

Local Databases and SQL

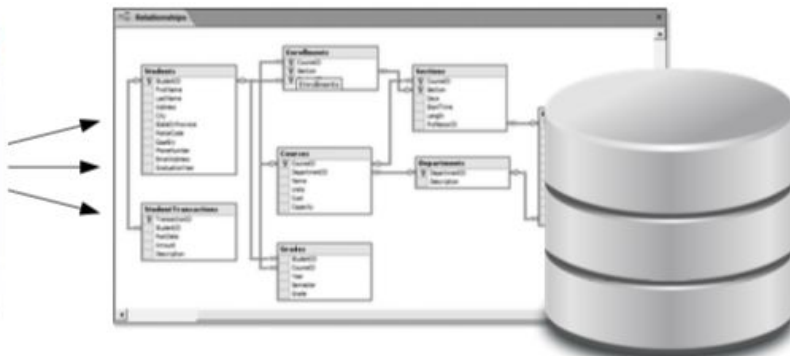
...

Lecture 11

What is a database?

- **relational database**: Data structured into associated **tables**.
 - think of an Excel worksheet as a table
 - a database is a collection of one or more tables, along with support for efficient operations
 - common operations: "CRUD" (create, read, update, dele~~t~~e); fast search
 - a table **row** corresponds to a unit of data called a record; a **column** corresponds to an attribute of that record

A	B	C	D	E
Comp employment	Contract Types & Dates	Total advance		
Editor Designers	Commercial	7084.41		
Law Plumbing - C	Commercial	175		
Motion, animat	Flows To Go Dental Care	8389.6		
Motion, Video	Law Plumbing	0		
Patrol Road Ertals	Commercial	20800.96		
Reyes Properties C	Commercial	13400.36		
Ruben, Dario	Commercial	1865		
Smethon, Fran	Commercial	0		
Smith, Lee	Commercial	0		



Where is the data?

- A database can be located in many places.
 - within your Android device (a "local database")
 - on a remote web server
 - spread throughout many remote servers ("in the cloud")
 - ...
- Today we will learn to create and use **local databases**.



Talking to a database

- **SQL** (Structured Query Language): relational databases typically use SQL to define, manage, and search data
 - a declarative language syntax that can be used in many situations

```
SELECT name
FROM   countries
WHERE  population > 20000000;
```

code	name	continent	independence_year	population	gnp	head_of_state	...
AFG	Afghanistan	Asia	1919	22720000	5976.0	Mohammad Omar	...
NLD	Netherlands	Europe	1581	15864000	371362.0	Beatrix	...
...

countries (Other columns: region, surface_area, life_expectancy, gnp_old, local_name, government_form, capital, codes)

id	name	country_code	district	population	country_code	language	official	percentage
3793	New York	USA	New York	8008278	AFG	Pashto	T	52.4
1	Los Angeles	USA	California	3694820	NLD	Dutch	T	95.6
...

cities

languages

Why use a database?

- **powerful:** can search, filter, combine data from many sources
- **fast:** can search/filter a database very quickly compared to a file
- **big:** scale well up to very large data sizes
- **safe:** built-in mechanisms for failure recovery (transactions)
- **multi-user:** concurrency features let many users view/edit data at same time
- **abstract:** layer of abstraction between stored data and app(s)
- **common syntax:** database programs use same SQL commands

Some database software

- **Oracle**
- **Microsoft**
 - **SQL Server** (powerful)
 - **Access** (simple)
- **PostgreSQL**
 - powerful/complex free open-source database system
- **SQLite**
 - transportable, lightweight free open-source database system
- **MySQL**
 - simple free open-source database system
 - many servers run "LAMP" (Linux, Apache, MySQL, and PHP)



Example database table 1

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

students

id	name	teacher_id
10001	Computer Science 142	1234
10002	Computer Science 143	5678
10003	Computer Science 190M	9012
10004	Informatics 100	1234

courses

id	name
1234	Krabappel
5678	Hoover
9012	Stepp

teachers

student_id	course_id	grade
123	10001	B-
123	10002	C
456	10001	B+
888	10002	A+
888	10003	A+
404	10004	D+

grades

Example database table 2

code	name	continent	independence_year	population	gnp	head_of_state	...
AFG	Afghanistan	Asia	1919	22720000	5976.0	Mohammad Omar	...
NLD	Netherlands	Europe	1581	15864000	371362.0	Beatrix	...
...

