Programowanie urządzeń mobilnych

Wykorzystanie baz danych

Tłumaczenie i adaptacja materiałów: dr Tomasz Xięski.
Na podstawie prezentacji udostępnionych przez Victor Matos, Cleveland State University.

Bazy danych

Wykorzystanie baz danych w Androidzie

W architekturze systemu Android został zintegrowany system zarządzania bazą danych **SQLite** który może być wykorzystany do:

Tworzenia bazy danych,

Zdefiniowania
- tabel,
- indeksów,
- zapytań,
- widoków,
- wyzwalaczy,

Dodawania,
Usuwania,
Zmiany wierszy,
Wykonywania zapytań,

Administracji bazą.

Właściwości bazy SQLite

- Transakcyjny system bazy danych,
- Małe wymogi (mniej niż 400Kb),
- Brak jawnego typowania,
- Brak konieczności stawiania dedykowanego serwera,
- Brak konieczności wstępnej konfiguracji,
- Źródło dostępne jest na licencji public domain.

Zgodnie ze stroną internetową, SQLite jest *most widely deployed SQL database engine in the world*.

Źródło:
http://sqlite.org/index.html

Właściwości bazy SQLite

1. SQLite większość wytycznych standardu SQL-92.
2. Posiada częściowe wsparcie dla wyzwalaczy oraz umożliwia definiowanie szczegółowych zapytań (za wyjątkiem: złączeń *right/full outer joins, poleceń grant/revoke, zapisu do widoku*).
3. Posiada pewne ograniczenia w sprawdzaniu relacji między kluczami głównym i obcym.
4. Typy danych powiązane są z każdym polem a nie z każdą kolumną.
5. Nie ma domyślnie sprawdzania typów danych, dlatego możliwe jest dodawanie ciągu znaków do pola reprezentującego liczby całkowite.

Dokumentacja dostępna jest pod: http://www.sqlite.org/sqlite.html
Narzędzia do zarządzania bazą:
SQL Administrator http://sqliteadmin.orbmu2k.de/
SQL Expert http://www.sqliteexpert.com/download.html

Bazy danych

Tworzenie bazy danych – metoda pierwsza

```
SQLiteDatabase.openDatabase( myDbPath, null,
                                SQLiteDatabase.CREATE_IF_NECESSARY);
```

Jeżeli baza nie istnieje – stwórz ją. W przeciwnym przypadku otwórz istniejącą bazę z flagami:
OPEN_READWRITE, OPEN_READONLY, CREATE_IF_NECESSARY .

Parametry:

path Ścieżka do pliku reprezentującego bazę danych

factory Opcjonalna referencja do klasy, której obiekt jest tworzony podczas trawersowania po bazie (domyślnie *null).*

flags Flagi do kontroli dostępu do bazy.

Zwraca Referencję do otwartego zasobu bazy danych, lub
Wyjątek *SQLiteException* jeśli baza nie może zostać otwarta

Bazy danych

Przykład1. Tworzenie bazy danych – metoda pierwsza

```
package cis70.matos.sqldatabases;
public class SQLDemo1 extends Activity {
   SQLiteDatabase db;
    @Override
    public void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.main);
         TextView txtMsg = (TextView)findViewById(R.id.txtMsg);

         // path for internal memory: "data/data/cis470.matos.sqldatabases“
         // path name to the external SD card: /storage/sdcard
         String SDcardPath = Environment.getExternalStorageDirectory().getPath();

         String myDbPath = SDcardPath + "/" + "myfriends";
         txtMsg.setText("DB Path: " + myDbPath);

         try {
               db = SQLiteDatabase.openDatabase(myDbPath, null,
                                                SQLiteDatabase.CREATE_IF_NECESSARY);
               // here you do something interesting with your database ...
               db.close();
               txtMsg.append("\nAll done!");
              }
         catch (SQLiteException e) {
            txtMsg.append( e.getMessage() );
        }
    }//onCreate
}//class
```

Bazy danych

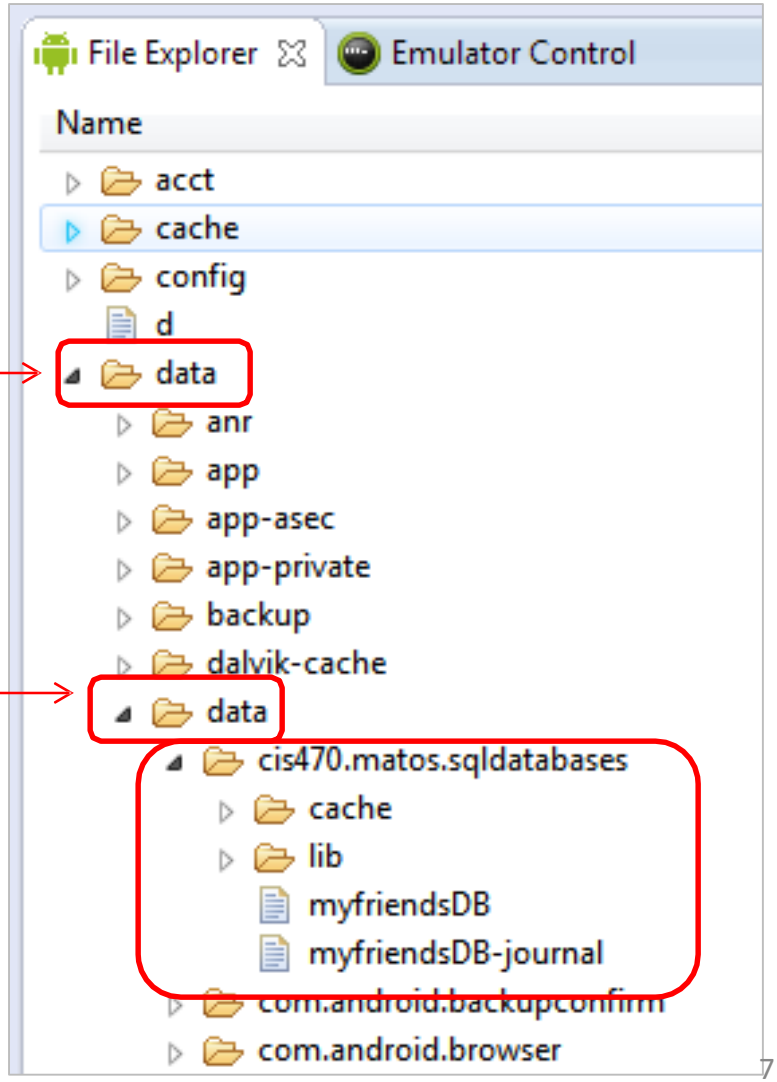
Przykład1: Lokalizacja bazy

Zakłada się, że baza SQLite przechowywana jest w wewnętrznej pamięci urządzenia.

Ścieżka:

/data/data/cis470.matos.sqldatabases/

gdzie: *cis470.matos.sqldatabases* to nazwa pakietu.



Bazy danych

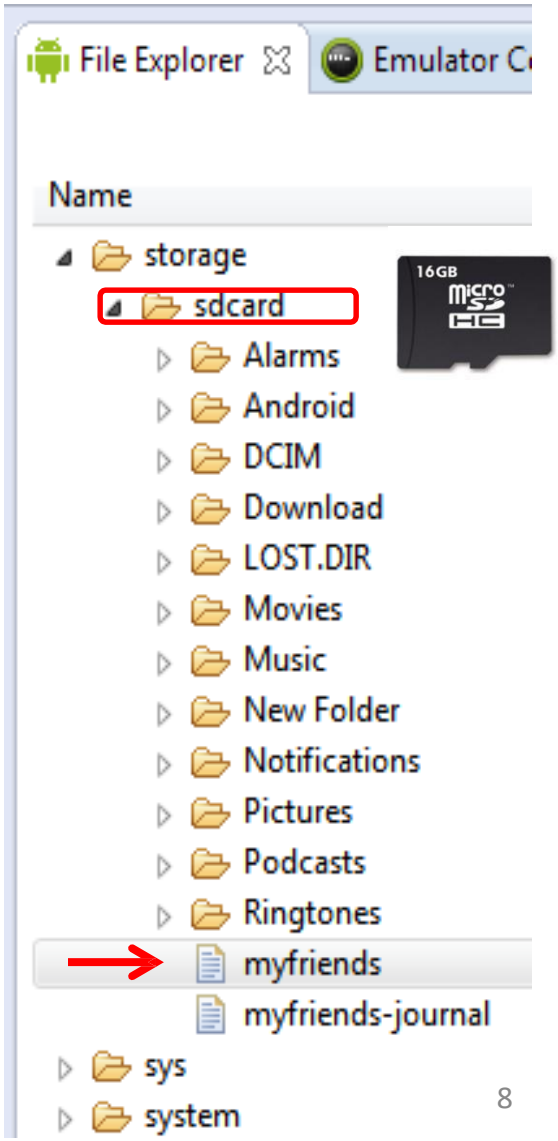
Przykład1: Lokalizacja bazy na karcie pamięci

