytania SQL znajdujące się w transakcji. Służy do tego polecenie ROLLBACK.

Transakcje mogą być jednego z trzech typów:

- odroczone (DEFFERED)
- natychmiastowe (IMMEDIATE)
- wyłączne (EXCLUSIVE)

Typ transakcji określamy podając go po słowie BEGIN. Domyślnym typem transakcji jest DEFFERED.

Transakcja odroczona oznacza, że z chwilą rozpoczęcia transakcji dostęp do pliku nie jest blokowany. Blokada zakładana jest dopiero w chwili rozpoczęcia pierwszej operacji odczytu lub zapisu do bazy. W czasie pomiędzy rozpoczęciem transakcji, a wykonaniem pierwszej operacji inny wątek może rozpocząć swoją transakcje i dokonać zapisu lub odczytu.

Transakcja natychmiastowa daje gwarancję, że żaden inny wątek nie będzie mógł zapisywać danych do bazy, ani rozpocząć transakcji natychmiastowej lub wyłącznej, aż do czasu zakończenia bieżącej transakcji natychmiastowej. Inne procesy w tym czasie mogą czytać z pliku bazy.

Transakcja wyłączna zaraz po rozpoczęciu zakłada zamki na plik blokując zarówno odczyt jak i zapis do pliku. Daje ona gwarancję, że do czasu zakończenia transakcji żaden inny wątek nie będzie ani czytał ani zapisywał danych do bazy.

## 3.11. Ograniczenia SQLite'a

W punkcie tym zostały opisane ograniczenia SQLite'a w sensie wielkości i rozmiarów, jakie nie mogą zostać przekroczone.

- Maksymalna długość łańcucha znaków (string) lub obiektu binarnego (BLOB) obsługiwana obecnie przez SQLite'a wynosi $2^{31}$-1 bajtów (domyślnie długość ta określona jest na 1 miliard bajtów, ale można ją zwiększyć wpisując przy kompilacji biblioteki SQLite inną wartość do stałej SQLITE_MAX_LENGTH). Parametr SQLITE_MAX_LENGTH określa jednocześnie maksymalną liczbę bajtów w wierszu.

- Maksymalna liczba kolumn w jednej bazie obsługiwana przez SQLite'a wynosi 32676 (domyślnie w SQLITE_MAX_COLUMN wpisane jest 2000, zmiana tej liczby możliwa przy kompilacji biblioteki)

- Maksymalna liczba złączonych tabel to 64. Limit ten wynika z architektury SQLite, mianowicie generator kodu w optymalizatorze zapytań używa bitmap z jednym bitem na złączenie tabeli.

- Maksymalna liczba połączonych baz tzn. baz, na których operacje wykonywane są w jednym połączeniu wynosi 30 na maszynach 32 bitowych i 62 na 64 bitowych (domyślnie ustawiona w stałej SQLITE_MAX_ATTACHED wynosi 10).

## 3.12. Licencja

SQLite udostępniany jest na licencji Public Domain. Oznacza to, że każdy bez opłaty może kopiować, używać, modyfikować, publikować i sprzedawać oryginalny kod SQLite'a, zarówno w wersji skompilowanej jak i kodu źródłowego, w celach niekomercyjnych i komercyjnych. Licencję Public Domain odróżnia od innych licencji to, że produkt dostępny na jej mocy nie posiada praw autorskich, jest on dobrem wspólnym.

Przyjrzyjmy się także, na jakich warunkach udostępniają swoje produkty konkurenci SQLite'a, czyli MySQL, PostgreSQL i Firebird.

MySQL jest dostępny na licencji wolnego oprogramowania GNU GPL (GNU General Public License) oraz licencji komercyjnej. Istnieje wymóg nabycia wersji komercyjnej, jeśli zamierzamy dystrybuować aplikację korzystającą z MySQL. Zmiana licencji (wprowadzenie licencji komercyjnej) nastąpiła od wersji 4.1 i była jednym z powodów rezygnacji w PHP z domyślnie włączonej obsługi MySQL na rzecz SQLite'a.

PostgreSQL dostępny jest na licencji BSD (Berkeley Software Distribution). Jest to jedna z licencji wolnego oprogramowania. Pozwala ona używać danego oprogramowania na wszelkie możliwe sposoby, byleby towarzyszyła temu załączona kopia licencji. Oznacza to możliwość stosowania bazy również w projektach komercyjnych bez żadnych opłat.

Firebird udostępniany jest na licencji InterBase Public License (licencja wolnego oprogramowania), opracowanej przez firmę Inprise Copr na potrzeby udostępnienia kodu źródłowego serwera InterBase 6.0. Na podstawie bazy InterBase 6.0 powstał projekt bazy Firebird. Licencja ta wymaga udostępniania kodu źródłowego aplikacji wraz z aplikacją, w której wykorzystana jest baza.

# 4.

# Składowanie danych przestrzennych w SQLite

## 4.1. Czym są dane przestrzenne?

Dane przestrzenne to dane dotyczące obiektów znajdujących się w przyjętym układzie współrzędnych. Dane te określają położenie, wielkość, kształt oraz związki topologiczne zachodzące między tymi obiektami. Jeżeli mamy do czynienia z obiektami znajdującymi się na powierzchni Ziemi to wtedy dane przestrzenne możemy nazwać danymi geograficznymi.

Dane przestrzenne dzielą się na dane:

- rastrowe
- wektorowe

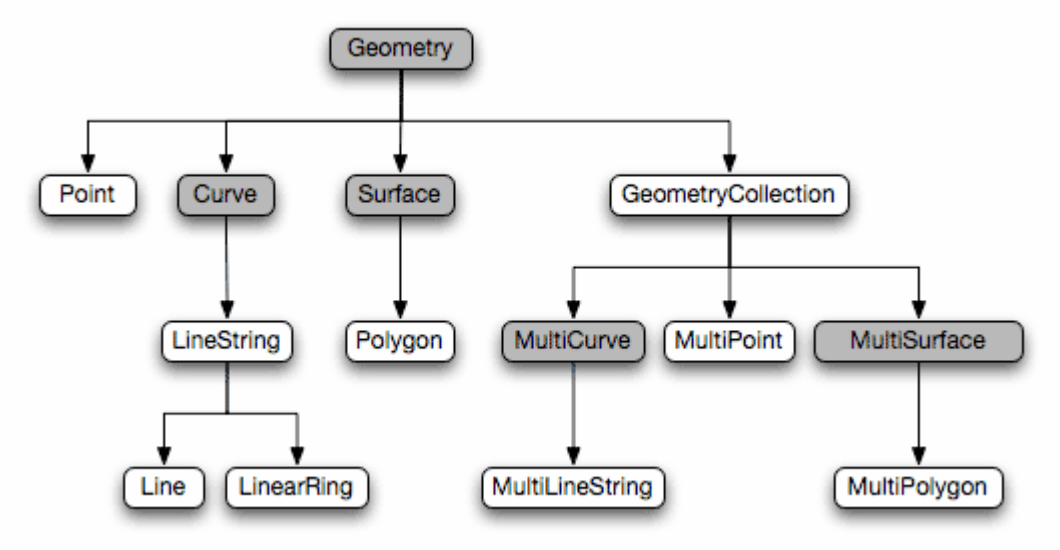
Do gromadzenia, przetwarzania oraz wizualizacji danych geograficznych (przestrzennych) służy System Informacji Geograficznej, w skrócie GIS.

## 4.2. Specyfikacja OpenGIS dla SQL-a

Specyfikacja OpenGIS definiuje rozszerzenia dla relacyjnych baz danych, które umożliwiają operacje na obiektach GIS-owych. Operacje na tych obiektach wykonywane są przy pomocy zwykłych zapytań SQLa.

Specyfikacja ta określa:

- Typy danych – typy mogące przechowywać informacje geograficzne, czyli wielowymiarowe, takim typem jest np. punkt czy linia (cała hierarchia typów danych zdefiniowanych przez OpenGIS na Rysunku 6.)
- Operacje – dodatkowe operacje wspierające obiekty przestrzenne, np. funkcja obliczająca powierzchnię wielokąta dowolnego kształtu
- Indeksowanie – dane przestrzenne indeksowane są przy użyciu tzw. R-drzew, które umożliwiają wyszukiwanie obiektów niepunktowych
- Formę reprezentacji, czyli sposób przechowywania obiektów GIS-owych w plikach binarnych i tekstowych




Rysunek 2. Hierarchia geometrycznych typów danych definiowanych przez specyfikacje OpenGIS
[źródło:www.opengeospatial.org]

Typy zaznaczone kolorem szarym na powyższym rysunku są typami abstrakcyjnymi, czyli nie można stworzyć obiektu tego typu.

