CS-A1150 Tietokannat

Osa laukaisimia käsittelevistä kalvoista perustuu kurssin oppikirjaan

29.4.2019

Oppimistavoitteet: tämän luennon jälkeen

- Osaat määritellä SQL:ssä laukaisimia, jotka reagoivat johonkin tietokannassa määriteltyyn tapahtumaan.
- Tiedät, miten SQL-käskyjä voidaan liittää toisella ohjelmointikielellä kirjoitetettuun ohjelmaan.
- Olet tutustunut kolmeen eri tapaan käsitellä SQL-tietokantoja toisella ohjelmointikielellä kirjoitetusta ohjelmassa:
  - ORM (object-relational mapping) (hyvin lyhyesti)
  - Sopivan kirjaston ja sen funktioiden käyttäminen
  - SQL-käskyjen kirjoittaminen ohjelmaan suoraan (embedded SQL) (hyvin lyhyesti).

Laukaisimet

- Laukaisimien avulla siirretään osa tarvittavista tarkastuksista ja niiden aiheuttamista toimenpiteistä sovellusohjelmasta tietokannan hallintajärjestelmän hoidettavaksi.
- Laukaisimia (engl. triggers) kutsutaan *Event-Condition-Action* eli *ECA*-säännöiksi: Kun *Event* tapahtuu, tarkista, onko *Condition* tosi. Jos on, suorita *Action*.
- Laukaisimet poikkeavat aikaisemmin esitetyistä eheysrajoitteista kolmella tavalla:
  1. Laukaisimet testataan ainoastaan tiettyjen tietokantaohjelmoijan määrittelemien tapahtumien sattuessa (yleensä määrättyyn tauluun kohdistuva lisäys, poisto tai muutos).
  2. Laukaisin testaa siihen liitetyn ehdon. Jos se ei ole voimassa, niin mitään laukaisimeen liittyvää toimintaa ei suoriteta.
  3. Jos laukaisimeen liittyvä ehto on voimassa, niin tietokannan hallintajärjestelmä suorittaa laukaisimeen liittyvän toiminnan, joka voi olla periaatteessa olla mikä tahansa tietokantaoperaatio.

Laukaisimet (jatkoa)

- Ohjelmoija voi määritellä, että toimintaosa suoritetaan joko
  - kertaalleen jokaisen päivytetyn monikon osalta (rivitason laukaisin, row-level trigger) tai
  - kertaalleen kaikkien päivitettyjen monikoiden osalta (lausetason laukaisin, statement-level trigger).
- Eri tietokannanhallintajärjestelmissä on suuria eroja sillä, millaisia laukaisimia niissä oikeasti pystyy määrittelmään ja mikä on käytettävä syntaksi.
- Tällä luennolla ei selitetä perusteellisesti kaikkia laukaisimiin liittyviä asioita, vaan näytetään periaatteita esimerkkien. Lisätietoa voi hakea oppikirjasta tai SQL-dokumentaatiosta.

Esimerkki rivitason laukaisimesta

- Tarkastellaan esimerkkinä taulua

  `Employee(ssNo, name, vacancy, salary)`

  Kirjoitetaan laukaisin, joka estää työntekijän palkan alentamisen (ei tällainen SQLitessä).

```
CREATE TRIGGER salaryTrigger
AFTER UPDATE OF salary ON Employee
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.salary > NewTuple.salary)
    UPDATE Employee
    SET salary = OldTuple.salary
    WHERE ssNO = NewTuple.ssNO;
```

Esimerkin selityksiä

- Rivillä 1 aloitetaan laukaisimen määrittely ja annetaan sille nimi.
- Rivillä 2 kerrotaan, että laukaisin suoritetaan silloin, jos jonkin *Employee*-taulun monikon attribuutin `salary` arvoa päivitetään.
- Riveillä 3–5 kerrotaan, että nimellä `OldTuple` tarkoitetaan päivityksen kohteena ollutta monikkoa ennen päivitystä ja nimellä `NewTuple` päivityksen kohteena ollutta monikkoa päivityksen jälkeen.
- Rivi 6 FOR EACH ROW määrittelee, että laukaisin suoritetaan kerran jokaista päivitettyä monikkoa kohti.
- Rivi 7 on laukaisimen ehto: riviä seuraavat käskyt suoritetaan vain silloin, jos tällä rivillä oleva ehto (`salary`-attribuutin arvo ennen päivitystä on suurempi kuin `salary`-attribuutin arvo päivityksen jälkeen) on tosi.
- Rivit 8–10 sisältävät käskyn, joka suoritetaan silloin, jos ehto on tosi. Periaatteessa UPDATE-käsky suoritetaan koko *Empoyee*-taululle, mutta WHERE-osassa oleva ehto rajoittaa päivityksen koskemaan vain juuri muutettua monikkoa.