Należy wykorzystać schemat:

```
SQLiteDatabase db;
String SDcardPath = Environment
            .getExternalStorageDirectory()
            .getPath() + "/myfriends";

db = SQLiteDatabase.openDatabase(
        SDcardPath, null,
        SQLiteDatabase.CREATE_IF_NECESSARY
        );
```

Manifest musi zawierać:

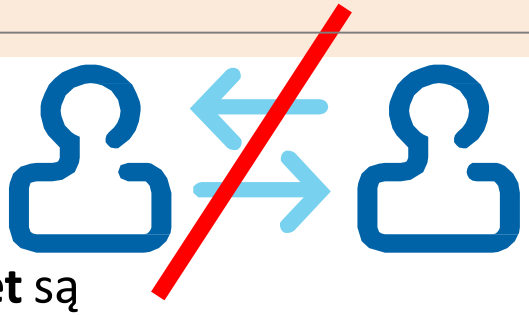
```
<uses-permission android:name=
"android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name=
"android.permission.READ_EXTERNAL_STORAGE" />
```



Bazy danych

Współdzielenie bazy

Uwaga



- Bazy danych przechowywane w **/data/data/pakiet** są prywatną własnością tego pakietu.
- *Nie można* uzyskać dostępu do baz danych innych aplikacji (w takim przypadku należy wykorzystać mechanizm Dostawcy treści lub zapisać bazę na zewnętrznej karcie pamięci).

Baza znajdująca się na karcie pamięci wymaga następujących uprawnień:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"  />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"  />
```

Uwaga: *Wielkość liter (np. dla identyfikatorów) w bazie SQLite nie ma znaczenia.*

Bazy danych

Alternatywny sposób: openOrCreateDatabase

Alternatywny sposób otwarcia/stworzenia bazy danych podany jest poniżej:

```
SQLiteDatabase   db =   this.openOrCreateDatabase(
                                        "myfriendsDB",
                                        MODE_PRIVATE,
                                        null);
```

Przy założeniu, że aplikacja korzystająca z bazy należy do pakietu **cis470.matos.sqldatabases**, pełna ścieżka do bazy danych to:

/data/data/cis470.matos.sqldatabases/myfriendsDB

Pamięć wewnętrzna | Nazwa pakietu | Nazwa bazy

- Plik bazy jest dostępny dla wszystkich komponentów danej aplikacji.
- Pozostałe tryby otwarcia: *MODE_WORLD_READABLE* oraz *MODE_WORLD_WRITEABLE* zostały wykluczone od API 17.
- Wartość **null** odnosi się parametru **factory.**

Typy zapytań SQL

Po stworzeniu, baza SQLite jest gotowa do wykonywania standardowych operacji jak: *tworzenie, edycja, usuwanie zasobów (tabel, indeksów, wyzwalaczy, widoków, zapytań itp.) lub administracji zasobami.*

Zapytania modyfikujące i ***wybierające dane*** reprezentują typowe grupy zapytań SQL.

- *Zapytanie wybierające* zwykle składa się z polecenie *Select, w którym wybierany jest zestaw danych pochodzący z jednej lub wielu tabel*.

- *Zapytanie modyfikujące* zwykle dotyczy zadań aktualizacyjnych bądź administracyjnych takich jak manipulacja tabelami czy zmiennymi środowiskowymi.

Przetwarzanie transakcji

Transakcje są pożądane ponieważ pomagają utrzymać i zachować integralność danych oraz zapobiec niezamierzonej czy nadpisaniu utracie danych (np. poprzez przedwczesne zakończenie działania zapytania).

Generalnie, zapytania modyfikujące powinny być wykonywane wewnątrz bloku transakcyjnym, który *implementuje politykę* "*prawidłowego zakończenia wszystkich elementów składowych bądź całościową porażkę*".

Jest to odwołanie do pojęcia ***atomizacji*** *by odzwierciedlić relację połączoną wszystkich elementów składowych transakcji.*

Bazy danych

Przetwarzanie transakcji

Standardowy sposób implementacji transakcji w systemie Android jest następujący (przy założeniu, że **db** jest obiektem typu *SQLiteDatabase*)

```
db.beginTransaction();
try {
   //perform your database operations here ...
   db.setTransactionSuccessful(); //commit your changes
}
catch (SQLiteException e) {
   //report problem
}
finally {
   db.endTransaction();
}
```

Transakcja zdefiniowana jest w bloku między metodami *beginTransaction* oraz *endTransaction*. Należy wywołać *setTransactionSuccessful()* by zatwierdzić wszystkie zmiany. Absencja tego wykonania powoduje cofnięcie zmian czyli tzw. *rollback* – *implikuje to reset bazy danych do stanu sprzed transakcji.*

Tworzenie i wypełnienie tabeli SQL

recID	name	phone
1	AAA	555-1111
2	BBB	555-2222
3	CCC	555-3333

Składnia **SQL** dotycząca tworzenia prostej tabeli oraz wypełnienia jej danymi zaprezentowana jest poniżej:

```
create table tblAMIGO (
              recID integer PRIMARY KEY autoincrement,
              name  text,
              phone text );
```

```
insert into tblAMIGO(name, phone) values ('AAA', '555-1111' );
```

Wartość typu *autoinkrement* dla kolumny *recID* nie została określona w zapytaniu SQL, gdyż jest wyznaczana przez system zarządzania bazą danych.

Bazy danych

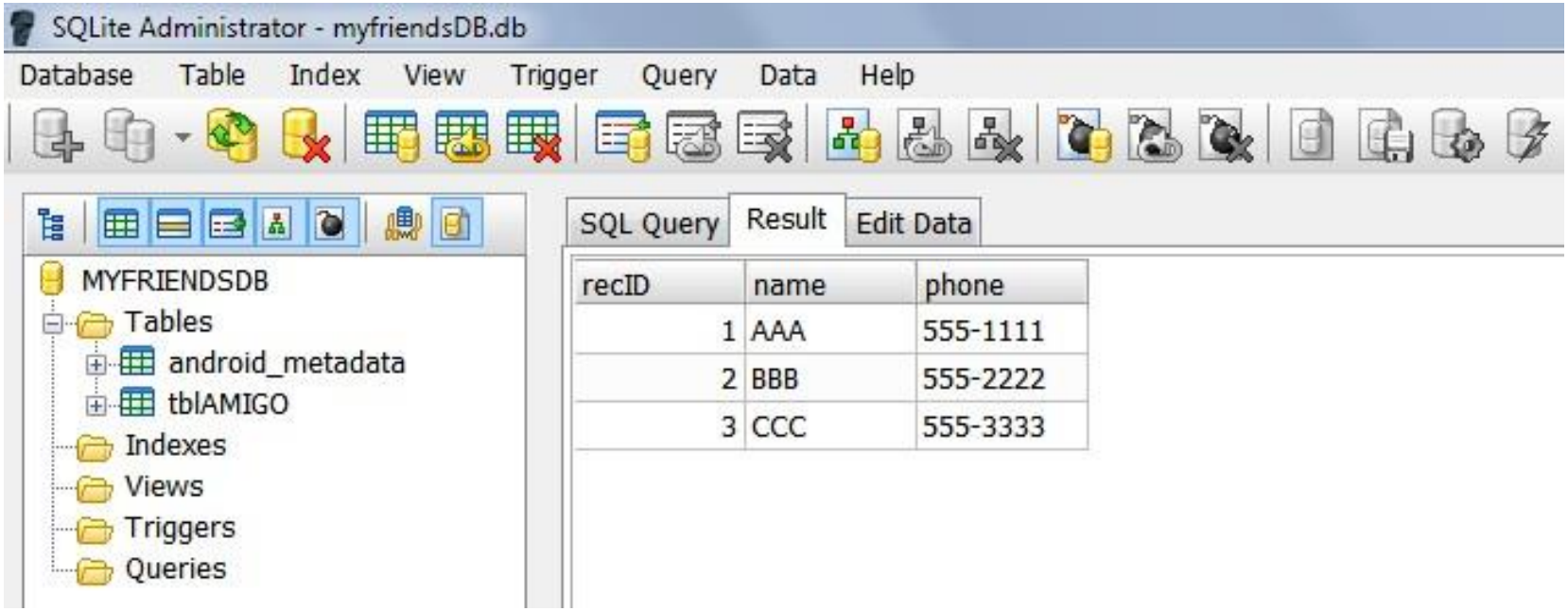
Przykład 2. Tworzenie i wypełnienie tabeli SQL

- Aplikacja w tym przykładzie wykorzystuje metodę **execSQL()** by wysłać zapytanie SQL. Tworzona jest tabela o nazwie **tblAmigo**.
- Tabela ma 3 kolumny: unikalny identyfikator ***recID*** i dwie kolumny reprezentujące ciągi ***imię*** oraz ***telefon***.
- Jeżeli tabela o takiej nazwie istnieje jest ona usuwana a w jej miejsce tworzona jest nowa.
- Ostatecznie trzy wiersze są dodawane do tabeli.

