# 

1. sudo service mysql status
2. sudo service mysql start

## Primitive Types

|  |  |  |
| --- | --- | --- |
| **Mapping type** | **Java type** | **ANSI SQL Type** |
| integer | int or java.lang.Integer | INTEGER |
| long | long or java.lang.Long | BIGINT |
| short | short or java.lang.Short | SMALLINT |
| float | float or java.lang.Float | FLOAT |
| double | double or java.lang.Double | DOUBLE |
| big\_decimal | java.math.BigDecimal | NUMERIC |
| character | java.lang.String | CHAR(1) |
| string | java.lang.String | VARCHAR |
| byte | byte or java.lang.Byte | TINYINT |
| boolean | boolean or java.lang.Boolean | BIT |
| yes/no | boolean or java.lang.Boolean | CHAR(1) ('Y' or 'N') |
| true/false | boolean or java.lang.Boolean | CHAR(1) ('T' or 'F') |

## Date and Time Types

|  |  |  |
| --- | --- | --- |
| **Mapping type** | **Java type** | **ANSI SQL Type** |
| date | java.util.Date or java.sql.Date | DATE |
| time | java.util.Date or java.sql.Time | TIME |
| timestamp | java.util.Date or java.sql.Timestamp | TIMESTAMP |
| calendar | java.util.Calendar | TIMESTAMP |
| calendar\_date | java.util.Calendar | DATE |

## Binary and Large Object Types

|  |  |  |
| --- | --- | --- |
| **Mapping type** | **Java type** | **ANSI SQL Type** |
| binary | byte[] | VARBINARY (or BLOB) |
| text | java.lang.String | CLOB |
| serializable | any Java class that implements java.io.Serializable | VARBINARY (or BLOB) |
| clob | java.sql.Clob | CLOB |
| blob | java.sql.Blob | BLOB |

## JDK-related Types

|  |  |  |
| --- | --- | --- |
| **Mapping type** | **Java type** | **ANSI SQL Type** |
| class | java.lang.Class | VARCHAR |
| locale | java.util.Locale | VARCHAR |
| timezone | java.util.TimeZone | VARCHAR |
| currency | java.util.Currency | VARCHAR |

# <https://www.tutorialspoint.com/hibernate/hibernate_mapping_types.htm>

### Duomenų išrinkimas**[**[**redaguoti**](https://lt.wikipedia.org/w/index.php?title=SQL&veaction=edit&section=2)**|**[**redaguoti vikitekstą**](https://lt.wikipedia.org/w/index.php?title=SQL&action=edit&section=2)**]**

[ANSI](https://lt.wikipedia.org/w/index.php?title=ANSI&action=edit&redlink=1) bei [ISO](https://lt.wikipedia.org/wiki/ISO) standartai apibrėžia šiuos SQL raktažodžius, skirtus duomenims išrinkti:

* SELECT sakinys naudojamas įrašams iš vienos ar daugiau lentelių atrinkti; šie įrašai dažniausiai atrenkami pagal tam tikrus kriterijus.
* FROM sakinyje nurodomos lentelės, iš kurių reikia išrinkti eilutes (sąryšiai gali būti nurodomi skirtingais [JOIN](https://lt.wikipedia.org/wiki/Jungties_operacija) variantais).
* WHERE sakinyje nurodoma sąlyga, kurią turi tenkinti grąžinamos eilutės.
* GROUP BY sąlygoje nurodoma, kad reikia grupuoti tam tikras eilutes. Grupuojant eilutes, dažniausiai naudojamos agregatinės funkcijos maksimalioms, vidutinėms ir panašioms reikšmėms išrinkti iš grupuotų eilučių.
* ORDER BY sakiniu nurodoma viena ar daugiau rikiavimo sąlygų.
* HAVING sakinyje nurodomas kriterijus, taikomas grupuojamoms eilutėms; šis raktinis žodis gali būti naudojamas tik tais atvejais, jeigu užklausoje yra GROUP BY sakinys.

Iš išvardytų sakinių bet kurioje užklausoje visada naudojami SELECT bei FROM sakiniai, o kiti naudojami pagal poreikį.