Specyfikacja dołącza także definicję dwóch formatów dla zewnętrznej reprezentacji danych przestrzennych. Są to: WKB (Well-Known Binary) i WKT (Well-Known Text). Pozwala to na importowanie i eksportowanie danych do formatów binarnych i tekstowych.

## 4.3. Model danych przestrzennych

### 4.3.1. Obiekty przestrzenne

Przestrzenne bazy danych przechowują zwykle reprezentacje obiektów fizycznych. Zapamiętanie w bazie faktycznego odwzorowania obiektu nie jest możliwe, dlatego musi zostać on zredukowany do abstrakcyjnej postaci. Reprezentacją takich obiektów w bazie danych są przeważnie punkty, linie i wielokąty (regiony).

**Punkt**

Jest najprostszą reprezentacją przestrzenną, określa go położenie w przestrzeni, nie zawiera on informacji o kształcie czy rozmiarze obiektu, który reprezentuje. Ponieważ operacje na punktach są stosunkowo proste, wybierane są one do reprezentowania obiektów, których jedyną ważną cechą jest położenie.

**Linia**

Jest obiektem posiadającym kształt, ale nieposiadającym powierzchni. Nadaje się do reprezentowania obiektów, dla których ważna jest długość np. drogi, rzeki, linie energetyczne, wodociągi itp.

**Wielokąt**

Jest najbardziej złożoną reprezentacją obiektu przestrzennego. Charakteryzuje go położenie, kształt i powierzchnia. Wielokąty wykorzystuje się do reprezentowania wszelkich powierzchni jak np. miasta, jeziora, parki itp.

### 4.3.2. Relacje i operacje na obiektach przestrzennych

Relacje i operacje na obiektach przestrzennych dzielimy na trzy grupy:

- topologiczne (określają wzajemne położenie obiektów)
- kierunkowe (określają umiejscowienie obiektów w układzie współrzędnych)
- metryczne (określają odległości i rozmiary obiektów)

Do relacji topologicznych zaliczamy:

- zawieranie (gdy jeden obiekt zawiera się całkowicie w innym, np. rzeki reprezentowane przez linie zawarte w zlewni reprezentowanej przez wielokąt)
- przyleganie (gdy obiekty graniczą ze sobą, ale nie zachodzą na siebie np. dzielnice miasta reprezentowane przez wielokąty przylegające do siebie)
- łączność (gdy obiekty liniowe łączą się ze sobą tworząc sieci np. drogi, rzeki)

Przykładowe operacje topologiczne to:

- przekrój
- suma

Wśród relacji metrycznych wyróżnić można:

- odległość (określa metryczny dystans między 2 obiektami)
- długość obiektu (np. długość linii reprezentującej drogę czy rzekę)
- pole powierzchni (relacja możliwa jedynie dla wielokątów, pozwalająca określić powierzchnię jakiegoś terenu reprezentowanego przez wielokąt)

Przykładowe operacje metryczne to:

- szukanie obiektów oddalonych o określoną odległość np. mniejszą lub większą od n kilometrów (range query)
- szukanie najbliższego obiektu (nearest-neighbour)
- szukanie k najbliższych obiektów (k-nearest-neighbor)

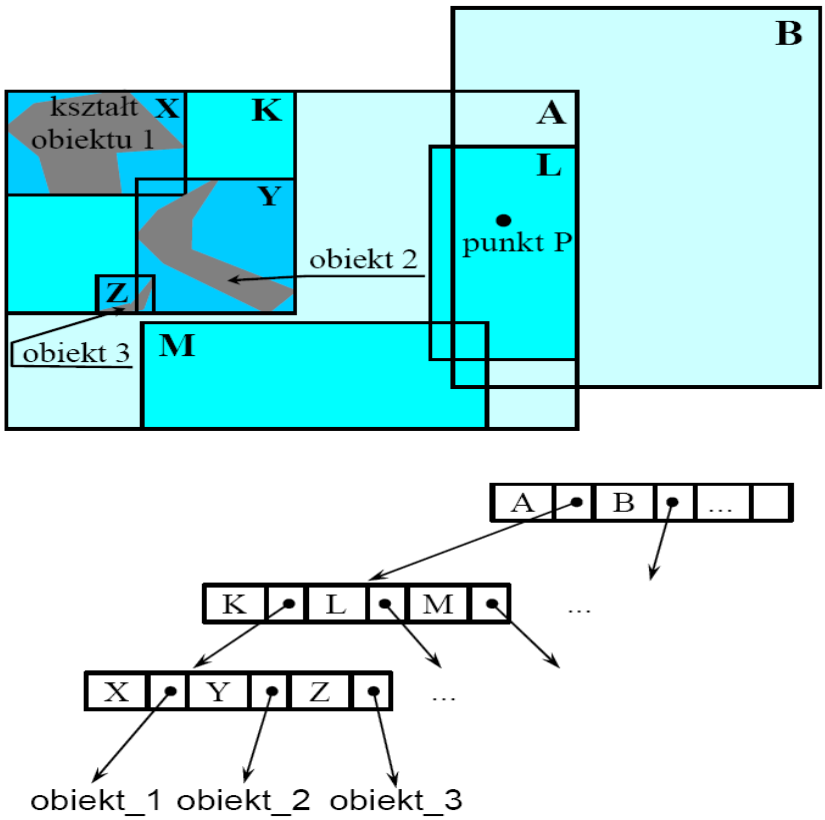
Relacja kierunkowa to np. „leży na północ”, a operacja kierunkowa to „szukanie obiektów leżących na północ od danego obiektu”.

### 4.3.3. Indeksowanie danych przestrzennych (R-drzewa)

Dane przestrzenne wymagają specjalnych indeksów, dlatego w przestrzennych bazach danych stosuje się indeksowanie za pomocą tzw. R-drzew.

R-drzewa są dynamicznymi strukturami danych służącymi do wyszukiwania obiektów wielowymiarowych w przestrzeni wielowymiarowej. W R-drzewach obiekty wielowymiarowe aproksymowane są za pomocą minimalnych regionów pokrywających MBR (minimum bounding rectangle).

Na poniższym rysunku przedstawiony został podział przestrzeni przez minimalne regiony pokrywające (MBR) oraz struktura R-drzewa powstała na podstawie MBR.




Rysunek 3. MBR i struktura R-drzewa
[źródło: hydrus.et.put.poznan.pl]

Charakterystyka R-drzew [hydrus.et.put.poznan.pl]:

- jest drzewem zrównoważonym tzn. wszystkie liście znajdują się na tym samym poziomie drzewa
- wszystkie obiekty znajdują się w liściach
- regiony znajdujące się w tym samym węźle drzewa mogą na siebie zachodzić
- MBR syna zawiera się w MBR ojca
- suma wszystkich regionów znajdujących się w danym węźle nie musi tworzyć regionu i w konsekwencji nie musi być równa zawierającemu je regionowi rodzicielskiemu

R-drzewa realizują następujące funkcje:

- zapytania punktowe (znajdź identyfikatory obiektów przestrzennych, których MBR zawiera punkt P)
- zapytania regionowe (znajdź identyfikatory obiektów przestrzennych, których MBR ma część wspólną z regionem R)
- najbliższy sąsiad (znajdź identyfikatory obiektów przestrzennych, których MBR są najmniej oddalone od punktu P)

Wyszukiwanie danych za pomocą R-drzew nie jest zbyt szybkie, szczególnie, gdy stopień nachodzenia na siebie minimalnych regionów pokrywających jest duży. Istnieje także silna zależność efektywności wyszukiwania od gęstości danych oraz liczby wymiarów.

## 4.4. SpatiaLite – rozszerzenie SQLite'a do składowanie danych przestrzennych

SpatiaLite jest rozszerzeniem bazy SQLite umożliwiającym składowanie i przetwarzanie danych przestrzennych. SpatiaLite tworzony jest przez włoską organizację Gaia-GIS i udostępniany jest na licencji Open Source.

### 4.4.1. Funkcje oferowane przez SpatiaLite

SpatiaLite wspiera następujące aspekty ze specyfikacji OpenGIS:

- wszystkie przestrzenne typy obiektów m.in. POINT, LINE, POLYGON itp.
- reprezentacje obiektów WKB (Well-Known Binary) i WKT (Well-Known Text)
- funkcje konwertujące takie jak AsText(), GeomFromText()
- przestrzenne funkcje do analizy danych takie jak Area(), Length(), Distance()
- predykaty binarne odpowiedzialne za sprawdzanie zależności pomiędzy dwoma obiektami np. Contains(), Overlaps(), Touches()
- indeksowanie z użyciem R-drzew

**Typy obiektów, jakie można tworzyć w SpatiaLite**

- **Point** – reprezentuje pojedynczą lokalizację, określają go współrzędne X i Y
- **LineString –** reprezentuje linię**,** jest liniową interpolacją między punktami
- **Line –** reprezentuje linię prostą (złożoną z dokładnie dwóch punktów)
- **LinearRing –** reprezentuje linię zamkniętą
- **Polygon –** reprezentuje wielokąt otoczony jedną granicą zewnętrzną, może zawierać także granice wewnętrzne (oznacza to, że powierzchnia zawiera wycięcia wewnątrz)
- **GeometryCollection –** określa obiekt, będący zbiorem obiektów geometrycznych dowolnego typu
- **MultiPoint –** reprezentuje zbiór punktów
- **MultiLineString –** reprezentuje zbiór linii
- **MultiPolygon –** typ reprezentujący obiekt złożony z wielokątów

SpatiaLite obsługuje dwa popularne formaty dla zewnętrznej reprezentacji danych przestrzennych: WKB (Well-Known Binary) i WKT (Well-Known Text).