```
db.execSQL("create table tblAMIGO ("

           + " recID integer PRIMARY KEY autoincrement, "
           + " name  text, "
           + " phone text );  "    );

db.execSQL( "insert into tblAMIGO(name, phone) values ('AAA', '555-1111');");

db.execSQL( "insert into tblAMIGO(name, phone) values ('BBB', '555-2222');");

db.execSQL( "insert into tblAMIGO(name, phone) values ('CCC', '555-3333');");
```

Przykład 2. Tworzenie i wypełnienie tabeli SQL

- Weryfikacja poprawności utworzonej struktury oraz wprowadzonych danych wykonywana jest często z użyciem narzędzia zewnętrznego.
- Poniżej zaprezentowano rezultat działania zapytania **select * from tblAmigo**.



recID	name	phone
1	AAA	555
2	BBB	777
3	CCC	999

Przykład 2. Tworzenie i wypełnienie tabeli SQL

Komentarz

1. Pole **recID** zdefiniowano jako **KLUCZ GŁÓWNY**.

2. Wykorzystanie wartości typu autoinkrement gwarantuje unikalność i rozróżnialność każdego z rekordów.

3. Podobnie jak inne systemy bazodanowe SQLite oferuje następujące typu danych: ***text, varchar, integer, float, numeric, date, time, timestamp, blob, boolean.***

4. Generalnie każde poprawne zapytanie typu (`insert, delete, update, create, drop, alter`, itp.) może zostać wykorzystane wewnątrz metody **execSQL()**.

Uwaga:

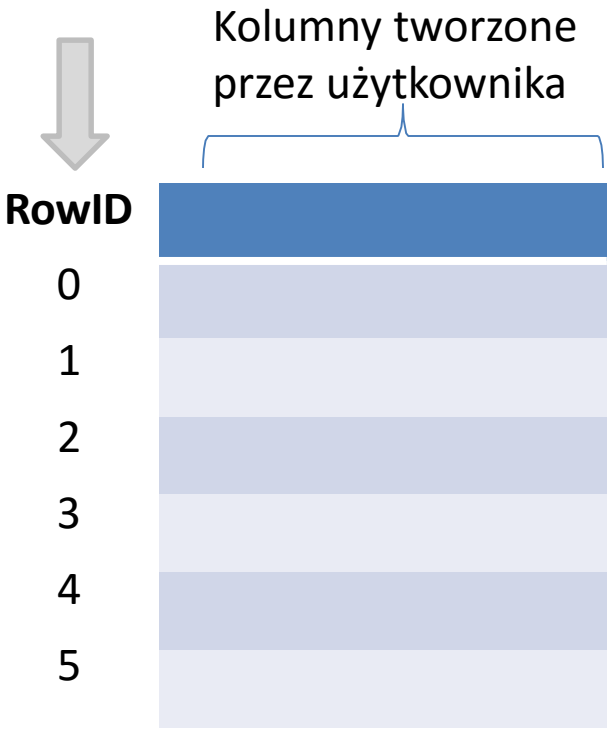
Metoda **execSQL()** powinna być wykonywana wewnątrz bloku ***try-catch-finally*** pamiętając o obsłudze wyjątków typu **SQLiteException**.

Przykład 2. Tworzenie i wypełnienie tabeli SQL

Uwaga:

SQLITE wykorzystuje **niewidoczne** pole zwane ***ROWID*** by unikalnie identyfikować rekordy.

W prezentowanych przykładach pola ***recID*** oraz ***ROWID*** są funkcjonalnie równoważne.



Przykłady poleceń SQL

Przykład A.

```
SELECT    LastName, cellPhone
  FROM    ClientTable
 WHERE    state = ‘Ohio’
ORDER BY  LastName
```

Przykład B.

```
SELECT    city, count(*) as TotalClients
  FROM    ClientTable
GROUP BY  city
```

Przykład3. Zapytania wykorzystujące kursory (wariant 1)

Rozważmy następujące zapytanie:

```
Cursor c1 = db.rawQuery("select * from tblAMIGO",  null);
```

1. Składa się ono z zapytania typu SELECT, wybierającego wszelkie dane zwarte w tabeli tblAMIGO. Rezultat zwracany jest jako obiekt klasy **Cursor**.
2. Kursor **c1** wykorzystywany jest do przejrzenia wszelkich wierszy zwróconego rezultatu.
3. Pobranie następnego rekordu w kursorze **c1** wymaga jawnego żądania przejścia do kolejnej pozycji w tabeli uzyskanej w odpowiedzi.
5. Pola rezultatu muszą być powiązanie ze zmiennymi lokalnymi dostępnymi po stronie Javy.

Przykład3. Zapytania wykorzystujące kursory (wariant 2)

Przekazywanie argumentów.

Załóżmy, że chcemy policzyć ile jest osób o imieniu ‘BBB’ oraz identyfikatorze recID > 1. W tym celu można wykorzystać zapis:

```
String mySQL = "select count(*) as Total "
                + " from tblAmigo "
                + " where recID > ? "
                + "   and name  = ? ";

 String[] args = {"1", "BBB"};

 Cursor c1 = db.rawQuery(mySQL, args);
```

Symbole ‘**?**’ pozwalają przyporządkować kolejne wartości podane jako tablica argumentów przekazana do metody **rawQuery**(). Istotna jest kolejność oraz prawidłowa liczba argumentów.

Przykład3. Zapytania wykorzystujące kursory (wariant 3)

Załóżmy, że cel tego przykładu pozostał niezmienny tj. chcemy wyznaczyć liczbę osób o imieniu BBB oraz identyfikatorze większym od jednyki. Inny sposób na realizację tego zadania to:

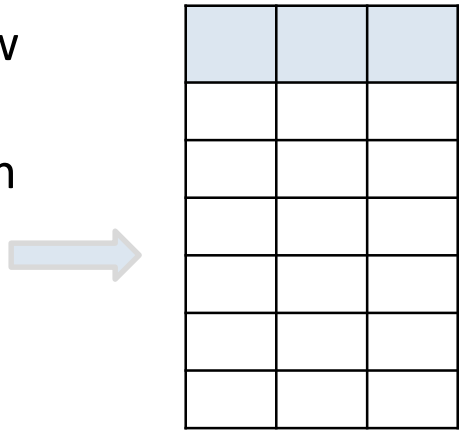
```
String[] args = {"1", "BBB"};

String mySQL = " select count(*) as Total "
               + "  from tblAmigo "
               + " where recID > "  + args[0]
               + "   and name  = '" + args[1] + "'";

Cursor c1 = db.rawQuery(mySQL, null);
```

Zamiast wykorzystania symboli ‘**?**’ została zastosowania konkatenacja.

Bazy danych

Kursory SQL

Kursory wykorzystywane są do uzyskiwania dostępu sekwencyjnego bądź swobodnego do tabel uzyskiwanych w wyniku działania polecenia SELECT.
Kursory działają w trybie *jeden wiersz naraz*. W niektórych SZBD kursory mogą być wykorzystywane do zmiany danych, ale w SQLite działają one w trybie **tylko do odczytu**.



Najpopularniejsze operacje na kursorach to:

1. **Informacja o pozycji:** isFirst(), isLast(), isBeforeFirst(), isAfterLast().
2. **Nawigacja po rekordach:** moveToFirst(), moveToLast(), moveToNext(), moveToPrevious(), move(n).
3. **Pobieranie pól:** getInt, getString, getFloat, getBlob, getDouble, etc.
4. **Informacje o schemacie:** getColumnName(), getColumnNames(), getColumnIndex(), getColumnCount(), getCount().

Przykład 4. Wykorzystanie kursora

```
1 private String showCursor( Cursor cursor) {
      // reset cursor's top (before first row)
      cursor.moveToPosition(-1);
      String cursorData = "\nCursor: [";

      try {
         // get SCHEMA (column names & types)
2        String[] colName =
         cursor.getColumnNames(); for(int i=0;
         i<colName.length; i++){
            String dataType = getColumnType(cursor, i);
            cursorData += colName[i] + dataType;

            if (i<colName.length-1){
               cursorData+= ", ";
            }
         }
      } catch (Exception e) {
         Log.e( "<<SCHEMA>>" , e.getMessage() );
      }
      cursorData += "]";

      // now get the rows
      cursor.moveToPosition(-1); //reset cursor's top
```

Przykład 4. Wykorzystanie kursora