### Duomenų valdymas**[**[**redaguoti**](https://lt.wikipedia.org/w/index.php?title=SQL&veaction=edit&section=3)**|**[**redaguoti vikitekstą**](https://lt.wikipedia.org/w/index.php?title=SQL&action=edit&section=3)**]**

* INSERT vartojamas naujų įrašų įterpimui į lentelę
* DELETE leidžia ištrinti įrašus iš lentelės.
* UPDATE naudojamas pakeisti vieno ar daugiau įrašų reikšmes.

### Transakcijos**[**[**redaguoti**](https://lt.wikipedia.org/w/index.php?title=SQL&veaction=edit&section=4)**|**[**redaguoti vikitekstą**](https://lt.wikipedia.org/w/index.php?title=SQL&action=edit&section=4)**]**

Sistemose, kurios palaiko [transakcijas](https://lt.wikipedia.org/wiki/Transakcija), galima naudoti šias komandas:

* BEGIN nurodoma pradėti atominę operaciją (transakciją)
* COMMIT patvirtinama sėkmingai baigiama transakcija
* ROLLBACK nurodoma, kad visa transakcija atšaukiama

### Duomenų apibrėžimas**[**[**redaguoti**](https://lt.wikipedia.org/w/index.php?title=SQL&veaction=edit&section=5)**|**[**redaguoti vikitekstą**](https://lt.wikipedia.org/w/index.php?title=SQL&action=edit&section=5)**]**

* CREATE naudojama sukurti įvairiems objektams, pavyzdžiui, lentelėms.
* DROP nurodoma sunaikinti tam tikrus objektus.

Kai kuriuos sistemos turi komandą ALTER, kuria galima pakeisti objektus sistemos darbo metu.

## Trūkumai[[redaguoti](https://lt.wikipedia.org/w/index.php?title=SQL&veaction=edit&section=6) | [redaguoti vikitekstą](https://lt.wikipedia.org/w/index.php?title=SQL&action=edit&section=6)]

* Nėra standartinio būdo skaidyti sudėtingas komandas į kelias smulkesnes
* SQL realizacijos skirtingose duomenų bazių valdymo sistemose nėra nuoseklios, nepilnai suderinamos.

Kadangi standartinių galimybių dažnai neužtenka, duomenų bazių sistemose SQL išplečiama jai pridedant daugiau programavimo kalbų funkcijų

# Basic SQL queries <https://www.tutorialspoint.com/mysql/index.htm>

Šioje pamokoje bus pateikta informacija apie SQL užklausas. Pavyzdžiuose bus naudojamas stalai kurie atrodo taip:

\------------------/

| Person |

--------------------

| ID | NAME | AGE |

--------------------

| 1 | Jonas | 10 |

| 2 | Tomas | 53 |

| 3 | Egle | 22 |

| 4 | Toma | 33 |

| 5 | Ieva | 22 |

/------------------\

\--------------------/

| Salary |

----------------------

| PERSON\_ID | PAY |

----------------------

| 1 | 1000 |

| 2 | 2500 |

| 3 | 0 |

| 5 | 1234 |

| 6 | 800 |

/--------------------\

Salary(PERSON\_ID) is FK for Person(ID);

### Operators

Operatoriai SQL veikia taip pat kaip ir daugumoje kitų vietų programavimo pasauly išskyrus loginius operatorius. SQL natively palaiko daugiau loginių operatorių ir jie išreiškiami žodžiais, pcz: AND, OR, LIKE, BETWEEN. Naudojami WHERE išraiškoje

SELECT \* FROM <table> WHERE <int\_column> = <int\_value> OR <some\_other\_string\_column> = '<some\_string\_value>'

Pavyzdys: Išraiška norit rasti visus Person kur amžius lygu 10 arba Person vardas 'Egle' atrodytu taip:

SELECT \* FROM person WHERE age = 10 OR name = 'Egle'

Grąžintų:

| 1 | Jonas | 10 |

| 4 | Egle | 22 |

Daugiau info: <https://www.tutorialspoint.com/sql/sql-operators.htm>

### Wildcards

Wildcard yra išraiška kuri save teigiamai prilyginą betkokiai kitai išraiškai. Kaip matėt ji gali būti naudojama kuriant naują duombazės vartotoją, tačiau dažnesnis panaudojimas yra LIKE statement'e apie kurį sužinosim daugiau kitame aprašyme.