WKT służy do wymiany danych w formacie ASCII. Np. punkt określany jest w postaci formuły POINT(100.00 100.00), linia w postaci LINESTRING(100.0 100.0, 200.0 150.0, 250.0 400.0).

WKB służy do wymiany danych w postaci binarnej. Np. punkt POINT(1 1) jest reprezentowany przez 21 bajtów (każdy reprezentowany przez dwie cyfry szesnastkowe):
01 01000000 000000000000F03F 000000000000F03F
gdzie:
Bajt kolejności: 01
Typ WKB: 01000000
Współrzędna X: 000000000000F03F
Współrzędna Y: 000000000000F03F

Do wymiany danych w formatach WKT i WKB służą odpowiednio funkcje: GeomFromText(wkt) oraz GeomFromWKB(wkb).

**Funkcje przestrzenne do analizy danych dostępne w SpatiaLite dla typu Geometry**

- Dimension(geom) – zwraca wymiar wartości podanej jako argument, wynikiem może być -1, 0, 1 lub 2, gdzie -1 oznacza pustą geometrię, 0 – punkt, 1- linię, 2 – wielokąt
- Envelope(geom) – zwraca MBR dla wartości geometrii w postaci wielokąta (wartość typu Polygon)
- GeometryType(geom) – zwraca nazwę typu geometrii
- IsEmpty(geom) – zwraca 1, jeśli wartość geometrii jest pusta oraz 0, jeśli nie jest pusta

**Funkcje przestrzenne dla typu Point**

- X(pt) – zwraca współrzędną X punktu
- Y(pt) – zwraca współrzędną Y punktu

**Funkcje przestrzenne dla typu LineString**

- EndPoint(line) – zwraca punkt będąc punktem końcowym linii
- IsRing(line) – zwraca 1, jeśli wartość line jest linią zamkniętą (okręgiem) oraz 0 jeśli nie jest okręgiem
- GLength(line) – zwraca długość linii
- NumPoints(line) – zwraca liczbę punktów zawartych w linii
- PointN(line, N) – zwraca n-ty punkt linii (punkty numerowane są od 1)
- StartPoint(line) – zwraca punkt będący punktem początkowym linii

**Funkcje przestrzenne dla typu Poligon**

- Area(polyg) – zwraca powierzchnię wielokąta
- Centroi(polyg) – zwraca środek ciężkości wielokąta w postaci punktu
- ExteriorRing(polyg) – zwraca zewnętrzny okrąg opisany na wielokącie w postaci linii
- InteriorRingN(polyg, N) – zwraca n-ty wewnętrzny okrąg dla wielokąta
- NumInteriorRings(polyg) – zwraca liczbę wewnętrznych okręgów dla wielokąta

### 4.4.2. Przykłady operacji na bazie zawierającej dane przestrzenne

SpatiaLite w celu przechowywania wszelkiego rodzaju geometrii wymaga standardowej kolumny SQLite'a typu BLOB. Tabelę z taką kolumną tworzymy w standardowy sposób:

*CREATE TABLE geo_table (....., geo_data BLOB NOT NULL);*

Do tak stworzonej tabeli dane możemy wstawić w dwojaki sposób, korzystając z formatu WKT lub WKB. Oczywiście format WKT wydaje się znacznie bardziej przyjazny dla użytkownika.

*INSERT INTO geo_table (...., geo_data) VALUES (...., GeomFromText('POINT(1 1)'));*

*INSERT INTO geo_table (...., geo_data) VALUES (...., GeomFromWKB(X'0x0101000000000000000000F03F000000000000F03F'));*

Geometryczne wartości przechowywane w bazie mogą być pobrane w formacie BLOB lub przekonwertowane na formaty WKT lub WKB.

Pobieranie danych w formacie BLOB (niezalecane, ponieważ wyświetla na ekranie mało czytelne dane binarne):

*SELECT Hex(geo_data) FROM geo_table;*

(Polecenie *SELECT geo_data FROM geo_table;* nie daje żadnego rezultatu)

Pobieranie danych w formacie WKT:

*SELECT AsText (geo_data) FROM geo_table;*

Funkcja AsText() konwertuje dane do formatu WKT.

Pobieranie danych w formacie WKB:

*SELECT AsBinary (geo_data) FROM geo_table;*

Funkcja AsBinary() konwertuje dane do formatu WKB, który nie jest zbyt czytelny dla użytkowników.

## 4.5. Dane przestrzenne w innych systemach bazodanowych

Pozostałe popularne darmowe systemy bazodanowe również wspierają w mniejszym bądź większym stopniu składowanie danych przestrzennych.

**MySQL**

Od wersji MySQL 4.1 wprowadzone zostały w tym systemie mechanizmy pozwalające na składowanie i analizę danych przestrzennych zgodne ze specyfikacją

OpenGIS. Zdefiniowano tam hierarchę klas geometrycznych (Geometry) stosowaną do opisu obiektów przestrzennych. Dane przestrzenne mogą być składowane tylko w tabeli typu MyISAM, która umożliwia indeksowanie przestrzenne oparte o R-drzewa. Podobnie jak rozszerzenie do SQLite'a, MySQL dostarcza jak na razie jedynie funkcje do rejestrowania obiektów przestrzennych oraz kilkunastu funkcji do elementarnych obliczeń przestrzennych. Brak natomiast zaawansowanych funkcji do analiz przestrzennych jak np. znajdowanie najkrótszej drogi.

**PostgreSQL**

Baza PostgreSQL posiada specjalny dodatek do obsługi danych przestrzennych o nazwie PostGIS. Rozpowszechniany jest on na licencji GNU GPL przez Refractions Research. PostGIS posiada pełną zgodność ze specyfikacją OpenGIS. Jest znacznie bardziej zaawansowany od rozszerzeń przestrzennych w SQLite czy MySQL. Poza standardowymi typami danych jak punkty, linie czy wielokąty, PostGIS posiada wsparcie dla zewnętrznych typów danych takich jak: grafy, dane rastrowe, dane 3D, splajny. Dane takie są pobierane z zewnętrznych plików (np. w formatach Shape, MapInfo, DGN, GML), następnie konwertowane i wstawiane do bazy. Dane z PostGIS-a mogą być także użyte jako dane źródłowe dla oprogramowania takiego jak MapServer czy GeoServer.

**Firebird**

Nie wspiera składowania danych przestrzennych. Brak zaimplementowanego w bazie indeksowania przestrzennego (indeksy R-tree).

# 5.

# PHP jako interfejs dostępu do bazy SQLite

## 5.1. Interfejs strukturalny

**Tworzenie tabel**

SQLite jest łatwy do uruchomienia, nie ma potrzeby konfiguracji zmiennych, takich jak nazwa serwera, nazwa użytkownika czy hasło. Jedynie, co jest potrzebne to nazwa pliku, gdzie przechowywane będą dane:

```
$db = sqlite_open('/www/users.db');
sqlite_query($db, 'CREATE TABLE users(username VARCHAR(100),
password VARCHAR(100))');
```

Powyższe polecenie tworzy tabelę „users", przechowywaną w pliku bazy danych o ścieżce dostępu /www/users.db. Jeżeli użytkownik próbuje otworzyć nieistniejący plik to SQLite automatycznie go tworzy. Nie trzeba używać żadnych komend do tworzenia nowej bazy.

**Wstawianie danych do bazy**

Aby dodać nowy rekord do tabeli używamy SQL-owego polecenia INSERT oraz funkcji sqlite_query().

```
$username = sqlite_escape_string($username);
$password = sqlite_escape_string($password);
sqlite_query($db, "INSERT INTO users VALUES ('$username', '$password')");
```

Funkcji sqlite_escape_string() użyto, aby pozbyć się problemu z cudzysłowami, przy wstawianiu „stringów" do bazy.