countries (Other columns: region, surface_area, life_expectancy, gnp_old, local_name, government_form, capital, code2)

id	name	country_code	district	population
3793	New York	USA	New York	8008278
1	Los Angeles	USA	California	3694820
...

cities

country_code	language	official	percentage
AFG	Pashto	T	52.4
NLD	Dutch	T	95.6
...

languages

Example database table 3

id	first_name	last_name	gender
433259	William	Shatner	M
797926	Britney	Spears	F
831289	Sigourney	Weaver	F
...			

actors

movie_id	genre
209658	Comedy
313398	Action
313398	Sci-Fi
...	

movies_genres

id	first_name	last_name
24758	David	Fincher
66965	Jay	Roach
72723	William	Shatner
...		

directors

id	name	year	rank
112290	Fight Club	1999	8.5
209658	Meet the Parents	2000	7
210511	Memento	2000	8.7
...			

movies

director_id	movie_id
24758	112290
66965	209658
72723	313398
...	

movies_directors

actor_id	movie_id	role
433259	313398	Capt. James T. Kirk
433259	407323	Sgt. T.J. Hooker
797926	342189	Herself
...		

roles

SQL ([link](#))

```
SELECT name FROM cities WHERE id = 17;  
INSERT INTO countries VALUES ('SLD', 'ENG', 'T', 100.0);
```

- **Structured Query Language (SQL):** a language for searching and updating a database
 - a standard syntax that is used by all database software (*with minor incompatibilities*)
 - generally case-insensitive
- a **declarative language:** describes what data you are seeking, not exactly how to find it

SELECT statement

```
SELECT column(s) FROM table WHERE condition;
```

```
SELECT name, population FROM cities  
        WHERE country_code = "FSM";
```

- searches a database and returns a set of results
 - column name(s) after SELECT filter which parts of rows are returned
 - table and column names are **case-sensitive**
 - SELECT DISTINCT removes any duplicates
 - SELECT * keeps all columns
- WHERE clause filters out rows based on columns' data values
 - in large databases, WHERE clause is critical to reduce result set size

Android SQLiteDatabase ([link](#))

```
SQLiteDatabase db = openOrCreateDatabase(  
    "name", MODE_PRIVATE, null);  
db.execSQL("SQL query");
```

- The `openOrCreateDatabase` method either creates a new empty database with that name or opens an existing one
 - once opened, you can use methods to execute SQL commands:
 - `rawQuery` - if your query **returns** results (e.g. `SELECT`)
 - `execSQL` - if your query does not return results (e.g. `INSERT`, `DELETE`)
 - If query has invalid SQL, throws an `SQLException`
 - SQLite databases get saved to `/data/data/packageName/databases/`
 - (but you should never need to manipulate them as files directly)

SQLiteDatabase methods ([link](#))

Method	Description
<code>db.beginTransaction();</code> <code>db.endTransaction();</code>	methods for "transactions", which are a series of SQL commands that can be run as a group
<code>db.delete("table", "whereClause", args);</code>	delete rows from a table
<code>db.deleteDatabase(file);</code>	delete an entire database <i>(be careful)</i>
<code>db.execSQL("query");</code>	run a query that doesn't return any results (e.g. insert, delete, update, etc.)
<code>db.insert("table", null, values);</code>	insert rows into a database table
<code>db.query(...);</code>	(we suggest <code>rawQuery</code> instead)
<code>db.rawQuery("SQL query", args)</code>	perform the given SQL SELECT query and return a Cursor to view the results
<code>db.replace("table", null, values);</code>	replace rows in a database table
<code>db.update("table", values, "whereClause", args);</code>	update existing rows in a database table