```
3 →    while (cursor.moveToNext()) {
          String cursorRow = "\n[";
          for (int i = 0; i < cursor.getColumnCount(); i++) {
             cursorRow += cursor.getString(i);
4 →          if (i<cursor.getColumnCount()-1)
                cursorRow +=  ", ";
          }
          cursorData += cursorRow + "]";
       }
       return cursorData + "\n";
    }

    private String getColumnType(Cursor cursor, int i) {
5 →    try {
          //peek at a row holding valid data
          cursor.moveToFirst();
          int result = cursor.getType(i);
          String[] types = {":NULL", ":INT", ":FLOAT", ":STR", ":BLOB", ":UNK"
          };
          //backtrack - reset cursor's top
          cursor.moveToPosition(-1);
          return types[result];
       } catch (Exception e) {
          return " ";
    }  }
```

Zapytania wykorzystujące szablony

Proste zapytania SQL mogą wykorzystać *szablon*, którego celem jest pomóc programistom nie zaznajomionym z językiem SQL w manipulacji bazami danych.

Szablon udostępnia wszystkie elementy prostego zapytania SQL.

Tego typu zapytania mogą jednak otrzymywać dane tylko z *jednej tabeli*.

Sygnatura metody posiada siedem następujących argumentów:

1. Nazwa tabeli,
2. Nazwy kolumn do wybrania,
3. Kryterium wyboru (klauzula *where*),
4. Argumenty dla klauzuli *where*,
5. Klauzula *group by*,
6. Klauzula *having*,
7. Klauzula *order by*.

Zapytania wykorzystujące szablony

Graficznie sygnaturę metody **query** (wykorzystującej szablon) można przedstawić jako:

```
db.query (String   table,
          String[] columns,
          String   selection,
          String[] selectionArgs,
          String   groupBy,
          String   having,
          String   orderBy )
```

Przykład5. Zapytania wykorzystujące szablony

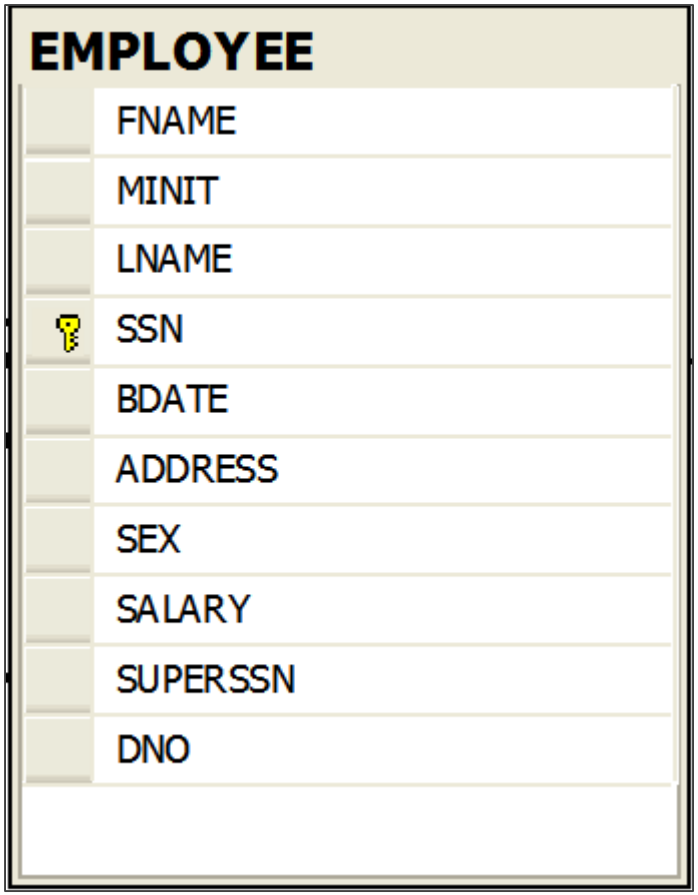
Załóżmy, że istnieje tabela **EmployeeTable** i na jej podstawie chcemy wyznaczyć średnią wysokość pensji wszystkich zatrudnionych kobiet, których kierownik to pracownik o numerze 123456789. Wyniki powinny być posortowane malejąco względem średniej wypłaty. Ponadto w wynikach nie uwzględniono działy mające dwóch pracowników bądź mniej.

```
String[] columns = {"Dno", "Avg(Salary) as AVG"};

String[] conditionArgs = {"F", "123456789"};

Cursor c = db.query("EmployeeTable",                  ← Nazwa tabeli
                    columns,                          ← Kolumny
                    "sex = ? And superSsn = ? ",      ← Warunek
                    conditionArgs,                    ← Argumenty warunku
                    "Dno",                            ← group by
                    "Count(*) > 2",                   ← having
                    "AVG Desc "                       ← order by
                   );
```

Przykład5. Zapytania wykorzystujące szablony

Slajd prezentuje graficzną prezentację schematu tabeli **EmployeeTable** użytej w poprzednim przykładzie.

Zawiera ona kolejno: imię, inicjał imienia, nazwisko, numer ubezpieczenia, datę urodzenia, adres, płeć, pensję, numer ubezpieczenia kierownika oraz dział w którym pracownik jest zatrudniony.



Przykład6. Zapytania wykorzystujące szablony

W tym przykładzie zostanie wykorzystana tabela **tblAmigo**. Interesują nas kolumny: *recID*, *name* oraz *phone*. Warunki wyboru zestawu rekordów to: RecID większe od 2, imię zaczynające się na ‘B’ oraz posiadające co najmniej 3 litery.

```
 String [] columns = {"recID", "name", "phone"};

 Cursor c1 = db.query (
       "tblAMIGO",
       columns,
       "recID > 2 and length(name) >= 3 and name like 'B%' ",
       null, null, null,
       "recID" );

int recRetrieved = c1.getCount();
```

Wartość **null** wpisywana jest dla każdej sekcji zapytania która powinna zostać pominięta (having, group by itp.).

Przykład7. Zapytania wykorzystujące szablony

W tym przykładzie zostanie zaprezentowane zapytanie agregujące.

Tym razem interesuje nas ile jest grup przyjaciół o identyfikatorach większych od 3, którzy mają te same imię. W każdej takiej grupie mogą być maksymalnie 4 osoby.

Przykładowe zapytanie SQL wygląda następująco:

```
 select  name, count(*) as TotalSubGroup
   from  tblAMIGO
  where  recID > 3
  group  by name
 having  count(*) <= 4;
```

Przykład7. Zapytania wykorzystujące szablony

Przykładowa implementacja wykorzystująca mechanizm szablonów:

```
1 String [] selectColumns =  {"name", "count(*) as TotalSubGroup"};
2 String    whereCondition = "recID > ? ";
  String [] whereConditionArgs = {"3"};
3 String    groupBy = "name";
  String    having =  "count(*) <= 4";
  String    orderBy = "name";

  Cursor cursor =  db.query ( "tblAMIGO",
                              selectColumns,
                              whereCondition,
                              whereConditionArgs,
                              groupBy,
                              having,
                              orederBy );
```

Zapytania modyfikujące

Zapytania modyfikujące pozwalają na zmianę danych i struktury poszczególnych zasobów bazodanowych. Przykładowe typy zapytań to: *insert, delete, update, create table, drop, itp*.

Przykłady:

```
insert into tblAmigos
   values ( ‘Macarena’,  ‘555-1234’ );

update tblAmigos
   set name = ‘Maria Macarena’
   where phone = ‘555-1234’;

delete from tblAmigos
   where phone = ‘555-1234’;

create table Temp ( column1 int, column2 text, column3 date );

drop table Temp;
```

Zapytania modyfikujące z wykorzystaniem ExecSQL

Jednym z najprostszych sposobów wykonania zapytania modyfikującego w systemie Android jest stworzenie tego zapytania jako ciąg znaków i podanie go jako argument dla metody ***execSQL*()**.

Niestety metoda **execSQL** w kontekście bazy SQLite **NIE** zwraca żadnych danych. Nie wiadomo zatem ile rekordów zostało faktycznie zmienionych. Dlatego preferowane są inne metody omówione w dalszej części prezentacji

```
db.execSQL(
    "update tblAMIGO set name = (name || 'XXX') where phone >= '555-1111' ");
```

Powyższe zapytanie dodaje ciąg ‘XXX’ do imion pracowników, których numer telefonu jest większy bądź równy od ‘555-1111’.

Uwaga
Symbol **||** oznacza operator konkatenacji

Zapytania modyfikujące z wykorzystaniem ExecSQL

Alternatywna wersja realizacji tego zadania zakłada bezpośrednie łączenie elementów ciągu zapytania za pomocą operatora +:

```
String theValue  =  " …";

db.execSQL( "update tblAMIGO set name = (name || 'XXX') " +
                    " where phone >= '" + theValue + "' " );
```

Ta sama strategia może zostać bezpośrednio zastosowana do innych zapytań modyfikujących takich jak:

“delete from … where…”,
“insert into ….values…”, itp.

Bazy danych

Metody do zapytań typu INSERT, DELETE, UPDATE

- System Android dostarcza szereg metod do wykonywania operacji typu *insert, delete, czy update*.
- Wszystkie zwracają pewien rezultat jak np. identyfikator ostatnio dodanego rekordu, czy liczba wierszy objęta daną operacją. Dlatego stanowią alternatywę dla metody **execSQL**.