Esimerkki lausetason laukaisimesta

- Ei halutakaan estää yksittäisen palkan laskemista edellisen kalvon `Employee`-taulussa, vaan sen sijaan halutaan pitää huoli siitä, että palkkojen päivitysten yhteydessä kaikkien työntekijöiden keskipalkka ei saa pudota alle 2500:n.
- Jos jokin taulussa olevia monikoita päivittävä SQL-käsky (joka voi sisältää useiden monikoiden päivityksen) aiheuttaisi palkkojen keskiarvon putoamisen alle 2500:n, perutaan kaikki käskyn sisältämät päivitykset.
- Laukaisinta ei nyt suoriteta erikseen jokaiselle päivitetylle riville, vaan koko päivityskäskyn suorittamisen jälkeen tarkistetaan haluttu ehto, ja jos se ei toteudu, perutaan yhdellä käskyllä kaikki tehdyt päivitykset.
- Nyt ei voida viitata yksittäisiin päivitettäviin monikoihin OLD ROW ja NEW ROW -ilmauksilla, vaan ilmaus OLD TABLE viittaa relaatioon, joka sisältää vanhat versiot kaikista tapahtuman vaikuttamista monikoista ja NEW TABLE viittaa relaatioon, joka sisältää uudet versiot kaikista niistä monikoista, johon tapahtuma vaikutti.

Esimerkki lausetason laukaisimesta, jatkoa

- Edellisen kalvon laukaisin (ei toimi SQLitessä):

```
CREATE TRIGGER AvgSalaryTrigger
AFTER UPDATE OF salary ON Employee
REFERENCING
    OLD TABLE AS OldStuff
    NEW TABLE AS NewStuff
FOR EACH STATEMENT
WHEN (2500 > SELECT AVG(salary) FROM Employee)
BEGIN
    DELETE FROM Employee
    WHERE (ssNo, name, vacancy, salary) IN NewStuff;
    INSERT INTO Employee
        (SELECT * FROM OldStuff);
END;
```

Esimerkin selityksiä

- Rivit 1 ja 2 ovat vastaavat kuin edellisessä esimerkissä.
- Riveillä 3–5 määritellään, että nimellä `OldStuff` viitataan relaatioon, joka sisältää vanhat versiot päivityksen vaikuttamista monikoista, ja nimellä `NewStuff` relaatioon, joka sisältää uudet versiot päivytyksen vaikuttamista monikoista. (Kummassakaan relaatiossa ei ole niitä monikoita, joita ei ole päivitetty.)
- Rivin 6 FOR EACH STATEMENT kertoo, että laukaisin suoritetaan kerran jokaista suoritettua rivin 2 mukaista SQL-käskyä kohti (ei siis jokaista muutettua riviä kohti).
- Rivi 7 sisältää ehdon (päivityksen jälkeen palkkojen keskiarvo koko relaatiossa on alle 2500), jonka ollessa tosi seuraavat käskyt suoritetaan.
- Riveillä 8–13 olevat käskyt suoritetaan, jos edellinen ehto on tosi: Taulusta *Employee* poistetaan ensin kaikki muutetut monikot. Sen jälkeen tauluun lisätään näiden monikoiden vanhat versiot taulusta `OldStuff`.

Lisää laukaisimen määrittelystä

- Laukaisimen määrittelyssä on useita eri mahdollisuuksia siihen, miten tapahtuma- (Event), ehto- (Condition) ja toimintaosat (Action parts) liittyvät toisiinsa:
  1. Ehdon tarkistaminen ja toimintaosa voidaan suorittaa ennen tapahtumaosaa tai tapahtumaosan suorittamisen jälkeen. Toimintaosa voidaan myös suorittaa tapahtumaosan sijaan.
  2. Ehto ja toimintaosa voi kohdistua tapahtumaosan käsittelemien (lisäämien, poistamien, muuttamien) monikoiden vanhoihin tai uusiin arvoihin.
  3. Jos tapahtuma on päivittävä, niin tapahtumaosa voi kohdistua yhteen tai useampaan sarakkeeseen.