Concept of a Cursor

- The Android SQLite API returns an object called a *Cursor* that allows you to iterate through the results of a SELECT query.
 - Similar to the concept of an *iterator*
- Like a pointer positioned to a given row from the set of results.
 - You can move the cursor forward to the next result row.
 - You can ask the cursor for values of columns of its "current" row.

```
SELECT id, email FROM students;
```

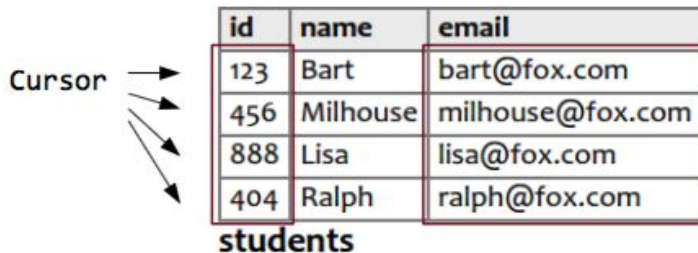
Cursor →

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

students

Cursor example ([link](#))

```
// Cursor iterates through row results one at a time
Cursor cr = db.rawQuery(
    "SELECT id, email FROM students", null);
if (cr.moveToFirst()) {
    do {
        int id = cr.getInt(cr.getColumnIndex("id"));
        String email = cr.getString(cr.getColumnIndex("email"));
        ...
    } while (cr.moveToNext());
    cr.close();
}
```



The diagram illustrates a database cursor pointing to the first row of a table. On the left, the word "Cursor" is written. Four arrows originate from it: the top arrow points to the first row (123, Bart, bart@fox.com), and the other three arrows point downwards to the subsequent rows (456, Milhouse, milhouse@fox.com), (888, Lisa, lisa@fox.com), and (404, Ralph, ralph@fox.com). The table itself has three columns: "id", "name", and "email". The first row is highlighted with a red border, indicating it is the current row being processed by the cursor. The table is labeled "students" at the bottom.

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

students

Cursor methods ([link](#))

Method	Description
<code>c.close();</code>	call this when done with the cursor
<code>c.getColumnIndex("name")</code>	index of a column based on its name
<code>c.getColumnName(index)</code>	name of column based on its index
<code>c.getCount()</code>	number of rows in result
<code>c.getDouble(index), c.getBlob(index), c.getFloat(index), c.getInt(index), c.getLong(index), c.getString(index)</code>	get data from a column
<code>c.isBeforeFirst()</code> <code>c.isFirst()</code> <code>c.isLast()</code>	ask about cursor's position
<code>c.moveToFirst();</code> <code>c.moveToLast();</code> <code>c.moveToNext();</code> <code>c.moveToPosition(index);</code>	tell cursor to move to a given position (each returns boolean indicating success)

Importing a .sql file

- A .sql file contains a sequence of SQL commands.
 - Common format for exporting an entire database and its contents.
 - Used to save a backup or restore db to another server.
- To import a .sql file into an Android app:
 - Put the .sql file into your app's res/raw folder
 - Open it with a Scanner
 - Read lines until you find a semicolon
 - Run the string you read as a query using execSQL
 - Repeat
 - ...

Import .sql example

```
// read example.sql into database named "example"
SQLiteDatabase db = context.openOrCreateDatabase("example");
Scanner scan = new Scanner(getResources()
                                .openRawResource(R.raw.example));

String query = "";
while (scan.hasNextLine()) {    // build and execute queries
    query += scan.nextLine() + "\n";
    if (query.trim().endsWith(";")) {
        db.execSQL(query);
        query = "";
    }
}
```

WHERE clauses

```
SELECT name, gnp FROM countries WHERE gnp > 2000000;
```

```
SELECT * FROM cities WHERE code = 'USA'  
AND population >= 2000000;
```

```
SELECT code, name, population FROM countries  
WHERE name LIKE 'United%';
```

- WHERE clause can use the following operators:

=, >, >=, <, <=

<> : not equal (some systems support !=)

BETWEEN *min* AND *max*

LIKE *pattern* (put % on ends to search for prefix/suffix/substring)