**Odczytywanie danych z bazy**

W celu pobrania danych z bazy używamy funkcji *sqlite_query()* w połączeniu z SQL-ową komendą SELECT.

```
$r = sqlite_query($db, 'SELECT username FROM users');
while ($row = sqlite_fetch_array($r)) {
// operacje na $row
}
```

Funkcja *sqlite_fetch_array()* zwraca standardowo tablicę zawierającą zarówno klucze liczbowe jak i asocjacyjne. Przykładowo taka tablica dla zapytania zwracającego jeden wiersz wygląda tak:

```
Array (
[0] => paul
[username] => paul
)
```

Jak widać na przykładzie tablica ta zawiera klucz liczbowy [0] i asocjacyjny [username]. Użytkownik ma wybór, za pomocą którego klucza chce się odwoływać do danych w tablicy.

Domyślne zwracanie dwóch kluczy niesie jednak ze sobą pewien problem przy zagnieżdżonych pętlach, takich jak w przykładzie poniżej:

```
$r = sqlite_query($db, 'SELECT * FROM users');
while ($row = sqlite_fetch_array($r)) {
foreach ($row as $column) {
print "$column\n";
}
}
```

Problemem tym jest to, że dane z każdej kolumny zostaną wyświetlone dwukrotnie, najpierw dla klucza numerycznego, a później dla asocjacyjnego. Aby temu zapowiedz trzeba przekazać do funkcji dodatkowy argument:

SQLITE_NUM – przechowywane będą tylko klucze numeryczne

SQLITE_ASSOC- przechowywane będą tylko klucze asocjacyjne

SQLITE_BOTH – przechowywane będą oba typy kluczy (tak jak domyślnie)

Jeśli zależy nam na szybkości działania, to zamiast *sqlite_query()* i *sqlite_fetch_array() lepiej* użyć funkcji *sqlite_array_query()*, która w pojedynczym zapytaniu odczyta dane i umieści je w tablicy.

```
$r = sqlite_array_query($db, 'SELECT * FROM users');
foreach ($r as $row) {
```

```
// operacje na  $row
}
```

Gdy chcemy otrzymać liczbę rekordów zwróconych przez zapytanie stosujemy funkcję *sqlite_num_row(),* np. w ten sposób zapisując tę liczbę w zmiennej $count:

```
$count = sqlite_num_rows($r);
```

W celu zakończenia połączenia z bazą wywołujemy funkcję *sqlite_close($db).*

Osoby używające MySQL-a zapewne zauważyły podobieństwo nazw funkcji w MySQL i SQLite. Rzeczywiście są one podobne, ale nie identyczne. W tabeli poniżej znajduje się zestawienie nazw głównych funkcji z obu systemów bazodanowych.

Tabela 1. Porównanie nazw głównych funkcji do obsługi baz MySQL i SQLite udostępnianych przez PHP 5

| **MySQL** | **SQLite** |
|---|---|
| mysqli_connect() | sqlite_connect() |
| mysqli_close() | sqlite_close() |
| mysqli_query() | sqlite_query() |
| mysqli_fetch_row() | sqlite_fetch_array() |
| mysqli_fetch_assoc() | sqlite_fetch_array() |
| mysqli_num_rows() | sqlite_num_rows() |
| mysqli_insert_id() | sqlite_last_insert_rowid() |
| mysqli_real_escape_string() | sqlite_escape_string() |

## 5.2. Interfejs obiektowy

Rozszerzenie języka PHP umożliwia także obiektową interakcję z baza SQLite.

Obiektowo zorientowany interfejs zamienia połączenie z bazą na obiekt i umożliwia wywoływanie na nim odpowiednich metod.

Poniżej przykład użycia obiektowego SQLite (odczyt danych z bazy):

```
$db = new SQLiteDatabase('/www/support/users.db');

$r = $db->query('SELECT * FROM users');
while ($row = $r->fetch( )) {
// operacje na $row
        }
// i drugi sposób z użyciem foreach, który nie wymaga wywołania metody fetch()
$r = $db->arrayQuery('SELECT * FROM users');
foreach ($r as $row) {
// operacje na $row
    }
unset($db);
```

Wszystkie proceduralne funkcje SQLite mają swoje odpowiedniki w SQLite obiektowym. Tabela 2 zawiera najczęściej używane funkcje w SQLite wraz z ich obiektowymi odpowiednikami.

Tabela 2. Funkcje do obsługi bazy SQLite w PHP 5

| **Proceduralny SQLite** | **Obiektowy SQLite** |
|---|---|
| $db = sqlite_open(sciezka_do_pliku) | $db = new SQLiteDatabase(sciezka) |
| sqlite_close($db) | unset($db) |
| $r = sqlite_query($db, $sql) | $r = $db->query($sql) |
| $r = sqlite_query_array($db, $sql) | $r = $db->arrayQuery($sql) |
| $r = sqlite_query_unbuffered($db, $sql) | $r = $db-> unbufferedQuery($sql) |
| sqlite_fetch_array($r) | $r->fetch() |
| sqlite_fetch_single($r) | $r->fetchSingle() |
| $safe = sqlite_escape_string($s) | $safe = $db->escapeString($s) |
| $id = sqlite_last_insert_rowid($r) | $id = $db->lastInsertRowid($r) |

## 5.3. Definiowanie własnych funkcji

Ciekawą i przydatną opcją SQLite jest możliwość tworzenia własnych funkcji w PHP rozszerzających możliwości bazy danych. Użytkownik może tworzyć dwa rodzaje funkcji: standardowe i agregacyjne.

Funkcje standardowe to tzw. funkcje jeden-do-jednego, czyli funkcja dostaje jako argument jeden rekord i zwraca jeden rekord.

Funkcje agregacyjne (wiele-do-jednego) jako argument dostają wiele rekordów, a jako wynik zwracają pojedynczą wartość. Przykładem takiej funkcji agregacyjnej zdefiniowanej w SQL-u jest funkcja *count()*, która zwraca liczbę rekordów wysłanych do niej.

Stworzenie SQLite'owej funkcji sprowadza się do wywołania funkcji *sqlite_create_function()* i przesłania do niej wcześniej stworzonej funkcji w PHP.

```
//tworzenie funkcji w php
function nazwa_php ($argument){
```

*//ciało funkcji*
*}*

//tworzenie funkcji SQLite użytkownika poprzez przesłanie funkcji php
*sqlite_create_function($db, 'nazwa_sql', 'nazwa_php');*

Tak stworzoną funkcję wykorzystujemy w zapytaniach SELECT, odwołując się do niej przez jej nazwę (z powyższego przykładu: nazwa_sql) i jako argument wysyłając do niej nazwę pola tabeli.

## 5.4. Obsługa błędów

Podobnie jak inne systemy bazodanowe, także SQLite posiada własny mechanizm obsługi błędów. W interfejsie dla języka PHP dostęp do komunikatów o błędach otrzymujemy dzięki funkcjom *sqlite_last_error()* i *sqlite_error_string()*. W przypadku niepowodzenia podczas wykonywania jakiejś czynności SQLite przypisuje kod błędu funkcji *sqlite_last_error()*, które zwraca numer ostatnio wygenerowanego przez SQLite błędu. Oczywiście sam numer błędu za wiele nam nie powie, więc trzeba go przekształcić w jakiś bardziej zrozumiały komunikat i do tego właśnie służy funkcja *sqlite_error_string()*. Zamienia ona numer błędu na opisową informację o błędzie.

Przykład:

```
if( sqlite_query($db, $sql) )
{
  // operacje
}
else
die( sqlite_error_string( sqlite_last_error($db) ) );
```

Nieco inaczej błąd zwracany jest w przypadku funkcji *sqlite_open()*. W przypadku niepowodzenia informacje o błędzie przypisywane są zmiennej, będącej trzecim argumentem funkcji. Drugi argument funkcji określa prawa dostępu do pliku bazy.

*sqlite_open('/sciezka /baza.db',0666,$sqlite_error);*

# 6.

# Porównanie wydajności SQLite'a z bazami MySQL, PostgreSQL i Firebird

## 6.1. Charakterystyka pomiarów

### 6.1.1. Środowisko testowe

Testy porównujące wydajność bazy SQLite z bazami MySQL, PostgreSQL i Firebird wykonano na komputerze klasy PC o parametrach technicznych określonych w tabeli 3 i lokalnie zainstalowanym serwerze Apache.

Tabela 3. Specyfikacja środowiska testowego

| Komponent | Typ/Wersja |
|---|---|
| Procesor | Athlon 1800+ (1.53 MHz) |
| Pamięć RAM | DDR 1GB |
| System operacyjny | Windows XP SP2 |
| Serwer | Apache 2.2.9 |
| Język skryptowy | PHP 5.2.6 |

W czasie testów porównano wydajność bazy SQLite z trzema darmowymi systemami bazodanowym, mianowicie MySQL, PostgreSQL i Firebird.
W tabeli 4 podano wersje użytych systemów.