Huomautuksia laukaisimista tällä kurssilla

- Harjoitustyössä ei tarvitse kirjoittaa laukaisimia.
- Tentissä ei pyydetä kirjoittamaan laukaisimia, mutta on tunnettava laukaisimien toimintaperiaate ja ymmärrettävä, mikä ero on FOR EACH ROW-tyyppisillä ja FOR EACH STATEMENT-tyyppisillä laukaisimilla.
- SQLitessa on toteutettu vain FOR EACH ROW-tyyppiset (rivitason) laukaisimet, ei FOR EACH STATEMENT-tyyppisiä (lausetason) laukaisimia.
- Laukaisimien syntaksi SQLitessä on vähän erilainen kuin ensimmäisessä esimerkissä käytetty SQL-standardin mukainen syntaksi, ks. esimerkkejä seuraavilla kalvoilla.
- SQLitessä laukaisin suoritetaan aina heti jonkin rivin lisäämisen/poistamisen/muuttamisen jälkeen, vaikka SQL-standardin mukaan laukaisin pitäisi suorittaa vasta sen jälkeen, kun koko käsky on suoritettu.

Kalvon 5 laukaisin kirjoitettuna SQLite-ympäristössä

```
CREATE TRIGGER salaryTrigger
AFTER UPDATE OF salary ON Employee
WHEN OLD.salary > NEW.salary
BEGIN
    UPDATE Employee
    SET salary = OLD.salary
    WHERE ssNO = NEW.ssNO;
END;
```

Lisää esimerkkejä laukaisimista SQLite-syntaksilla

- Jos yritetään lisätä työntekijä, jonka palkka on yli 20 % suurempi kuin taulussa `Employee` tällä lisäyshetkellä oleva suurin palkka, lisäystä ei suoriteta.

```
CREATE TRIGGER salaryTrigger2
BEFORE INSERT ON Employee
WHEN NEW.salary > 1.2 * (SELECT MAX(salary) FROM Employee)
BEGIN
    SELECT raise(ignore);
END;
```

Lisää esimerkkejä laukaisimista SQLite-syntaksilla

- Jos jonkun `Employee`-taulun työntekijän palkkaa nostetaan yli 20 %:lla, niin kaikkien muiden työntekijöiden palkkaa nostetaan 10 %:lla.

```
CREATE TRIGGER salaryTrigger3
AFTER UPDATE OF salary ON Employee
WHEN NEW.salary > 1.2 * OLD.salary
BEGIN
    UPDATE Employee
    SET salary = 1.1 * salary
    WHERE ssNo <> NEW.ssNo;
END;
```

Lisää esimerkkejä laukaisimista SQLite-syntaksilla

- Laukaisin estää sen, että `Employee`-tauluun lisättäisiin uusi työntekijä, jolla on sama nimi kuin relaatiossa jo olevalla työntekijällä. (Laukaisin ei kuitenkaan estä taulussa jo olevan työntekijän nimen vaihtamista samaksi kuin toisella työntekijällä.)

```
CREATE TRIGGER nameTrigger
BEFORE INSERT ON Employee
WHEN NEW.name IN (SELECT name FROM Employee)
BEGIN
    SELECT raise(ignore);
END;
```

SQL-käskyjen lisääminen toisella kielellä kirjoitettuun ohjelmaan

- Tähänastisissa esimerkeissä SQL-käskyt on annettu suoraan tietokannan hallintajärjestelmälle.
- Yleensä halutaan kuitenkin kirjoittaa sovelluksia, joissa käyttäjän ei tarvitse tietää SQL:stä mitään. Sovellusohjelma kirjoitetaan jollain toisella kielellä ja sen avulla välitetään tarvittavat SQL-käskyt tietokannan hallintajärjestelmälle.
- SQL-käskyjä voidaan liittää ohjelmiin kahdella tavalla:
  - Käytetään hyväksi jotain sopivaa kirjastoa (esim. JDBC Java-ohjelmissa) ja sen tarjoamia funktioita, joiden avulla voidaan käsitellä tietokantaa.
  - SQL-käskyt kirjoitetaan suoraan esim. C-kieliseen ohjelmaan (ympäröityinä sopivilla tunnuksilla, jotta ne tunnistetaan SQL-käskyiksi). Kääntäjän esiprosessori käsittelee käskyt sopivasti, esimerkiksi korvaa ne sopvien kirjastofunktioiden kutsuilla.
- Lisäksi on mahdollisuus käyttää SQL-tietokantaa kirjoittamatta varsinaisia SQL-käskyjä ORM-tekniikan avulla.