* % - lygina bent vieną ar daugiau ženklų
* \_ - lygina vieną ženklą

### LIKE

SQL loginis operatorius skirtas ieškoti apytikslėm reikšmėm. Dažniausiai naudojama WHERE išraiškoje.

SELECT \* FROM <table> WHERE <column> LIKE 'XXXX%'

Pavyzdys: Su Person lentele tokia užklausa:

SELECT \* FROM person WHERE name LIKE 'Tom%'

Grąžintų:

| 2 | Tomas | 53 |

| 4 | Toma | 33 |

Nes išraiska Tom% tinka ir Tomas ir Toma stringam.

### Ordering

Duombazė gali grąžinti jau surikiuotus rezultatus. Tai padaryti galima naudojant ORDER BY išraišką kuri atrodo taip:

SELECT \* FROM <table> ORDER BY <column> ASC; -- didėjimo tvarka.

SELECT \* FROM <table> ORDER BY <column> DESC; -- mažėjimo tvarka.

Pavyzdys:

SELECT \* FROM person ORDER BY age DESC;

Grąžintų:

| 2 | Tomas | 53 |

| 4 | Toma | 33 |

| 3 | Egle | 22 |

| 5 | Ieva | 22 |

| 1 | Jonas | 10 |

Tokiu būdu Java programoje išsaugant užklausa ResultSet klasėje pirmas įrašas būtų Tomas su indeksu 2

### DISTINCT

Renkantis įrašus su unikaliomis reikšmėmis galima utilizuoti raktažodį DISTINCT

SELECT DISTINCT <column> FROM <table>;

Pavyzdys:

SELECT DISTINCT age FROM person;

Grąžintų:

| 10 |

| 53 |

| 22 |

| 33 |

Tai yra visos unikalios amžiaus reikšmės

### GROUP BY

Sąrašo užklausos (SELECT statement'as) rezultatus įmanoma grupuoti. Kuriant grupavimo užklausas reikia gerai suprasti rezultato išvestys (tai kas yra tarp SELECT ir FROM) turi gera sąryšį. Jei sąryšis neįmanomas, užklausa nesuveiks.

Pavyzdys teisingo sąryšio: Person grupavimas pagal amžių. Toks grupavimas galėtų būti išreikštas išvedant amžių ir skaičių kiek tokios amžiaus žmonių yra. Užklausos pavizdys žemiau.

Pavyzdys neįmanomo sąryšio: Jei prie paminėtos amžiaus išvesties pridėtume vardo (name) stulpelį duombazė nežinotu kaip susieti visus gautus vardus prie vienos rezultato eilutės (kuri parodo amžių ir kiek to amžiaus žmonių yra dviejuose stulpeliuose) išvesties. Atsiminkit, kad vienas stulpelis gali laikyti tik vieną reikšmę.

SELECT name, age, COUNT(age) FROM person GROUP BY age; <-- klaidinga

Struktūra:

SELECT <column\_1>, COUNT(<column\_1>) FROM <table> GROUP BY <column\_1>;

Pavyzdys:

SELECT age, COUNT(age) FROM person GROUP BY age;

Grąžintų:

| 10 | 1 |

| 22 | 2 |

| 33 | 1 |

| 53 | 1 |

### LIMIT

LIMIT raktažodis leidžia limituoti rezultatų sąrašą taip, kad parodytu tik pirmus N įrašų, kur N yra nustatytas skaičius prie raktažodžio

SELECT \* FROM LIMIT <N>

Pavyzdys:

select \* from employee limit 3;

Grąžintų:

| 1 | Jonas | 10 |

| 2 | Tomas | 53 |

| 3 | Egle | 22 |

## Advanced SQL queries

### Constraints

Constraints (apribojimai) yra taisyklės pritaikytos lentelių stulpeliams kurios limituoja nurodytas savybes. Pavyzdžiui pridedant NOT NULL kuriant stulpelį - tam stulpelyje negali būti null reikšmių.