Tabela 4. Systemy bazodanowe użyte w testach

| Nazwa bazy | Wersja |
|---|---|
| SQLite | 2.8.17 |
| MySQL | 5.0.51 |
| PostgreSQL | 8.3.1 |
| Firebird | 2.1.1 |

### 6.1.2. Sposób przeprowadzenia pomiarów wydajności

Testy wydajności silników bazodanowych przeprowadzone zostały przy pomocy skryptów napisanych w języku PHP.

Miały one na celu sprawdzenie szybkości wykonywania przez silniki bazodanowe operacji zapisu danych do bazy i odczytu danych z bazy. Testom poddano dwa główne zapytania SQL-owe INSERT i SELECT.

Do pomiaru czasu wykonania zapytań wykorzystano funkcję microtime(), zwracającą czas w mikrosekundach od tzw. ery unixa. Za pomocą tej funkcji zmierzono czas rozpoczęcia zapytania i czas zakończenia, a następnie wyliczono różnicę będącą czasem wykonania zapytania. W funkcji microtime() użyto argumentu TRUE, dzięki czemu wynik jest liczbą zmiennoprzecinkową określającą sekundy.

W testach użyto standardowej konfiguracji systemów bazodanowych.

## 6.2. Test wydajności zapisu danych do bazy (INSERT)

Testy wydajności zapisu danych przeprowadzono w dwóch wersjach tj.:

- wstawianie danych do tabeli (100 i 10000 instrukcji INSERT)
- wstawianie danych do tabeli z użyciem transakcji (1000 i 10000 instrukcji INSERT)

Zostały one wykonane na tabeli o nazwie test1 składającej się z trzech kolumn: id (typu INTEGER), osoba (typu VARCHAR(50)) i placa (typu INTEGER).

**TEST 1**

Test 1 polegał na wstawieniu do tabeli 100 i 10000 rekordów za pomocą instrukcji

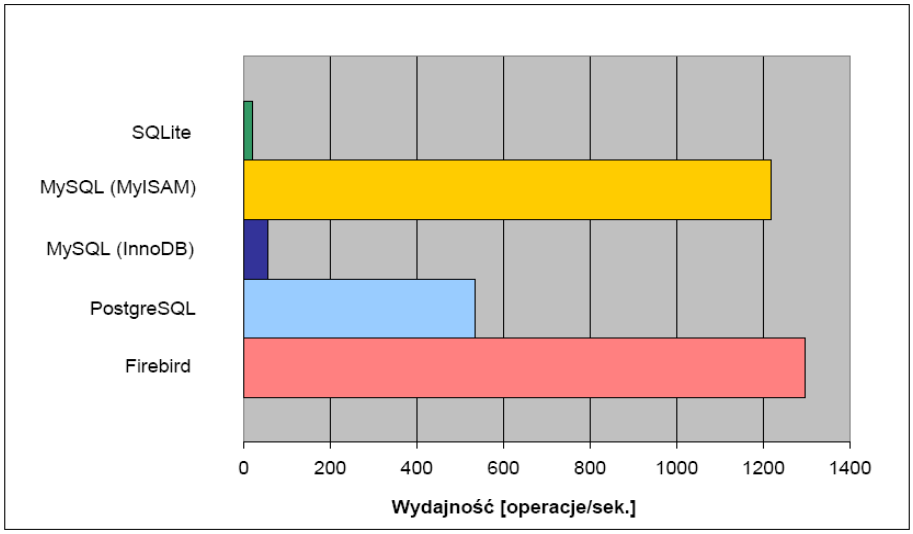
*INSERT INTO test1 VALUES (‘$id’, ‘$osoba’, ‘$placa’);* wykonanej w pętli while.

Tabela 5. Czasy wykonania 100 i 10000 operacji INSERT

| Baza | Czas [s] | |
|---|---|---|
| | 100 INSERT | 10000 INSERT |
| SQLite | 5.179 | 508.725 |
| MySQL (MyISAM) | 0.082 | 7.287 |
| MySQL (InnoDB) | 1.866 | 204.144 |
| PostgreSQL | 0.187 | 16.492 |
| Firebird | 0.077 | 6.324 |

Tabela 6. Wydajność zapisu danych (100 i 10000 operacji INSERT)

| Baza | Wydajność [operacje / sek.] | |
|---|---|---|
| | 100 INSERT | 10000 INSERT |
| SQLite | 19 | 19 |
| MySQL (MyISAM) | 1219 | 1373 |
| MySQL (InnoDB) | 55 | 48 |
| PostgreSQL | 534 | 606 |
| Firebird | 1298 | 1581 |




Rysunek 4. Wydajność zapisu danych (100 operacji INSERT)
(im wyższa wydajność tym lepiej)

Otrzymane wyniki znacznie różnią się od siebie, co wynika z różnych typów tabel oraz sposobu przeprowadzenia operacji w poszczególnych bazach. Przy wstawianiu danych do bazy bardzo słabo wypadł SQLite i MySQL z tabelą InnoDB – odstają one zdecydowanie pod względem wydajności od pozostałych baz. Spowodowane jest to tym, że SQLite (podobnie jak tabela InnoDB MySQL-a) korzysta domyślnie z transakcji. Każde zapytanie w takim przypadku traktowane jest jako oddzielna transakcja, co wymaga najpierw zapisania danych do cache-u, utworzenia logów dla każdego z zapytań i dopiero wtedy zapisania danych do bazy na dysku. Dla każdego zapytania musi także nastąpić otwarcie i zamknięcie pliku bazy. Zabiegi takie drastycznie wydłużają czas wykonania zapytań. Najszybsze pod względem wstawiania rekordów do bazy okazały się MySQL korzystający z tabel MyISAM (nieobsługujący transakcji) oraz Firebird.

**TEST 2**

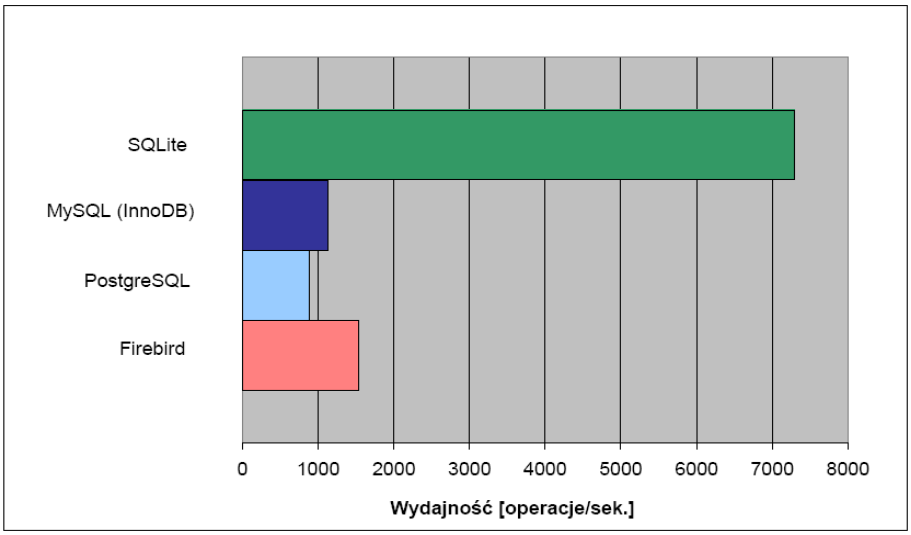
Test 2 polegał na wstawieniu do tabeli 1000 i 10000 rekordów w operacji transakcyjnej za pomocą instrukcji
*INSERT INTO test1 VALUES (‘$id’, ‘$osoba’, ‘$placa’);* wykonanej w pętli while.
Przed pętlą while znajdowała się instrukcja BEGIN rozpoczynająca transakcję, a za pętlą instrukcja COMMIT zatwierdzająca transakcję.

Tabela 7. Czasy wykonania 1000 i 10000 operacji INSERT zawartych w transakcji

| **Baza** | **Czas [s]** | |
|---|---|---|
| | **100 INSERT w transakcji** | **10000 INSERT w transakcji** |
| SQLite | 0.137 | 1.658 |
| MySQL (InnoDB) | 0.887 | 7.896 |
| PostgreSQL | 1.138 | 10.514 |
| Firebird | 0.605 | 5.802 |

Tabela 8. Wydajność zapisu danych (1000 i 10000 operacji INSERT w transakcji)

| Baza | Wydajność [operacje/sek.] | |
|---|---|---|
| | 1000 INSERT w transakcji | 10000 INSERT w transakcji |
| SQLite | 7299 | 6031 |
| MySQL (InnoDB) | 1127 | 1266 |
| PostgreSQL | 879 | 946 |
| Firebird | 1534 | 2083 |




Rysunek 5. Wydajność zapisu danych (1000 operacji INSERT w transakcji) (im wyższa wydajność tym lepiej)

Zgrupowanie zapytań w pojedynczą transakcje znacznie skraca czas ich wykonania dla baz korzystających domyślnie z transakcji. Największą różnicę pomiędzy wykonywaniem zapytań pojedynczo i zawartych w jednej transakcji widać na przykładzie SQLite'a. Na wzrost wydajności główny wpływ miał brak konieczności otwierania i zamykania pliku dla każdego zapytania. Plik otwierany jest na początku transakcji (po poleceniu BEGIN) i zamykany po poleceniu COMMIT zatwierdzającym transakcje. SQLite uzyskał zdecydowanie najlepszą wydajność przy transakcyjnym wstawianiu danych do bazy spośród testowanych baz.