ORM

- Object-relational mapping (ORM) on ohjelmointitekniikka, jonka avulla ohjelmoija voi kirjoittaa olio-ohjelmia, jotka käyttävät hyväkseen taustalla toimivaa relaatiotietokantaa.
- Tässä tekniikassa ohjelmoija ei itse määrittele esimerkiksi käytettäviä tauluja ja SQL-kyselyitä, vaan hän kirjoittaa ohjelman jollain olio-ohjelmointikielellä. Lisäksi hän käyttää hyväkseen ORM-työkalua, joka huolehtii yhteydestä taustalla pyörivään relaatiotietokantaan sekä ohjelman määrittelyjen ja käskyjen muuttamisesta tarvittaviksi SQL-käskyiksi.
- Esimerkiksi tietokantaolioiden luominen olio-ohjelmassa aiheuttaa yleensä uusien monikoiden lisäämisen taustalla olevan relaatiotietokannan tauluihin. Olion metodien kutsuminen aiheuttaa usein SQL-kyselyjä taustalla olevassa relaatiotietokannassa.

ORM, jatkoa

- Ohjelmoija ei kuitenkaan mieti kyselyitä relaatiotietokannan tasolla, vaan hän määrittelee luokkia, luo olioita ja kutsuu niiden metodeita. ORM pitää huolta näiden toimenpiteiden muuttamisesta relaatiotietokannan tarvitsemaan muotoon.
- ORMin käyttämisen etuja
  - Tietokannan hallintajärjestelmään liittyvät yksityiskohdat abstrahoidaan pois (voi tosin tarkoittaa joidenkin TKHJ-spesifisten etujen menettämistä).
  - Olioiden luominen ja metodien kutsuminen on usein helpompaa kuin vastaavien SQL-käskyjen kirjoittaminen.
- ORMin käyttämisen mahdollisia puutteita
  - Jotkut kyselyt voivat olla tehottomampia kuin suoraan SQL:ää käytettäessä.
  - Jotkut ongelmat voivat olla helpompia ratkaista ongelmaan erityisesti suunnitellulla SQL:llä
- Tällä kurssilla ei käsitellä ORMeja enempää. Kurssilla Web Software Development käytetään harjoitustyössä Django web-kehitysympäristön ORMia.

SQL-käskyjen liittäminen Python-ohjelmaan

- Jos halutaan käyttää tietokantaa SQL-käskyjen avulla Python-ohjelmassa, käytetään yleensä hyväksi tarkoitukseen sopivaa Python-kirjastoa.
- Kirjastoja on paljon erilaisia eri tietokannan hallintajärjestelmille. Tässä esitellään kirjastoa sqlite3, jonka avulla voi käyttää SQLite-nimistä tietokannan hallintajärjestelmää.
- Huomaa, että tämän kurssin harjoitustyössä riittää pelkkä SQL-käskyjen kirjoittaminen eikä seuraavaksi esiteltyjä asioita tarvitse käyttää harjoitustyössä. Harjoitustehtävissä kierroksen 5 viimeisessä tehtävässä kirjoitetaan Python-ohjelma, jonka avulla luodaan ja käytetään tietokantaa.

SQLite ja moduuli sqlite3

- SQLite on tietokannan hallintajärjestelmä, joka ei tarvitse erillista tietokantapalvelinta. Tietokannan käyttämä data tallennetaan suoraan tiedostoihin.
- SQLite tulee Python-tulkin asennuksen mukana. Sen voi myös asentaa erikseen. Myös moduuli sqlite3 on Python-asennuksessa oletuksena mukana.
- SQLite toteuttaa SQL92 (= SQL2):n piirteet muutamaa poikkeusta lukuunottamatta.
- SQLite käyttää dynaamista tyypitystä. Se sallii taulujen monikoiden attribuuttien arvoille myös muut tyypit kuin mitä niille on määritelty. Tyyppien valikoima (integer, real, text, blob, null) on pienempi kuin SQL-standardissa, mutta SQLite käsittelee myös SQL-standardilla kirjoitetut määrittelyt ilman virheilmoituksia.
- Kurssilla käytetty SQLiteStudio on käyttöliittymäohjelma SQLitelle.