IN (*value, value, ..., value*)

condition1 AND *condition2* ; *condition1* OR *condition2*

ORDER BY, LIMIT

```
SELECT code, name, population FROM countries  
WHERE name LIKE 'United%' ORDER BY population;
```

```
SELECT * FROM countries ORDER BY population DESC, gnp;
```

```
SELECT name FROM cities WHERE name LIKE 'K%' LIMIT 5;
```

- ORDER BY sorts in ascending (default) or descending order
 - can specify multiple orderings in decreasing order of significance
- LIMIT gets first N results of the query
 - useful as a sanity check to make sure query doesn't return 10^7 rows

INSERT and REPLACE

```
INSERT INTO table (columnName, ..., columnName)  
VALUES (value, value, ..., value);
```

```
REPLACE INTO table (columnName, ..., columnName)  
VALUES (value, value, ..., value) WHERE columnName = value;
```

```
INSERT INTO students (name, email)  
VALUES ("Lewis", "lewis@fox.com");
```

```
REPLACE INTO students (id, name, email)  
VALUES (789, "Martin", "prince@fox.com")  
WHERE id = 789;
```

- columns can have default or automatic values (such as IDs)
- omitting them from the INSERT statement uses the defaults
- REPLACE is like INSERT but modifies an existing row

INSERT with SQLiteDatabase

```
// use execSQL instead of rawQuery, because no results
String query = "INSERT INTO students (name, email) "
               + "VALUES ('Lewis', 'lewis@fox.com')";
db.execSQL(query);

String query2 = "REPLACE INTO students (id, name, email) "
               + "VALUES (789, 'Martin', 'prince@fox.com') "
               + "WHERE ID = 789";
db.execSQL(query2);
```

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

students

ContentValues ([link](#))

```
// alternative syntax using insert method
String name = "Lewis";
String email = "lewis@fox.com";

db.execSQL("INSERT INTO students (name, email) "
+ "VALUES ('" + name + "', '" + email + "')");

// use ContentValues to store values to put in row
ContentValues cvalues = new ContentValues();
cvalues.put("name", name);
cvalues.put("email", email);
db.insert("students", null, cvalues);
```

- ContentValues can be optionally used as a level of abstraction for statements like INSERT, UPDATE, REPLACE
 - meant to provide cleaner Java syntax rather than raw SQL syntax

UPDATE

```
UPDATE table
SET column1 = value1,
    ...,
    columnN = valueN
WHERE condition;
```

```
UPDATE students SET email = "lisasimpson@gmail.com"
WHERE id = 888;
```

- modifies an existing row(s) in a table
- Be careful! If you omit WHERE clause, it modifies ALL rows

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

students

UPDATE with SQLiteDatabase

```
// an update statement using execSQL
String query = "UPDATE students "
               + "SET email = 'lisasimpson@gmail.com' "
               + "WHERE id = 888";

db.execSQL(query);

// alternative syntax using ContentValues
ContentValues cvalues = new ContentValues();
cvalues.put("email", "lisasimpson@gmail.com");
db.update("students", cvalues, "id = 888", null);
```

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

students

DELETE

DELETE FROM *table* WHERE *condition*;

DELETE FROM students WHERE id = 888;

- removes existing row(s) in a table
- can be used with other syntax like LIMIT, LIKE, ORDER BY, etc.
- Be careful! If you omit WHERE clause, it deletes ALL rows

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

students

Delete with SQLiteDatabase

```
// an update statement using execSQL
String query = "DELETE FROM students "
               + "WHERE id = 888";
db.execSQL(query);

// alternative syntax using delete method
db.delete("students", "id = 888", null);
```

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

students

Useful Library for data visualization

- GraphView library information can be found at:

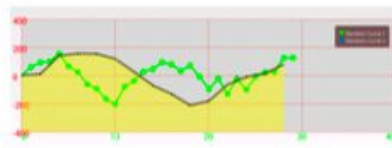
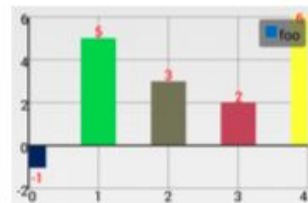
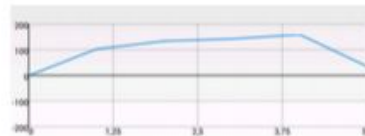
- <http://www.android-graphview.org/>