## 6.3. Test wydajności odczytu danych z bazy (SELECT)

Testy wydajności odczytu danych przeprowadzono w dwóch wersjach:

- pobieranie danych z tabeli (100 operacji SELECT dla baz zawierających 5 tys. i 100 tys. rekordów)
- pobieranie danych z porównywaniem łańcuchów znakowych (100 operacji SELECT dla baz zawierających 5 tys. i 100 tys. rekordów)

**TEST 3**

Test 3 polegał na wyszukaniu w tabeli danych spełniających podane kryterium wyszukiwania. Zapytanie wykorzystane w teście to:

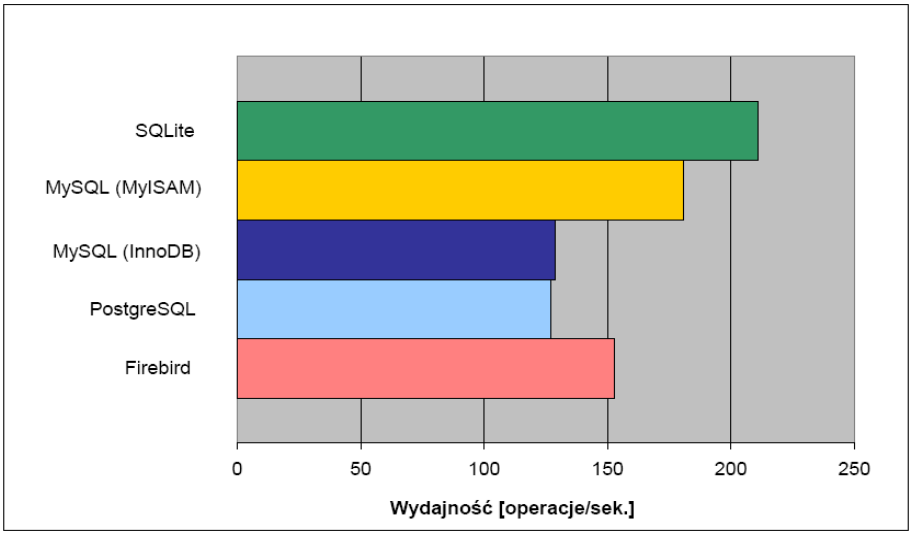
*SELECT * FROM test1 WHERE placa>3000 AND placa<6000;*

Tabela 9. Czasy wykonania 100 operacji SELECT

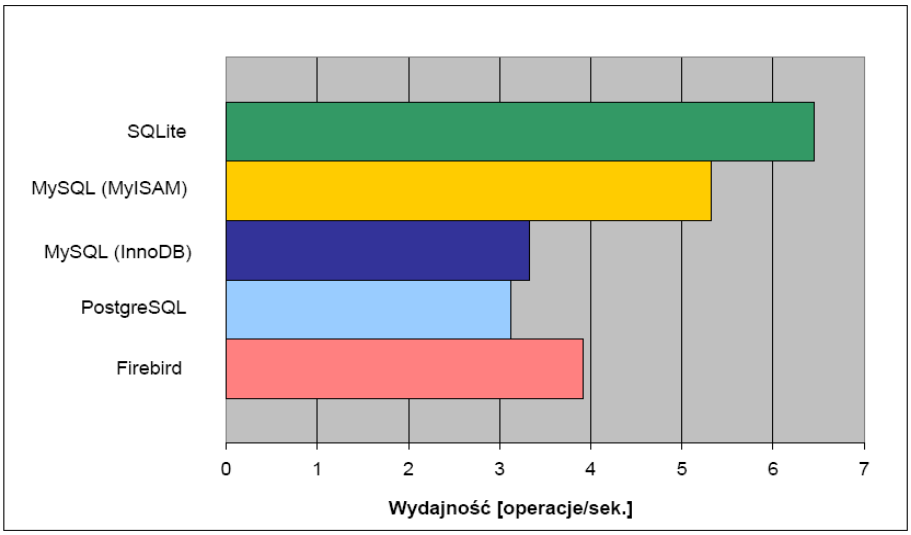
| **Baza** | **Czas [s]** | |
|---|---|---|
| | **100 SELECT baza 5 tys. rekordów** | **100 SELECT baza 100 tys. rekordów** |
| SQLite | 0.473 | 15.495 |
| MySQL (MyISAM) | 0.551 | 18.832 |
| MySQL (InnoDB) | 0.773 | 30.059 |
| PostgreSQL | 0.783 | 32.063 |
| Firebird | 0.652 | 25.514 |

Tabela 10. Wydajność odczytu danych (100 operacji SELECT)

| Baza | Wydajność [operacje/sek.] | |
|---|---|---|
| | **100 SELECT baza 5 tys. rekordów** | **100 SELECT baza 100 tys. rekordów** |
| SQLite | 211 | 6.45 |
| MySQL (MyISAM) | 181 | 5.32 |
| MySQL (InnoDB) | 129 | 3.33 |
| PostgreSQL | 127 | 3.12 |
| Firebird | 153 | 3.92 |




Rysunek 6. Wydajność odczytu danych (100 operacji SELECT na bazie liczącej 5 tys. rekordów)
(im wyższa wydajność tym lepiej)




Rysunek 7. Wydajność odczytu danych (100 operacji SELECT na bazie liczącej 100 tys. rekordów)
(im wyższa wydajność tym lepiej)

Z wyszukiwaniem i pobieraniem danych z bazy, SQLite radzi sobie znacznie lepiej niż z ich zapisem. Na tle innych baz wypada można powiedzieć rewelacyjnie, bo okazuje się lepszy od uznawanego za bardzo szybki mechanizmu MyISAM bazy MySQL.

**TEST 4**

Test 4 polegał na wyszukaniu w tabeli indeksowanej rekordów posiadających w kolumnie “osoba” frazę podaną w zapytaniu. Zapytanie wyglądało następująco:

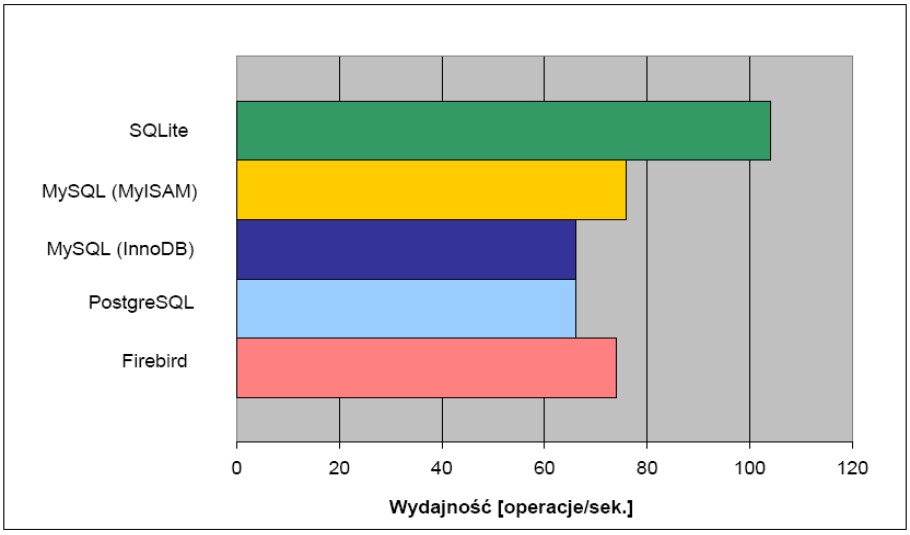
*SELECT * FROM test1 WHERE osoba LIKE “%kowalski%”;*

Tabela 11. Czasy wykonania 100 operacji SELECT z porównywaniem łańcuchów znakowych