Moduulin sqlite3 käyttö

- Python-ohjelman aluksi on määriteltävä, että ohjelma käyttää tätä moduulia:

  ```
  import sqlite3
  ```

- Seuraavaksi muodostetaan yhteys haluttuun tietokantaan. Tietokannan nimi (tietokannan sisältävän tiedoston nimi) annetaan komennon parametrina. Jos annetun nimistä tietokantaa ei ole, se luodaan.

  ```
  conn = sqlite3.connect("webstoredatabase.db")
  ```

  Jos tietokannan nimeksi antaa `:memory`, tietokantaa ei tallenneta minnekään.

- Kun yhteys on luotu, luodaan kursori, jonka avulla voidaan tehdä tietokantaan kyselyitä ja muutoksia.

  ```
  cursor = conn.cursor()
  ```

SQL-käskyjen suorittaminen

- Kun kursori on määritelty, voidaan sen avulla suorittaa SQL-käskyjä metodin `execute` avulla. Suoritettava SQL-käsky annetaan merkkijonona (kolme lainausmerkkiä, koska käsky koostuu useammasta rivistä). Esimerkkinä uuden taulun määrittely:

```
cursor.execute("""CREATE TABLE Products (
                  number TEXT PRIMARY KEY,
                  prodName TEXT,
                  description TEXT,
                  price REAL,
                  manufID TEXT)""")
```

SQL-käskyjen suorittaminen: monikon lisääminen

- Toisena esimerkkinä monikon lisääminen luotuun tauluun:

```
cursor.execute("""INSERT INTO Products
VALUES('P-48221', 'Teema 26', 'plate', 16.00,'F542')""")
```

- Kun halutut päivitykset on tehty, on yhteydelle suoritettava `commit`-operaatio. Muuten tiedot eivät tallennu:

```
conn.commit()
```

- Kun tietokantaa ei enää tarvita, yhteys siihen voidaan sulkea (Huom! `commit`-operaatio on ehdottamasti suoritettava ennen tätä.)

```
conn.close()
```

Kyselyiden tekeminen ja tulosten läpikäynti

- Kyselyitä voidaan tehdä kursorin `execute`-metodilla:

```
cursor.execute("SELECT * FROM Products WHERE price < 99")
```

- Kun kysely on tehty, voidaan tulosrivit hakea joko kaikki kerralla `fetchall`-metodilla tai sitten yksi kerrallaan `fetchone`-metodilla. Esimerkki ensimmäisen käytöstä (ja haettujen rivien tulostuksesta):

```
rows = cursor.fetchall()
for row in rows:
    print(row)
```

- Esimerkki tulosrivien käymisestä läpi yksi kerrallaan:

```
while True:
    row = cursor.fetchone()
    if row == None:
        break
    else:
        print(row)
```

Kyselyn tulostuksesta

- Kyselyn tulokset ovat (Python-kielen) monikoita, joista voi helposti ottaa tulokseen vain osan indeksoimalla. Indeksinä voi käyttää attribuutin indeksiä (kuinka mones attribuutti on kysymyksessä).
- Esimerkki, jossa tulosrelaation tuotteista on tulostettu vain nimi ja hinta:

```
rows = cursor.fetchall()
for row in rows:
    print(row[1], row[3])
```

Käyttäjän syötteen käyttäminen kyselyissä

- Kyselyihin (ja muihinkin SQL-käskyihin) voi lisätä käyttäjältä saatua tietoa muuttujien avulla.
- Muuttujien arvoja ei saa kuitenkaan liittää suoraan kyselyssä käytettävään merkkijonoon, koska se mahdollistaisi *SQL-injektion* (SQL injection), jossa pahaa tarkoittava käyttäjä voi syötteensä mukana lähettää tietokannan hallintajärjestelmälle ei-toivottuja käskyjä. Katso esimerkki sivulta `http://xkcd.com/327/`
- SQL-injektion riski vältetään, kun SQL-käskyt välitetään SQLitelle parametrisoituna: komentotekstiin kirjoitetaan kysymysmerkit niille paikoille, johon sijoitetaan käyttäjän antamia arvoja. Kyselytekstin jälkeen annetaan lista tai monikko, joka sisältää ne muuttujat, joiden arvot sijoitetaan kysymysmerkkien paikalle.
- Esimerkki:

```
name1 = input("Give the name of the product: ")
cursor.execute("SELECT * FROM Products WHERE name =?",
               (name1,))
```