Kiti constraints kuriuos jau galimai žinot yra AUTO\_INCREMENT, PRIMARY KEY, UNIQUE ir t.t.

Kaikurie apribojimai gali turėti nuosavą pavadinimą, pavyzdžiui FOREIGN KEY. Standartinė praktika yra pavadinti šiuos apribojimus pagal pačio apribojimo trumpinimą (FOREIGN KEY -> FK), pridėti \_ ir trumpai nusakyti koks tai apribojimas. Pavyzdžiui fk\_employee\_Id. Tai yra reikalinga pagrinde tam, kad klaidų žinutės būtų aiškesnės. Taip pat tai padaro tvarkingesnė duombazė, lengviau ieškoti reikiamų apribojimų naudojant LIKE raktažodį ir t.t.

MySQL pažiūrėti kokie apribojimai egzistuoja galima su šia užklausa:

select \* from information\_schema.table\_constraints where constraint\_schema = 'playground'; -- <- "playground" - duombazės pavadinimas kur ieškote apribojimų.

O jei norima šiek tiek daugiau detalių, galima taip:

SELECT tc.constraint\_schema,tc.constraint\_name,tc.table\_name,tc.constraint\_type,kcu.table\_name,kcu.column\_name,kcu.referenced\_table\_name,kcu.referenced\_column\_name,rc.update\_rule,rc.delete\_rule

FROM information\_schema.table\_constraints tc

inner JOIN information\_schema.key\_column\_usage kcu

ON tc.constraint\_catalog = kcu.constraint\_catalog

AND tc.constraint\_schema = kcu.constraint\_schema

AND tc.constraint\_name = kcu.constraint\_name

AND tc.table\_name = kcu.table\_name

LEFT JOIN information\_schema.referential\_constraints rc

ON tc.constraint\_catalog = rc.constraint\_catalog

AND tc.constraint\_schema = rc.constraint\_schema

AND tc.constraint\_name = rc.constraint\_name

AND tc.table\_name = rc.table\_name

WHERE tc.constraint\_schema = 'playground' -- <- "playground" - duombazės pavadinimas kur ieškote apribojimų.

### Alias

Alias yra alternatyvus pavadinimas kažkokiai reikšmei: stulpeliui, stalui, funkcijai ar betkokiai išraiškai. Alias'ai leižia kurti aiškiasnius rezultatų setus ir aiškiasnes užklausas. Pavyzdžiui galima funkcija COUNT(age) pavadinti tiesiog skaicius (ar number) ir po to tą pačią išraiška naudoti toliau užklausoje.

SELECT <alias\_name.column1> FROM <table\_name> AS <alias\_name>

Pavyzdys:

SELECT age, COUNT(age) AS number FROM person GROUP BY age ORDER BY number;

Grąžintų:

\----------------/

| Person |

------------------

| age | number |

------------------

| 10 | 1 |

| 53 | 1 |

| 33 | 1 |

| 22 | 2 |

/----------------\

Čia išraiškai COUNT(age) buvo duotas alias number ir toliau tą patį alias galima naudoti duomenų rikiavimui ORDER BYišraiškoje. Taip pat atkreipkite dėmesį į rezultato stulpelio pavadinimą kuris taip pat yra number.

### Queries within queries

SQL įmanoma kurti užklausas užklausose. Esminė taisyklė yra ta, kad reikšmė kurią gražina užklausa turi būti suvokiama loginėje išraiškoje kurioje ji naudojama.

Pavyzdys: Norint rasti žmogaus vardą kuris turi didžiausią atlyginimą utilizuojant dvi lenteles, Person ir Salary, galima daryti taip:

SELECT name FROM person where id = (SELECT PERSON\_ID FROM SALARY ORDER BY SALARY DESC LIMIT 1);

Mes žinom, kad PERSON\_ID stulpelis letelėje SALARY reprezentuoja PERSON lentelės stulpelį ID. Tai yra todėl, kad PERSON\_IDyra FOREIGN KEY į Person(id). Todėl mes sukuriam pirmą užklausą:

SELECT PERSON\_ID FROM SALARY ORDER BY SALARY DESC LIMIT 1;

kad rasti id žmogaus kuris turi didžiausią atlyginimą. Atkreipkite dėmsį, kad ši užklausa gražiną vieną stulpelį su viena reikšmę. Tai darom todėl, kad lygybės ženklas ( where id =) gali lyginti tik su viena reikšme (kaip ir >, <, LIKE ir t.t.). Taigi, gavę šią vieną reikšmę mes galim surasti įrašą kitoje lentelėje naudodami užklausą užklausoje.