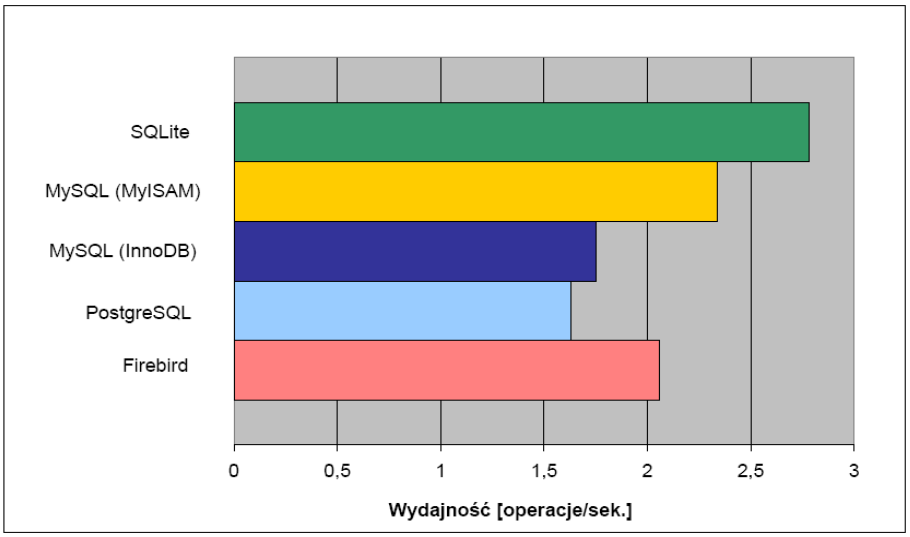
| **Baza** | **Czas [s]** | |
|---|---|---|
| | **100 SELECT baza 5 tys. rekordów** | **100 SELECT baza 100 tys. rekordów** |
| SQLite | 0.958 | 35.985 |
| MySQL (MyISAM) | 1.319 | 42.635 |
| MySQL (InnoDB) | 1.502 | 57.122 |
| PostgreSQL | 1.510 | 61.320 |
| Firebird | 1.339 | 48.554 |

Tabela 12. Wydajność zapisu danych (100 operacji SELECT z porównywaniem łańcuchów znakowych)

| **Baza** | **Wydajność [operacje/sek.]** | |
|---|---|---|
| | **100 SELECT baza 5 tys. rekordów** | **100 SELECT baza 100 tys. rekordów** |
| SQLite | 104 | 2.78 |
| MySQL (MyISAM) | 76 | 2.34 |
| MySQL (InnoDB) | 66 | 1.75 |
| PostgreSQL | 66 | 1.63 |
| Firebird | 74 | 2.06 |




Rysunek 8. Wydajność odczytu danych (100 operacji SELECT z porównywaniem łańcuchów znakowych na bazie liczącej 5 tys. rekordów) (im wyższa wydajność tym lepiej)




Rysunek 9. Wydajność odczytu danych (100 operacji SELECT z porównywaniem łańcuchów znakowych na bazie liczącej 100 tys. rekordów) (im wyższa wydajność tym lepiej)

Przy wyszukiwaniu rekordów z zastosowaniem porównywania łańcuchów znakowych podobnie jak przy wyszukiwaniu z porównywaniem numerycznym najszybszą bazą okazał się SQLite. Wydajność wszystkich baz była ok. dwa razy mniejsza niż przy wyszukiwaniu danych z użyciem porównywania numerycznego.

## 6.4. Test wydajności operacji zmiany danych w bazie (UPDATE)

**TEST 5**

Test 5 polegał na edycji danych w bazie poprzez wpisanie w polu „placa" wartości 3000 dla wszystkich rekordów, gdzie placa < 5000. Zapytanie wykorzystane w teście wyglądało następująco:

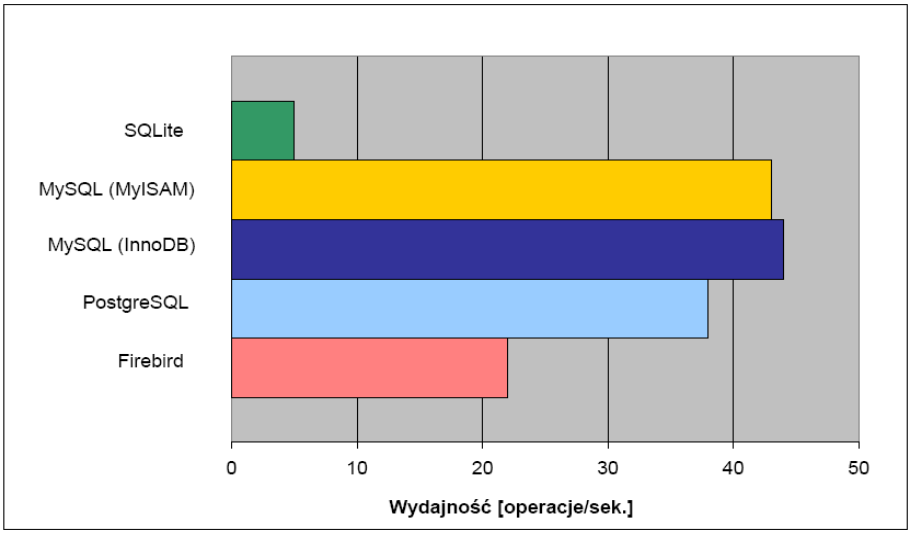
*UPDATE test1 SET placa=3000 WHERE placa<5000;*

Tabela 13. Czasy wykonania 100 operacji UPDATE

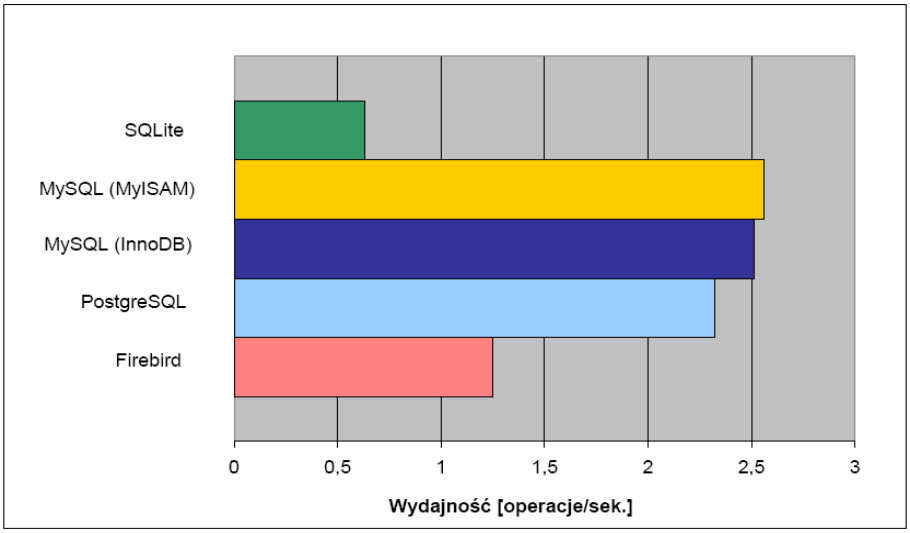
| Baza | Czas [s] | |
|---|---|---|
| | **100 UPDATE baza 5 tys. rekordów** | **100 UPDATE baza 100 tys. rekordów** |
| SQLite | 17.824 | 157.166 |
| MySQL (MyISAM) | 2.312 | 39.175 |
| MySQL (InnoDB) | 2.225 | 39.834 |
| PostgreSQL | 2.633 | 43.153 |
| Firebird | 4.519 | 80.094 |

Tabela 14. Wydajność wykonania 100 operacji UPDATE

| Baza | Wydajność [operacje/sek.] | |
|---|---|---|
| | **100 UPDATE baza 5 tyś. rekordów** | **100 UPDATE baza 100 tyś. rekordów** |
| SQLite | 5 | 0.63 |
| MySQL (MyISAM) | 43 | 2.56 |
| MySQL (InnoDB) | 44 | 2.51 |
| PostgreSQL | 38 | 2.32 |
| Firebird | 22 | 1.25 |




Rysunek 10. Wydajność wykonania operacji UPDATE na bazie liczącej 5 tys. rekordów
(im wyższa wydajność tym lepiej)




Rysunek 11. Wydajność wykonania operacji UPDATE na bazie liczącej 100 tys. rekordów
(im wyższa wydajność tym lepiej)

Operacja zmiany danych w bazie (UPDATE) jest połączeniem operacji odczytu i zapisu danych, najpierw konkretne dane musza być znalezione w bazie, a następnie nadpisane. SQLite w powyższym teście (podobnie jak przy operacji INSERT) wypadł

znacznie słabiej od konkurentów. Główny powód taki sam jak przy operacji zapisu danych, czyli traktowanie każdego zapytania jako pojedynczej transakcji i co za tym idzie każdorazowe otwieranie i blokowanie pliku do zapisu danych.