- add it to your build.gradle:

```
dependencies {  
    ...  
    compile 'com.jjoe64:graphview:4.2.1'  
}
```

- create a graph in your activity XML

```
<com.jjoe64.graphview.GraphView  
    android:layout_width="width"  
    android:layout_height="height"  
    android:id="@+id/id" />
```



Line Graphs in GraphView

- talk to a graph in Java:

```
GraphView graph = (GraphView) findViewById(R.id.id);  
graph.setTitle("title");
```

```
// set X bounds; Y is the same idea
```

```
graph.getViewPort().setXAxisBoundsManual(true);  
graph.getViewPort().setMinX(minX);  
graph.getViewPort().setMaxX(maxX);  
...
```

- add a line to the graph:

```
LineGraphSeries<DataPoint> series = new LineGraphSeries<>();  
DataPoint point = new DataPoint(x, y);  
series.appendData(point, false, maxPoints);  
...  
graph.addSeries(series);
```


Creating tables

```
CREATE TABLE IF NOT EXISTS name (  
    columnName type constraints,  
    ...  
    columnName type constraints  
);  
DROP TABLE name;
```

```
CREATE TABLE students (  
    id INTEGER,  
    name VARCHAR(20),  
    email VARCHAR(32),  
    password VARCHAR(16)  
);
```

INTEGER	32-bit integer
REAL or DOUBLE	real number
VARCHAR(<i>Length</i>)	string up to given length
BLOB	binary data

- all columns' names and types must be listed (*see table above*)

Table column constraints

```
CREATE TABLE students (  
  id INTEGER NOT NULL PRIMARY KEY,  
  name VARCHAR(20) NOT NULL,  
  email VARCHAR(32),  
  password VARCHAR(16) NOT NULL DEFAULT '12345'  
);
```

-
- NOT NULL: empty value not allowed in any row for that column
 - PRIMARY KEY / UNIQUE: no two rows can have the same value
 - DEFAULT **value**: if no value is provided, use the given default

Create with SQLiteDatabase

```
// a create table statement using execSQL
String query = "CREATE TABLE students ( "
    + "  id INTEGER PRIMARY KEY, "
    + "  name VARCHAR(20) NOT NULL, "
    + "  email VARCHAR(32) NOT NULL, "
    + "  password VARCHAR(16) NOT NULL "
    + ")";
db.execSQL(query);
```

id	name	email	password

Modifying existing tables

```
ALTER TABLE name RENAME TO newName;
```

```
ALTER TABLE name  
ADD COLUMN columnName type constraints;
```

```
ALTER TABLE name DROP COLUMN columnName;
```

```
ALTER TABLE name  
CHANGE COLUMN oldColumnName newColumnName type constraints;
```

-
- SQL has many commands for modifying existing data
 - the above is not a complete reference

Related tables

id	name	email
123	Bart	bart@fox.com
456	Milhouse	milhouse@fox.com
888	Lisa	lisa@fox.com
404	Ralph	ralph@fox.com

students

student_id	course_id	grade
123	10001	B-
123	10002	C
456	10001	B+
888	10002	A+
888	10003	A+
404	10004	D+

grades

id	name	teacher_id
10001	Computer Science 142	1234
10002	Computer Science 143	5678
10003	Computer Science 190M	9012
10004	Informatics 100	1234

courses

id	name
1234	Krabappel
5678	Hoover
9012	Stepp

teachers

- **primary key:** column guaranteed to be unique for each row (ID)
- **normalizing:** splitting tables to improve structure / redundancy

JOIN

```
SELECT column(s) FROM table1 name1  
        JOIN table2 name2 ON condition(s)  
        ...  
        JOIN tableN nameN ON condition(s)  
WHERE condition;
```

```
SELECT name, course_id, grade  
FROM students s  
JOIN grades g ON s.id = g.student_id  
WHERE s.name = 'Bart';
```

- JOIN combines related records from two or more tables
 - ON clause specifies which records from each table are matched
 - rows are often linked by their key columns ('id')
 - joins can be tricky to understand; out of scope of this course