Užklausą užklausoje taip pat galima naudoti ir su keliomis reikšmėmis, tačiau reikia naudoti kitokį loginį operatorių, pavyzdžiui IN - reikšmės lyginimas su aibe, kitaip tariant jei reikšmė egzistuoja duotoje aibėje loginė reikšmė gražins true, kitu atvėju false . Taigi jei modifikuojam buvusę užklausą, kad ji grąžintų stulpelį su keliais įrašais:

SELECT PERSON\_ID FROM SALARY ORDER BY SALARY DESC;

rezultatu:

| 2 |

| 5 |

| 1 |

| 6 |

| 3 |

ir pakeičiam = į IN:

SELECT name FROM person where id IN (SELECT PERSON\_ID FROM SALARY ORDER BY SALARY DESC);

rezultatas būtų:

| Tomas |

| Ieva |

| Jonas |

| Egle |

Toma nebūtų rezultate nes Salary stalas neturi jos įrašo (PERSON\_ID kuris būtų 4). Taip pat Person lentelė neturi įrašo su ID = 6 todėl dėl šios reikšmės niekas neįvyksta.

### Joins

Viena iš stipriausių SQL funkcijų. Join'as (sujungimas) leidžia pasirinkti reikšmes iš kelių skirtingų lentelių duombazėj ir su jomis dirbti. Join'ai veikia tarp lentelių kurios gali turėti saryšį, nors constraint'as (apribojimas) nebūtinai turi egzistuoti.

Sintaksė:

SELECT \* FROM <table1> as <t1>

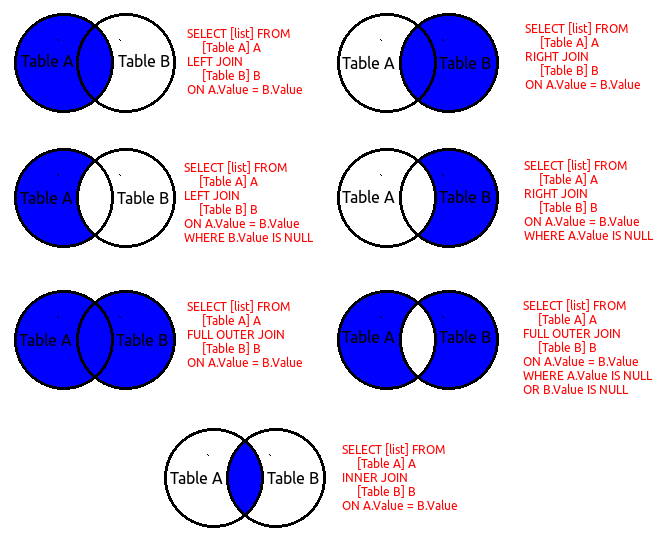
<join\_type> JOIN <table2> as <t2> ON <t2>.<column> = <t1>.<column>

ON raktažodis rodo kokiu stulpeliu sujungimas turi būti paremtas. Kitaip tariant <t2>.<column> ir <t1>.<column> turi reprezentuoti tą pačią reikšmę skirtinguose staluose, kad ši užklausa turėtu logikos. Galima sujunginėti kelis stalus.

Duombazės struktura naudoja savokas kairė ir dešinė lentelė. Kairė - lentelė prie kurio jungiama (SELECT \* FROM <kairė\_lentelė <..>). Dešinė - ta kuri yra jungiama (<..> JOIN <dešinė\_lentelė>). Vizualizacija galite pamatyti apačioje.