```
public long insert(String table,
                   String nullColumnHack,
                   ContentValues values )

public int update(String table,
                  ContentValues values,
                  String whereClause,
                  String[] whereArgs )

public int delete(String table,
                  String whereClause,
                  String[] whereArgs)
```

Bazy danych

Klasa ContentValues

- Wspomniana klasa wykorzystywana jest do zapisywania par **[klucz, wartość]** (co funkcjonalnie odpowiada klasie Bundle).
- W kontekście bazy SQLite, obiekt typu ContentValues stanowi wygodny sposób na dostarczenie dowolnej liczby argumentów do zapytania.
- Podobnie jak Bundle, klasa posiada szereg standardowych metod do odczytu oraz zapisu tego typu danych.

```
ContentValues myArgs= new ContentValues();

myArgs.put("name", "ABC");
myArgs.put("phone", "555-7777");
```

myArgs

Klucz	Wartość
name	ABC
phone	555-7777

Operacja INSERT

public long **insert**(String **table**, String **nullColumnHack**, ContentValues **values**)

Prezentowana metoda służy dodawaniu wierszy do tabeli. Nowe wartości przekazywane są w strukturze typu ContentValues. Metoda zwraca identyfikator **nadany nowemu rekordowi**, lub -1 w przeciwnym przypadku.

Parametry

table	Tabela do której mają zostać dodane nowe wiersze
nullColumnHack	Puste i Null to dwie różne rzeczy. Jeżeli zostanie podana pusta wartość (np. pusty ciąg) do danego pola zostanie wstawiona wartość NULL.
values	Podobnie jak dla obiektów typu Bundle – pary (*klucz, wartość*) zawierające wartości dla poszczególnych pól.

Operacja INSERT

```
1 → ContentValues rowValues= new ContentValues();

    rowValues.put("name", "ABC");
    rowValues.put("phone", "555-1010");
2 → long rowPosition = db.insert("tblAMIGO", null, rowValues);

3 → rowValues.put("name", "DEF");
    rowValues.put("phone", "555-2020");
    rowPosition = db.insert("tblAMIGO", null, rowValues);

4 → rowValues.clear();

5 → rowPosition = db.insert("tblAMIGO", null, rowValues);

6 → rowPosition = db.insert("tblAMIGO", "name", rowValues);
```

Operacja UPDATE

```
public int update (  String table,    ContentValues values,
                     String whereClause,  String[] whereArgs  )
```

Zadaniem metody jest aktualizacja danych. Klauzula SQL **set column=newvalue** przekształcona została na tablicę par (klucz, wartość). Metoda zwraca liczbę zmienionych wierszy.

Parametry

table	Nazwa tabeli, której dane należy zaaktualizować.
values	Podobnie jak w Bundle, para (*klucz, wartość*) zawierająca nazwę kolumny oraz nową wartość jaka ma zostać wpisana.
whereClause	Warunek określający jakie wiersze mają zostać zmienione. Podanie wartości **null** skutkuje aktualizacją całej tabeli.
whereArgs	Wartości argumentów (?) zawartych w poprzedniej części.

Operacja UPDATE

Przykład

Celem przykładu jest wykorzystanie metody update() by zrealizować zapytanie SQL:

Update tblAmigo set name = ‘maria’ where (recID > 2 and recID < 7)

Przykładowa realizacja wygląda następująco:

```
    String [] whereArgs = {"2", "7"};

1 →  ContentValues updValues = new ContentValues();

2 →  updValues.put("name", "Maria");

3 →  int recAffected =  db.update( "tblAMIGO",
                                   updValues,
                                   "recID > ? and recID < ?",
                                   whereArgs );
```

Operacja DELETE

```
public int delete (   String table,   String whereClause,  String[] whereArgs  )
```

Celem metody jest usunięcie poszczególnych wierszy z tabeli. Należy jednak pamiętać o ograniczeniu liczby usuwanych wierszy klauzulą *WHERE*. Metoda zwraca liczbę usuniętych wierszy.

Parametry

table	Tabela z której usuwane będą dane
whereClause	Warunek, który ograniczy liczbę usuwanych wierszy. Przykładowym warunkiem może być “name = ? “ gdzie wartość ? zostanie dobrana na podstawie argumentu whereAgrs. Podanie wartości **null** usuwa wszystkie wiersze tabeli.
whereArgs	Wartości argumentów (?) zawartych w poprzedniej części.

Operacja DELETE

Przykład

Celem przykładu jest wykorzystanie metody delete() by zrealizować zapytanie SQL:

```
Delete from tblAmigo wehere recID > 2 and recID < 7
```

Przykładowa implementacja jest następująca:

```
String [] whereArgs = {"2", "7"};