Esimerkkejä 1

- Pyydetään käyttäjältä tuotteen kuvaus ja maksimihinta. Haetaan ehdon täyttävien tuotteiden tiedot:

```
import sqlite3

conn = sqlite3.connect("webstoredatabase.db")
cursor = conn.cursor()
desc1 = input("Give the description: ")
price1 = float(input("Give the maximum price: "))
query = """SELECT * FROM Products
           WHERE description =? AND  price <=?"""
cursor.execute(query, (desc1, price1))
rows = cursor.fetchall()
for row in rows:
    print(row)
conn.close()
```

Esimerkkejä 2

- (29.4. luennolla päästiin tähän asti, ja kalvojen loppu käsitellään 13.5.)
- Lisätään Products-tauluun tuote, jonka tiedot saadaan käyttäjältä

```
import sqlite3
conn = sqlite3.connect("webstoredatabase.db")
cursor = conn.cursor()
print("Give information of the product to be inserted")
number1 = input("number: ")
name1 = input("name: ")
desc1 = input("description: ")
price1 = float(input("price: "))
manufid1 = input("manufacturer id: ")
insertion = "INSERT INTO Products VALUES(?, ?, ?, ?, ?)"
cursor.execute(insertion, (number1, name1, desc1,
                           price1, manufid1))
conn.commit()
conn.close()
```

Huomattavaa

- Näillä kalvoilla on esitetty vain pieni osa `sqlite3`-kirjaston tarjoamista mahdollisuuksista. Aiheeseen voi tutustua lisää esimerkiksi sivulla `http://zetcode.com/db/sqlitepythontutorial/` olevan tutorialin avulla tai Pythonin omasta dokumentaatiosta sivulla `http://docs.python.org/3/library/sqlite3.html`.
- Kirjasto tarjoaa mahdollisuuden esimerkiksi suorittaa useita SQL-käskyjä yhdellä kirjastofunktion kutsulla, hakea tauluun lisättävät monikot listasta jne. Näitä käsitellään lisää 13.5. luennolla.
- Esimerkkien yksinkertaistamiseksi niistä oli jätetty virheenkäsittely pois. Kirjastofunktioiden suorituksissa tulleita ongelmia voi käsitellä `sqlite3.Error`-tyyppisinä poikkeuksina.

SQL-tietokannat Scala-ohjelmissa

- Sivulla http://manuel.bernhardt.io/2014/02/04/a-quick-tour-of-relational-database-access-with-scala/ on esitetty erilaisia vaihtoehtoja käyttää SQL-käskyjä Scala-ohjelmassa. Sivulla on esitelty sekä kirjasto- että ORM-pohjaisia ratkaisuja.
- Seuraavaksi esitellään lyhyesti Anorm-nimistä työkalua, joka tarjoaa yhden kirjastopohjaisen ratkaisun.
- Esitetyt esimerkit on muokattu Anormin dokumentaation pohjalta eikä niitä ole testattu.

Anormin käyttö: aloitus

- Lataa (esimerkiksi Eclipseen) seuraavat kirjastot ja lisää ne projektiin: `play`, `jdbc`, `anorm`.
- Ohjelmatiedoston alussa ota käyttöön tarvittavat kirjastot:

```
import anorm._
import play.api.db.DB
```

- Määritellään oletustietokanta. Tässä esimerkissä se on SQLite-tietokanta, joka on tallennettu tiedostoon nimeltä `/path/to/db-file` (alku on polku siihen hakemistoon, joka sisältää ko. tiedoston).

```
db.default.driver=org.sqlite.JDBC
db.default.url="jdbc:sqlite:/path/to/db-file"
```

Anorm: yhteys tietokantaan ja SQL-käskyjen suorittaminen

- Avataan yhteys tietokantaan ja suoritetaan ensimmäinen kysely:

```
DB.withConnection { implicit c => {
  val result: Boolean = SQL("Select 1").execute()
  // muita kyselyita ym}
}
```

  Muiden saman tietokantayhteyden aikana suoritettavien SQL-käskyjen suorittamiseksi tarvittava koodi kirjoitetaan kohtaan
  `// muita kyselyita ym`

- Seuraavaksi annetaan joitain esimerkkejä käskyistä, joita tähän kohtaan voi kirjoittaa.