## 6.5. Podsumowanie testów

Baza SQLite wypada słabo, jeśli chodzi o zapis informacji. Jest zdecydowanie najwolniejsza spośród testowanych baz. Główny powód takiego stanu rzeczy to traktowanie pojedynczego zapytania jako transakcję oraz konieczność każdorazowego otwierania i zamykania pliku dyskowego, co jest bardzo wolną operacją. Jeżeli zapytania INSERT zgrupujemy i wykonamy w jednej transakcji to SQLite staje się znacznie bardziej wydajny, bo operacje otwierania i zamykania pliku wykonywane są tylko raz.

Jeśli chodzi o odczyt danych, czyli wyszukiwanie i pobieranie informacji z bazy to SQLite radzi sobie z tym znacznie lepiej niż z zapisem. W przeprowadzonych testach okazał się najszybszą bazą pod względem odczytu danych, pozostawiając w tyle nawet uznawany za bardzo szybki silnik MyISAM wykorzystywany w bazie MySQL. Jednak do wyników tych należy podejść z lekkim dystansem, ponieważ testy zostały przeprowadzone na stosunkowo niewielkiej liczbie rekordów i tylko przy obsłudze jednego użytkownika. Nie zostały również rozpatrzone przypadki pobierania danych z kilku tabel z wykorzystaniem złączeń. Na tej podstawie nie można przewidzieć jak zachowywałby się SQLite przy bazie dużo większej skali i przy jednoczesnej obsłudze wielu użytkowników.

Przy operacji edycji danych (UPDATE) SQLite, podobnie jak przy zapisie, okazuje się zdecydowanie najwolniejszy ze wszystkich testowanych baz. Powód jest ten sam, co przy operacji INSERT. Aby zwiększyć wydajność tej operacji należy stosować w SQLite transakcje.

# 7.

# Wnioski

Praca ta pokazała, że SQLite może śmiało konkurować z bardziej znanymi i rozbudowanymi systemami bazodanowymi jak MySQL, PostgreSQL czy Firebird i może być dla nich alternatywą.

- Mimo swej prostoty SQLite wspiera większość cech standardu SQL-92, m.in. obsługę transakcji, wyzwalaczy czy widoków
- Brak konieczności konfiguracji i administracji sprawia, że SQLite doskonale sprawdza się jako wbudowana baza danych
- Mimo, że SQLite nie jest oparty na architekturze klient-serwer i nie posiada oddzielnego procesu serwera to umożliwia jednoczesny dostęp do bazy wielu użytkownikom (klientom)
- Minusem SQLite'a jest słaba ochrona przed nieautoryzowanym dostępem ze strony aplikacji klienckich (brak praw użytkowników, jedynie prawa dostępu nadawane plikom bazy)
- Baza zawarta w pojedynczym pliku dyskowym daje możliwość łatwego przenoszenia oraz jest wygodnym rozwiązaniem dla aplikacji desktopowych
- Domyślne korzystanie z transakcji zapewnia lepsze bezpieczeństwo podczas wykonywania operacji SQL-owych i integralność danych w bazie, ale z drugiej strony powoduje wolniejsze wykonywanie operacji wstawiania danych
- Testy wydajności pokazały, że SQLite jest znacznie wolniejszy od innych testowanych baz przy wykonywaniu operacji wstawiania danych (ok. 500 razy wolniejszy od MySQL-a wykorzystującego silnik MyISAM), powodem tego jest traktowanie pojedynczego zapytania jako transakcji, co wymaga stosowania specjalnych czasochłonnych operacji na pliku
- Zgrupowanie zapytań w pojedynczą transakcję w przypadku SQLite znacznie skraca czas ich wykonania (przy wstawianiu danych w ten sposób SQLite okazuje się bazą zdecydowanie najszybszą spośród testowanych)
- SQLite jest najszybszy przy wyszukiwaniu danych w bazie (operacja SELECT), zarówno w przypadku małych baz (5 tyś. rekordów), jak i średnich (100 tyś. rekordów), jest szybszy o ok. 20% od najszybszego silnika MySQL (MyISAM)
- Przy aktualizacji danych w bazie (operacja UPDATE) SQLite wypada słabo na tle konkurentów, główny powód to wolny zapis danych

# 8.

# Podsumowanie

Celem niniejszej pracy było przedstawienie możliwości, jakie oferuje baza SQLite. Choć sama nazwa (ang. lite = lekki, niskokaloryczny) mogłaby wskazywać na coś niepełnowartościowego, okrojonego itp. to jednak nic bardziej mylnego. Mimo pozornej prostoty, SQLite posiada obsługę transakcji, wyzwalaczy, widoków, możliwość tworzenia własnych funkcji oraz wykonywania zagnieżdżonych zapytań, spełniając w większości cechy standardu SQL-92. Ponadto jest bardzo łatwy w utrzymaniu, ponieważ baza mieści się w pojedynczym pliku dyskowym, co ułatwia przenoszenie bazy z jednego komputera na inny.

SQLite nie korzysta z typowego silnika pracującego w tle, ale ze zwykłej biblioteki dostępnej dla większości współczesnych języków programowania. W wielu rozwiązaniach, a w szczególności w systemach wbudowanych, takie rozwiązanie jest najpraktyczniejsze. Dlatego coraz częściej SQLite znajduje zastosowanie w telefonach komórkowych czy odtwarzaczach mp3.

Przeprowadzone na potrzeby tej pracy testy pokazały, że przy obsłudze jednego użytkownika i niewielkich bazach, wydajność SQLite w porównaniu do innych systemów bazodanowych jest całkiem wysoka. O ile wydajność zapisu danych mogłaby być lepsza, to odczyt danych z bazy jest bardzo szybki. Polecenia SELECT wykonywane są przez SQLite'a znacznie szybciej niż przez pozostałe bazy poddane testom, czyli MySQL, PostgreSQL i Firebird.

SQLite jest ciągle rozwijany i ulepszany, co pewien czas dodawane są nowe funkcje oraz tworzone różnego rodzaju rozszerzenia. Jednym z takich rozszerzeń jest opisany w niniejszej pracy SpatiaLite pozwalający na przechowywanie i przetwarzanie danych przestrzennych. SpatiaLite zgodny jest ze specyfikacją OpenGIS, określającą wymagania stawiane rozszerzeniom GIS-owym dla relacyjnych baz danych.

Wszystkie wyżej przedstawione argumenty sprawiają, że SQLite jest naprawdę ciekawą alternatywą dla bardziej znanych systemów bazodanowych i z miesiąca na miesiąc staje się coraz popularniejszy, zyskując wielu nowych użytkowników.

# Spis tabel

# Spis rysunków

# Bibliografia

1. Dubois Paul, „MySQL Podręcznik administratora”, Wydawnictwo Helion, Gliwice 2005
2. Trachtenberg Adam, „PHP 5 Nowe możliwości”, Wydawnictwo Helion, Gliwice 2006
3. Harrington Jan, „SQL dla każdego”, Wydawnictwo MIKOM, Warszawa 1998
4. Richard Stones, „Bazy danych i PostgreSQL od podstaw”, Wydawnictwo Helion, Gliwice 2002
5. Guting R. H., „An introduction to spatial database systems”, Hagen 1994
6. Elmasri Ramez, “Wprowadzenie do systemów baz danych”, Wydawnictwo Helion, Gliwice 2005
7. Allen Sharon, „Modelowanie danych”, Wydawnictwo Helion, Gliwice 2006
8. Kent William, “A Simple Guide to Five Normal Forms in Relational Database Theory ", Communications of the ACM 26(2), 1983
9. Codd Edgar Frank, "A Relational Model of Data for Large Shared Data Banks", Comm. ACM 13 (6), 1970
10. Whitehorn Mark, „Relacyjne bazy danych”, Wydawnictwo Helion, Gliwice 2003

# Netografia

1. Oficjalna dokumentacja SQLite: http://www.sqlite.org/docs.html
2. Oficjalna dokumentacja MySQL: http://dev.mysql.com/doc/
3. Oficjalna dokumentacja PostgreSQL: http://postgresql.org/docs/8.3/static/index.html
4. Oficjalna dokumentacja Firebird: http://www.firebirdsql.org/index.php?op=doc
5. Oficjalna dokumentacja PHP: http://www.php.net/manual/en
6. SpatiaLite – oficjalna strona projektu: http://www.gaia-gis.it/spatialite
7. Open Geospatial Consortium: http://www.opengeospatial.org
8. Wikipedia – model relacyjny: http://pl.wikipedia.org/wiki/Model_relacyjny
9. Oficjalna strona projektu PostGIS: http://postgis.refractions.net/
10. R-drzewa - materiały dydaktyczne Politechniki Poznańskiej: http://hydrus.et.put.poznan.pl/~junior/skisr/materialy/sem7/ZSBD