int recAffected = db.delete("tblAMIGO",
                            "recID > ? and recID < ?",
                            whereArgs);
```

Rekord powinien zostać usunięty jeżeli jego identyfikator ma wartość z przedziału [2, 7]. Metoda zwraca liczbę usuniętych wierszy (lub 0 jeśli nie usunięto żadnego).

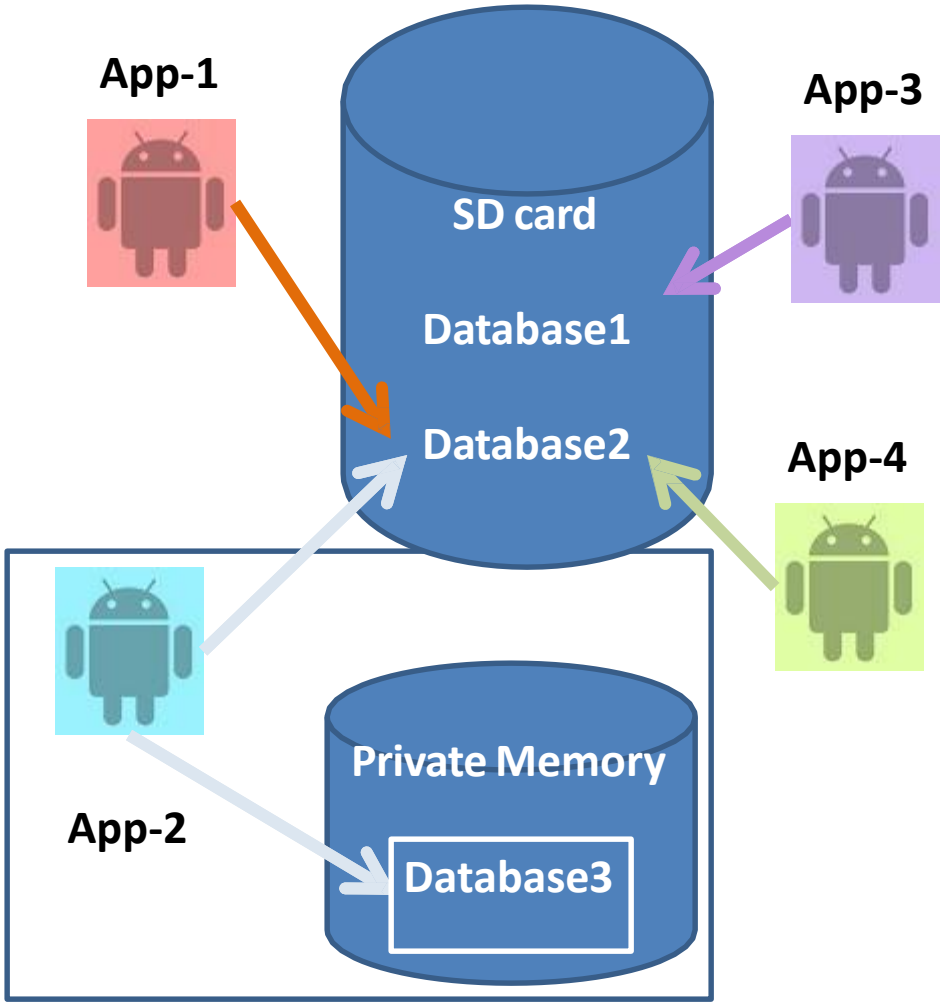
Bazy danych

Widoczność bazy danych

1. Każda aplikacja może uzyskać dostęp do bazy **umieszczonej na zewnętrznej karcie SD**. Jedyne co jest w tym celu potrzebne to ścieżka i nazwa bazy danych.

2. Bazy danych przechowywane wewnątrz katalogu /data/ są prywatne dla danej aplikacji i nie mogą zostać współdzielone.

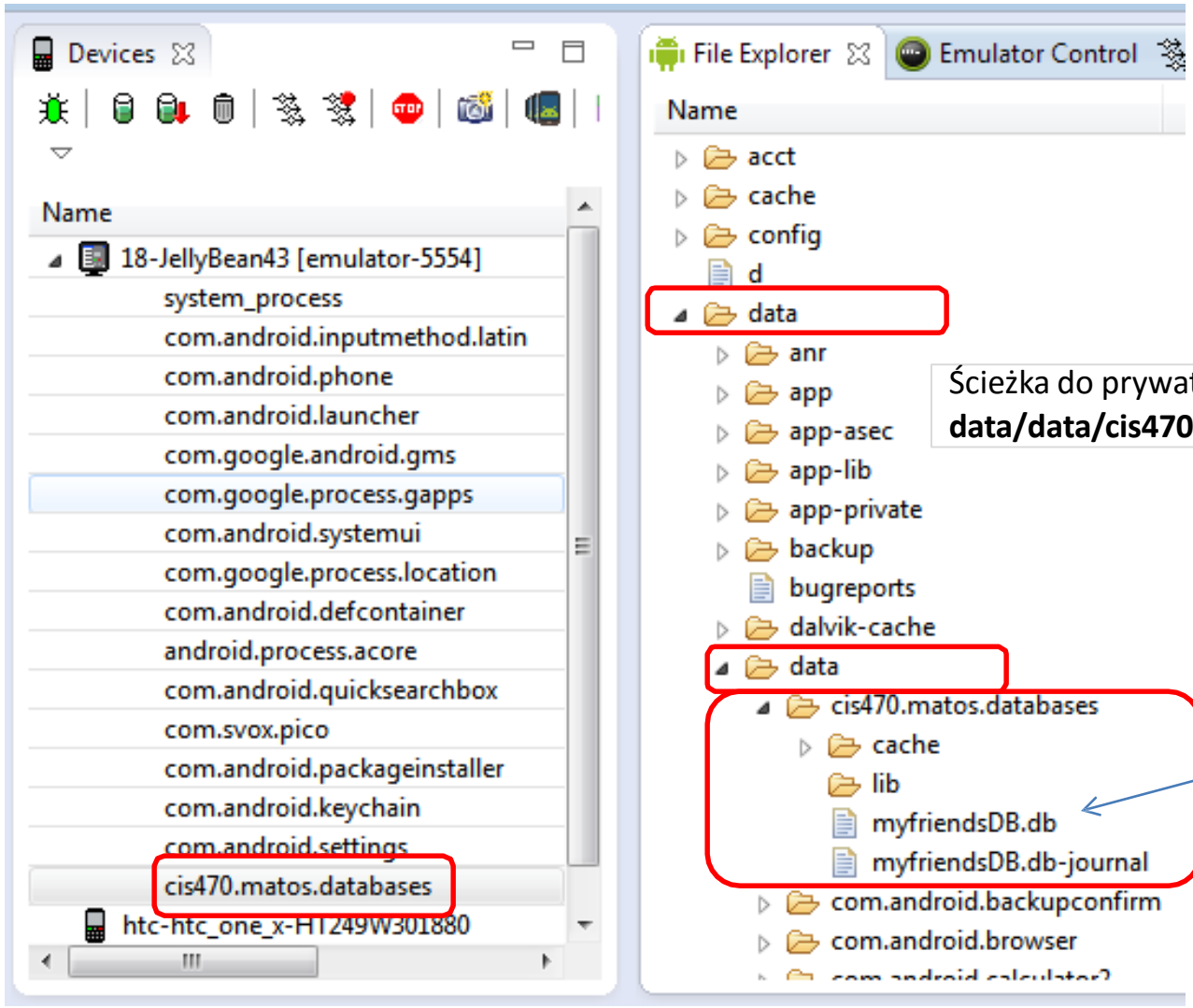
3. Innym sposobem współdzielenia jest wykorzystanie mechanizmu Dostaców Treści (**ContentProvider**).



Bazy danych

Widoczność bazy danych

Widok emulatora przedstawiający lokalizację bazy danych



Ścieżka do prywatnej bazy danych to:
data/data/cis470.matos.databases/myfriendsDB

Bazy danych

Wykorzystanie narzędzi do manipulacji bazą danych

By przenieść bądź skopiować bazę danych na emulator (bądź fizyczne urządzenie) można wykorzystać mechanizm ADB i standardowe komendy do przesyłania/pobierania plików:

adb pull <*full_path_to_database*> oraz
adb push <*full_path_to_database*>.

Można również wykorzystać przystawkę Android Monitor (kiedyś DDMS Perspective ze środowiska Eclipse).

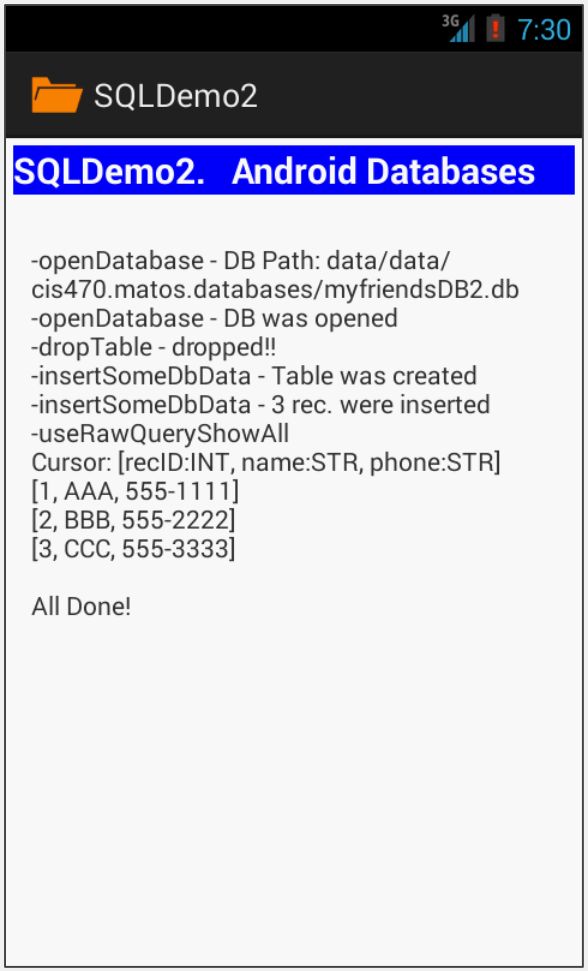
Jeżeli baza zostanie pobrana np. na komputer programisty można wykorzystać dowolne środowisko do manipulacji bazami SQLite:

- **SQLite Administrator** (http://sqliteadmin.orbmu2k.de)
- **SQLite Manager** (dodatek do Firefox)

Bazy danych

Kompletny kod prezentowanych przykładów

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="4dp"
    android:orientation="vertical" >
    <TextView android:id="@+id/txtCaption"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#ff0000ff"
        android:text="SQLDemo2.   Android Databases"
        android:textColor="#ffffffff"
        android:textSize="20dp"
        android:textStyle="bold" />
    <ScrollView
        android:id="@+id/ScrollView01"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="10dp" >
        <TextView android:id="@+id/txtMsg"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="" />
    </ScrollView>
</LinearLayout>
```



Bazy danych

Kompletny kod prezentowanych przykładów

```
public class SQLDemo2 extends Activity {
   SQLiteDatabase db;
   TextView txtMsg;

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      txtMsg = (TextView) findViewById(R.id.txtMsg);

      try {
         openDatabase(); // open (create if needed) database
         dropTable(); // if needed drop table tblAmigos
         insertSomeDbData(); // create-populate tblAmigos
         useRawQueryShowAll(); // display all records
         useRawQuery1(); // fixed SQL with no arguments
         useRawQuery2(); // parameter substitution
         useRawQuery3(); //manual string concatenation
         useSimpleQuery1(); //simple (parametric) query
         useSimpleQuery2(); //nontrivial 'simple query'
         showTable("tblAmigo"); //retrieve all rows from a table
         updateDB(); //use execSQL to update
         useInsertMethod(); //use insert method
         useUpdateMethod(); //use update method
         useDeleteMethod(); //use delete method
         db.close(); // make sure to release the DB
         txtMsg.append("\nAll Done!");
```