Anorm: esimerkki lisäyksestä

```
val id: Option[Long] = SQL(
  """
    insert into BelongsTo(orderNo, productNo, count)
    values ({number1}, {number2}, {count1})
  """
  ).on('number1 -> "469666",
       'number2 -> "S-65221",
       'count1 -> 2
  ).executeInsert()
```

Lisäysesimerkin selityksiä

- Itse suoritettava SQL-käsky on annettu `SQL()`:n parametrina. Käskyn ympärillä on kolme lainausmerkkiä, koska käsky koostuu useasta rivistä.
- Aaltosuluilla merkityt nimet (esim. `{number1}`) ovat samanlaisia paikanvaraajia kuin kysymysmerkit sqlite3-kirjastoa käytettäessä. Seuraavan `on`-kutsun parametrina kerrotaan, mitkä arvot paikoille tulevat käskyä suoritettaessa.
- Kutsu `executeInsert()` varsinaisesti aiheuttaa käskyn suorittamisen.
- Lisäyksen yhteydessä lisättävälle riville luodaan keinotekoinen avain, joka palautetaan ja otetaan talteen muuttujaan `id`.

Anorm: esimerkki kyselyn määrittelemistä

```
val selectProducts = SQL(
                       """
                         select number, name
                         from Prodcuts
                         where description = {desc1};
                       """
                     ).on("desc1" -> "cellphone")
```

- Tässä ei vielä suoriteta kyselyä, vaan määritellään se ja tallennetaan se muuttujaan `selectProducts`.
- Jälleen `{desc1}` on paikanvaraaja, joka korvataan `on`-kohdassa määritellyllä arvolla.

Anorm: esimerkkikyselyn suorittaminen ja tulosten tallentaminen listaan

- Suoritetaan edellisellä kalvolla määritelty kysely, muutetaan saadut rivit pareiksi (tuotteen numero, tuotteen nimi), ja kootaan näistä pareista lista.
- Koottuun listaan viitataan muuttujalla `productinfo`

```
val productinfo = selectProducts().map(row =>
  row[String]("number") -> row[String]("name")
).toList
```

SQL-käskyjen kirjoittaminen ohjelmaan suoraan

- C-ohjelmiin voidaan kirjoittaa suoraan SQL-käskyjä ympäröitynä sopivilla tunnuksilla, jotka C-kääntäjän esiprosessori käsittelee. Tekniikkaa kutsutaan upotetuksi SQL:ksi (embedded SQL).
- C-ohjelmaan kirjoitetut SQL-käskyt aloitetaan aina ilmauksella EXEC SQL
- Tällä kurssilla ei käsitellä asiaa yksityiskohtaisesti (lisää asiasta on oppikirjassa), mutta seuraavalla kalvolla on mainittu lyhyesti muutamia keskeisiä asioita.

Upotettun SQL:n käyttöön liittyviä keskeisiä käsitteitä

- SQL-käskyn ja isäntäkielisen (C-kielellä kirjoitun) ohjelman avulla voidaan välittää tietoa *yhteisten muuttujien* (shared variables) avulla, joita voi käyttää sekä C-kielisessä osassa että SQL-käskyissä. Kun näitä muuttujia käytetään SQL-käskyissä, niiden nimen eteen kirjoitetaan kaksoispiste.
- Muuttuja SQLSTATE on viiden merkin mittainen merkkijono, jonka arvo kertoo kunkin SQL-käskyn suorituksen jälkeen käskyn onnistumisesta.
- Jos suoritetaan kyselyjä, joiden tuloksena voi olla useita rivejä, voidaan tulosrivejä käydä läpi *kursorin* avulla (eri kuin sqlite3-kirjaston cursor).

ORMit, SQL:n yhdistäminen Python- tai Scala-ohjelmiin ja upotettu SQL tentissä

- Tentissä riittää, että tietää periaatteessa, mistä näissä asioissa on kysymys.
- Tentissä ei pyydetä kirjoittamaan SQL-käskyjä sisältäviä Python- tai Scala-ohjelmia tai upotettua SQL:ää.