Bazy danych

Kompletny kod prezentowanych przykładów

```
      } catch (Exception e) {
         txtMsg.append("\nError onCreate: " + e.getMessage());
         finish();
      }
   }// onCreate

   // ////////////////////////////////////////////////////////////////////////
   private void openDatabase() {
      try {
         // path to private memory database:
          String SDcardPath = "data/data/cis470.matos.databases";
         // ------------------------------------------------------------
         // this provides the path name to the SD card
         // String SDcardPath = Environment.getExternalStorageDirectory().getPath();

         String myDbPath = SDcardPath + "/" + "myfriendsDB2.db";
         txtMsg.append("\n-openDatabase - DB Path: " + myDbPath);

         db = SQLiteDatabase.openDatabase(myDbPath, null,
               SQLiteDatabase.CREATE_IF_NECESSARY);

         txtMsg.append("\n-openDatabase - DB was opened");
      } catch (SQLiteException e) {
         txtMsg.append("\nError openDatabase: " + e.getMessage());
         finish();
      }
   }// createDatabase
```

Bazy danych

Kompletny kod prezentowanych przykładów

```
private void insertSomeDbData() {
   // create table: tblAmigo
   db.beginTransaction();
   try {
      // create table
      db.execSQL("create table tblAMIGO ("
            + " recID integer PRIMARY KEY autoincrement, "
            + " name  text, " + " phone text );  ");
      // commit your changes
      db.setTransactionSuccessful();

      txtMsg.append("\n-insertSomeDbData - Table was created");

   } catch (SQLException e1) {
      txtMsg.append("\nError insertSomeDbData: " + e1.getMessage());
      finish();
   } finally {
      db.endTransaction();
   }

   // populate table: tblAmigo
   db.beginTransaction();
   try {

      // insert rows
      db.execSQL("insert into tblAMIGO(name, phone) "
            + " values ('AAA', '555-1111' );");
```

Bazy danych

Kompletny kod prezentowanych przykładów

```
        db.execSQL("insert into tblAMIGO(name, phone) "
              + " values ('BBB', '555-2222' );");
        db.execSQL("insert into tblAMIGO(name, phone) "
              + " values ('CCC', '555-3333' );");

        // commit your changes
        db.setTransactionSuccessful();
        txtMsg.append("\n-insertSomeDbData - 3 rec. were inserted");

     } catch (SQLiteException e2) {
        txtMsg.append("\nError insertSomeDbData: " + e2.getMessage());

     } finally {
        db.endTransaction();
     }

  }// insertSomeData


  // ////////////////////////////////////////////////////////////////
  private void useRawQueryShowAll() {
     try {
        // hard-coded SQL select with no arguments
        String mySQL = "select * from tblAMIGO";
        Cursor c1 = db.rawQuery(mySQL, null);

        txtMsg.append("\n-useRawQueryShowAll" + showCursor(c1) );
```

Kompletny kod prezentowanych przykładów

```
        } catch (Exception e) {
            txtMsg.append("\nError useRawQuery1: " + e.getMessage());

        }
    }// useRawQuery1

    // ////////////////////////////////////////////////////////////
    private String showCursor( Cursor cursor) {
        // show SCHEMA (column names & types)
        cursor.moveToPosition(-1); //reset cursor's top
        String cursorData = "\nCursor: [";

        try {
            // get column names
            String[] colName = cursor.getColumnNames();
            for(int i=0; i<colName.length; i++){
                String dataType = getColumnType(cursor, i);
                cursorData += colName[i] + dataType;

                if (i<colName.length-1){
                    cursorData+= ", ";
                }
            }
        } catch (Exception e) {
            Log.e( "<<SCHEMA>>" , e.getMessage() );
        }
        cursorData += "]“;
```

Kompletny kod prezentowanych przykładów

```
        // now get the rows
        cursor.moveToPosition(-1); //reset cursor's top
        while (cursor.moveToNext()) {
            String cursorRow = "\n[";
            for (int i = 0; i < cursor.getColumnCount(); i++) {
                cursorRow += cursor.getString(i);
                if (i<cursor.getColumnCount()-1)
                    cursorRow +=  ", ";
            }
            cursorData += cursorRow + "]";
        }
        return cursorData + "\n";
    }
    // ////////////////////////////////////////////////////////////////////////
    private String getColumnType(Cursor cursor, int i) {
        try {
            //peek at a row holding valid data
            cursor.moveToFirst();
            int result = cursor.getType(i);
            String[] types = {":NULL", ":INT", ":FLOAT", ":STR", ":BLOB", ":UNK" };
            //backtrack - reset cursor's top
            cursor.moveToPosition(-1);
            return types[result];
        } catch (Exception e) {
            return " ";
        }
    }
```

Bazy danych

Kompletny kod prezentowanych przykładów

```
private void useRawQuery1() {
   try {
      // hard-coded SQL select with no arguments
      String mySQL = "select * from tblAMIGO";
      Cursor c1 = db.rawQuery(mySQL, null);

      // get the first recID
      c1.moveToFirst();
      int index = c1.getColumnIndex("recID");
      int theRecID = c1.getInt(index);

      txtMsg.append("\n-useRawQuery1 - first recID   " + theRecID);
      txtMsg.append("\n-useRawQuery1" + showCursor(c1) );

   } catch (Exception e) {
      txtMsg.append("\nError useRawQuery1: " + e.getMessage());

   }
}// useRawQuery1
// ////////////////////////////////////////////////////////////////
private void useRawQuery2() {
   try {
      // use: ? as argument's placeholder

      String mySQL = " select recID, name, phone "
                + " from tblAmigo "
                + " where recID > ? " + "    and name   = ? ";
      String[] args = { "1", "BBB" };
```

Bazy danych

Kompletny kod prezentowanych przykładów

```
        Cursor c1 = db.rawQuery(mySQL, args);

        // pick NAME from first returned row
        c1.moveToFirst();
        int index = c1.getColumnIndex("name");
        String theName = c1.getString(index);

        txtMsg.append("\n-useRawQuery2 Retrieved name: " + theName);
        txtMsg.append("\n-useRawQuery2 " + showCursor(c1) );

    } catch (Exception e) {
        txtMsg.append("\nError useRawQuery2: " + e.getMessage());
    }
}// useRawQuery2

// ////////////////////////////////////////////////////////////////
private void useRawQuery3() {
    try {
        // arguments injected by manual string concatenation
        String[] args = { "1", "BBB" };

        String mySQL = " select recID, name, phone"
                    + "   from tblAmigo "
                    + "  where recID > "  + args[0]
                    + "    and name  = '" + args[1] + "'";

        Cursor c1 = db.rawQuery(mySQL, null);
```

Kompletny kod prezentowanych przykładów

```
        // pick PHONE from first returned row
        int index = c1.getColumnIndex("phone"); //case sensitive
        c1.moveToNext();
        String thePhone = c1.getString(index);

        txtMsg.append("\n-useRawQuery3 - Phone: " + thePhone);
        txtMsg.append("\n-useRawQuery3   " + showCursor(c1) );

    } catch (Exception e) {
        txtMsg.append("\nError useRawQuery3: " + e.getMessage());

    }
}// useRawQuery3

 // ////////////////////////////////////////////////////////////////
private void useSimpleQuery1() {
    try {
        // simple-parametric query on one table.
        // arguments: tableName, columns, condition, cond-args,
        //            groupByCol, havingCond, orderBy
        // the next parametric query is equivalent to SQL stmt:
        //    select recID, name, phone from tblAmigo
        //    where recID > 1 and length(name) >= 3
        //    order by recID
```

Kompletny kod prezentowanych przykładów

```
        Cursor c1 = db.query(
              "tblAMIGO",
              new String[] { "recID", "name", "phone" },
              "recID > 1 and length(name) >= 3 ",
              null,
              null,
              null,
              "recID");

        // get NAME from first data row
        int index = c1.getColumnIndex("phone");
        c1.moveToFirst();
        String theName = c1.getString(index);

        txtMsg.append("\n-useSimpleQuery1 - Total rec " + theName);
        txtMsg.append("\n-useSimpleQuery1 " + showCursor(c1) );

    } catch (Exception e) {
        txtMsg.append("\nError useSimpleQuery1: " + e.getMessage());
    }
}// useSimpleQuery1
```

Kompletny kod prezentowanych przykładów

```
private void useSimpleQuery2() {
   try {
      // nontrivial 'simple query' on one table
      String[] selectColumns = { "name", "count(*) as TotalSubGroup" };
      String whereCondition = "recID >= ?";
      String[] whereConditionArgs = { "1" };
      String groupBy = "name";
      String having = "count(*) <= 4";
      String orderBy = "name";

      Cursor c1 = db.query("tblAMIGO", selectColumns, whereCondition,
            whereConditionArgs, groupBy, having, orderBy);

      int theTotalRows = c1.getCount();
      txtMsg.append("\n-useSimpleQuery2 - Total rec: " + theTotalRows);
      txtMsg.append("\n-useSimpleQuery2 " + showCursor(c1) );

   } catch (Exception e) {
      txtMsg.append("\nError useSimpleQuery2: " + e.getMessage());

   }
}// useSimpleQuery2
```

Bazy danych

Kompletny kod prezentowanych przykładów

```
private void showTable(String tableName) {
   try {
      String sql = "select * from " + tableName ;
      Cursor c = db.rawQuery(sql, null);
      txtMsg.append("\n-showTable: " + tableName + showCursor(c) );

   } catch (Exception e) {
      txtMsg.append("\nError showTable: " + e.getMessage());

   }
}// useCursor1

// ////////////////////////////////////////////////////////////////
private void useCursor1() {
   try {
      // this is similar to showCursor(...)
      // obtain a list of records[recId, name, phone] from DB
      String[] columns = { "recID", "name", "phone" };
      // using simple parametric cursor
      Cursor c = db.query("tblAMIGO", columns, null, null, null, null,
                          "recID");
      int theTotal = c.getCount();
      txtMsg.append("\n-useCursor1 - Total rec " + theTotal);
      txtMsg.append("\n");
      int idCol = c.getColumnIndex("recID");
      int nameCol = c.getColumnIndex("name");
      int phoneCol = c.getColumnIndex("phone");
```

Kompletny kod prezentowanych przykładów

```
        c.moveToPosition(-1);
        while (c.moveToNext()) {
            columns[0] = Integer.toString((c.getInt(idCol)));
            columns[1] = c.getString(nameCol);
            columns[2] = c.getString(phoneCol);

            txtMsg.append(columns[0] + " " + columns[1] + " " + columns[2]
                  + "\n");
        }

    } catch (Exception e) {
        txtMsg.append("\nError useCursor1: " + e.getMessage());
        finish();
    }
}// useCursor1

// ////////////////////////////////////////////////////////////////
private void updateDB() {
    // action query performed using execSQL
    // add 'XXX' to the name of person whose phone is 555-1111
    txtMsg.append("\n-updateDB");

    try {
        String thePhoneNo = "555-1111";
        db.execSQL(" update tblAMIGO set name = (name || 'XXX') "
                + " where phone = '" + thePhoneNo + "' ");
        showTable("tblAmigo");
```

Bazy danych

Kompletny kod prezentowanych przykładów

```
        } catch (Exception e) {
            txtMsg.append("\nError updateDB: " + e.getMessage());
        }
        useCursor1();
    }

    // ////////////////////////////////////////////////////////////////
    private void dropTable() {
        // (clean start) action query to drop table
        try {
            db.execSQL(" drop table tblAmigo; ");
            // >>Toast.makeText(this, "Table dropped", 1).show();
            txtMsg.append("\n-dropTable - dropped!!");
        } catch (Exception e) {
            txtMsg.append("\nError dropTable: " + e.getMessage());
            finish();
        }
    }
    // ////////////////////////////////////////////////////////////////
    public void useInsertMethod() {
        // an alternative to SQL "insert into table values(...)"
        // ContentValues is an Android dynamic row-like container
        try {
            ContentValues initialValues = new ContentValues();
            initialValues.put("name", "ABC");
            initialValues.put("phone", "555-4444");
            int rowPosition = (int) db.insert("tblAMIGO", null, initialValues);
```

Kompletny kod prezentowanych przykładów

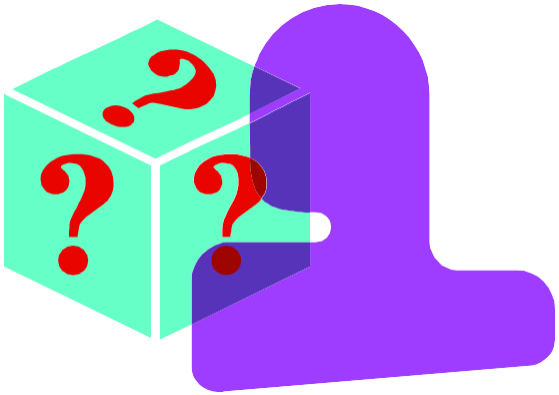
```
            txtMsg.append("\n-useInsertMethod rec added at: " + rowPosition);
            showTable("tblAmigo");

        } catch (Exception e) {
            txtMsg.append("\n-useInsertMethod - Error: " + e.getMessage());
        }
    }// useInsertMethod

    // ////////////////////////////////////////////////////////////////
    private void useUpdateMethod() {
        try {
            // using the 'update' method to change name of selected friend
            String[] whereArgs = { "1" };

            ContentValues updValues = new ContentValues();
            updValues.put("name", "Maria");

            int recAffected = db.update("tblAMIGO", updValues,
                    "recID = ? ", whereArgs);

            txtMsg.append("\n-useUpdateMethod - Rec Affected " + recAffected);
            showTable("tblAmigo");

        } catch (Exception e) {
            txtMsg.append("\n-useUpdateMethod - Error: " + e.getMessage() );
        }
    }
```

Kompletny kod prezentowanych przykładów

```
   private void useDeleteMethod() {
      // using the 'delete' method to remove a group of friends
      // whose id# is between 2 and 7

      try {
         String[] whereArgs = { "2" };

         int recAffected = db.delete("tblAMIGO", "recID = ?",
               whereArgs);

         txtMsg.append("\n-useDeleteMethod - Rec affected " + recAffected);
         showTable("tblAmigo");

      } catch (Exception e) {
         txtMsg.append("\n-useDeleteMethod - Error: " + e.getMessage());
      }
   }

}// class
```

Bazy danych

Pytania

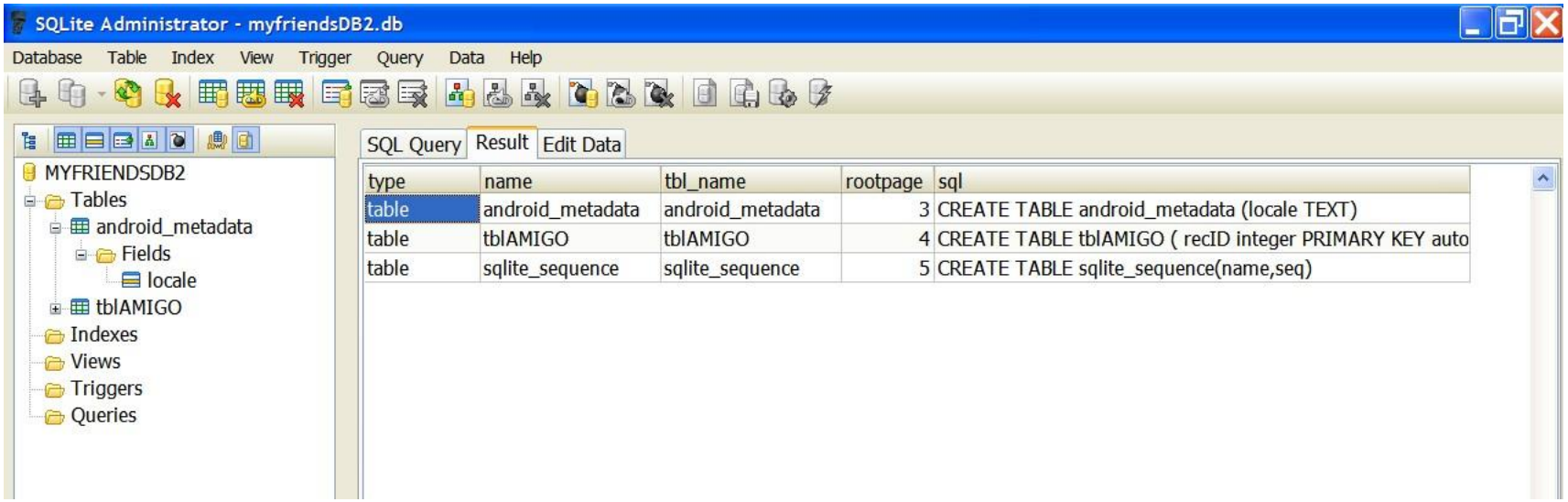


Dodatek 1: Tabela główna SQLite

Można odpytywać tabelę główną SQLite (nazwaną: ***sqlite_master***) by odpytać SZBD o tabelę, indeks czy inny zasób bazodanowany.

Przykład

```
select * from sqlite_master;
```



Dodatek 1: Tabela główna SQLite

Poniżej zaprezentowano kod Javy, który umożliwia odpytanie SZBD o istnienie danego zasobu (np. tabeli).

```
public boolean tableExists(SQLiteDatabase db, String tableName)
{
   //true if table exists, false otherwise
   String mySql = "SELECT  name FROM sqlite_master "
                + " WHERE  type='table'              "
                + "    AND name='" + tableName  + "'";

   int resultSize = db.rawQuery(mySql, null).getCount();

   if (resultSize ==0) {
      return true;
   } else
      return false;
}
```

Dodatek 2: Przydatna funkcja

Mechanizm typu „usuń tylko jeśli istnieje” można zrealizować za pomocą polecenia:

```
db.execSQL(" DROP TABLE IF EXISTS tblAmigo; ");
```