Egzistuoja skirtingų tipų JOIN'ai:

* INNER JOIN - gražina tik stulpelius kurie turi visas reikšmes
* LEFT JOIN - gražina visus stulpelius iš kairės lentelės net jei jos neturi reikšmės dešinėje lentelėje
* RIGHT JOIN - gražina visus stulpelius iš dešinės lentelės net jei jos neturi reikšmės kairėje lentelėje
* FULL JOIN - gražina stulpelius jei yra reikšmės bent viename stulpelyje. MySQL nepalaiko šitos užklausos
* CARTESIAN/CROSS JOIN - Tai yra kiekvienos reišmės prijungimas kairėje lentelėje prie kiekvienos reikšmės dešinėje lentelėje. Pvz jei viena letelė turi 10 įrašų ir kita 100, rezultate bus 1000 įrašų. (MySQL naudoja CROSS raktažodį)
* Self join - nėra raktažodžio lentelės. Tai yra lentelės prijungimas prie savęs. Turi būti naudojami alias (žiūrėti Alias aprašymą)  
  .

[](https://camo.githubusercontent.com/38efe3c2300e696190d86c45d6244c9c83ef1e3d/68747470733a2f2f692e696d6775722e636f6d2f62336f3152316a2e706e67)

### UNION

UNION raktažodis yra naudojamas kai norima sujungti rezultatus iš kelių SELECT užklausų ir nenorima gauti duplikuotų reikšmių. Utilizuoti UNION išraišką galima jei naudojamos SELECT užklausos turi:

* Vienodą sakičių rezultatų
* Vienodą skaičių stulpelių
* Tą patį datos tipą
* Tą patį eiliškumą

Sintaksė:

SELECT <column\_1>, <column\_2> from <table\_1>

UNION

SELECT <column\_1>, <column\_2> from <table\_1>;

Pavizdys:

select \* from person where age < 22 --<- užklausa 1

union

select \* from person where AGE > 33; --<- užklausa 2

Grąžintų:

| 1 | Jonas | 10 |

| 2 | Tomas | 53 |

nes pirmos užklausos rezultatai yra sujungiami su antros užklausos rezultatais.

# Advanced SQL

Pavyzdžiuose bus naudojamas stalai kurie atrodo taip:

\------------------/

| Person |

--------------------

| ID | NAME | AGE |

--------------------

| 1 | Jonas | 10 |

| 2 | Tomas | 53 |

| 3 | Egle | 22 |

| 4 | Toma | 33 |

| 5 | Ieva | 22 |

/------------------\

\--------------------/

| Salary |

----------------------

| PERSON\_ID | PAY |

----------------------

| 1 | 1000 |

| 2 | 2500 |

| 3 | 0 |

| 5 | 1234 |

/--------------------\

Salary(PERSON\_ID) yra FOREIGN KEY lentelei Person(ID);

Person(ID) turi AUTO\_INCREMENT savybę

### VIEW

VIEW yra tiesiog SQL užklausa išsaugota duombazėje kuriai duotas vardas. Kai View SQL užklausa yra sukurta ją vėliau galima panaudoti kada nori kreipiantis į ją per jai suteiktą vardą.

Sintaksė:

CREATE VIEW <any\_view\_name> AS SELECT <..>

Iš principo View sukurimas yra tiesiog raktažodžiųCREATE VIEW <any\_view\_name> AS pridėjimas prieš SELECT užklausą. Kuomet View sukurta, ant jos galima paleisti SELECT užklausas tokiu pačiu būdu kaip jos būtų paleistos ant duombazės lentelės.  
Taigi, tokia užklausa:

CREATE VIEW age\_groups as SELECT age, COUNT(age) FROM person GROUP BY age;

Sukurtu View pavadinimu age\_groups kurią dabar galima SELECT'inti:

SELECT \* FROM age\_groups

Tai grąžintų tą patį ką gražina nusakyta SELECT užklausa panaudota sukuriant View (netgi stulepių pavadinimai išliktų):

| age | COUNT(age) |

--------------------

| 10 | 1 |

| 22 | 2 |

| 33 | 1 |

| 53 | 1 |

View taip pat galima atnaujinti(UPDATE) ir į jas įvesti naujas reikšmes (INSERT). Tačiau tam reikia tenkinti kelias taisykles:

* SELECT užklausa negali turėti stulpelių su savybe DISTINCT.
* SELECT užklausa negali turėti funkcijų kurios dirba su keliais stulpeliais, tokios kaip SUM(), MIN(), MAX() ir t.t.
* SELECT užklausa negali turėti ORDER BY savybės.
* FROM dalis negali turėti kelių stalų (dirbant su UNION arba JOIN).
* WHERE negali turėti Sub-queries.
* Užklausa negali turėti savybės GROUP BY arba HAVING (HAVING tas pats kas WHERE kuris gali dirbti su funkcijomis).
* Stulpeliai kurie yra paskaičiuojami (pvz su Trigger'iais) negali būti atnaujinti su naujomis reikšmėmis.
* Visi NOT NULL stulpeliai turi būti įtraukti į View, kad INSERT užklausa veiktu.

Kai šios taisyklės patenkintos INSERT ir UPDATE užklausos veikia taip pat kaip su paprastu stalu. Nors šias užklausas leistumėt ant View, už širmos duombazė pasirūpina teisingai modifikuoti stalus kurie sudaro nurodyta View. Taigi, betkokie pakeitimai ant View taip pat matysis ant stalų iš kurių ji padaryta

Pridedant WITH CHECK OPTION savybę gale CREATE VIEW užklausos galima padayti, kad naujos INSERT ir UPDATE užklausos tenkintu logines taisykles nurodytas ``WHERE` dalyje. Taigi sukuriant toki view:

CREATE VIEW seniors as SELECT p.name, p.age from person p

where age > 50 WITH CHECK OPTION ;

Nauji INSERT'ai ar UPDATE'ai į senios neveiktų jei amžius mažiau nei 51

Atreipkit dėmesį, kad jei WITH CHECK OPTION nebūtu, INSERT'ai ar UPDATE'ai su amžium > 50 veiktų, tačiau kai SELECT'intumėt šį seniors View - jie nesimatytų, nes tiesiog View esančio SELECT'o taisyklės nėra tenkinamos.

Pagrindinė View funkcija yra supraprastinti sudėtingas, dažnai naudojamas užklausas ir apriboti bei apsaugoti duombazės naudotoją kad jis nesukurtų neteisingų užklausų.

### Indeksai

Indeksai yra specialios paieškos lentelės kurių tikslas yra pagreitinti užklausų veikimą panaudojant įrašų indeksavimą panašiu būdu kaip indeksas veikia knygos gale ieškoti reikiamos informacijos. Indeksai pagreitina SELECT ir WHERE užklausų veikimą, tačiau įrašymas (UPDATE ir INSERT) būna šiek tiek lėtesnis.

Vieno stulpelio indekso sintakse atrodytų taip:

CREATE INDEX <any\_index\_name> ON <table\_name>(<column\_name>);

Taip pat gali egzistuoti UNIQUE indeksai:

CREATE UNIQUE INDEX <index\_name> on <table\_name>(<column1>, <column2>);

Tokie indeksai iš principo veikia taip pat kaip pridėjus UNIQUE savybė prie stulpelio kuriant lentelė: jei neleistu duplikuotų reikšmių įrašymo. Taip pat atreipkit dėmesi į **du stulpelius** šiame pavyzdyje. Indeksuoti galima du stuleplius, tai vadinamia composite index, tokie indeksai gerai veikia dažnai ieškant įrašų lentelėje su užklausomis naudojančiomis WHERE savybę.

Jei prie stulpelio pavadinimo pridėsit raktažodį DESC:

CREATE INDEX <any\_index\_name> ON <table\_name>(<column\_name> DESC);

indeksas lyginimą pradės nuo didesnių reikšmių, tai reiškia didesnės reikšmės bus randamos greičiau. Default ASC.

### Triggers

Trigeriai yra operacijos egzistuojančios pačioje duombazėje kurios vykdomos atsitikus nustatytam veiksmui. Pavyzdžiui tai gali būti operacija kuri modifikuoja įrašą pagal nustatytą tvarką kiekvieną kartą kai naujas įrašas būna INSERT'inams į stalą

Pavyzdys:

CREATE TRIGGER agecheck BEFORE INSERT ON person

FOR EACH ROW

IF NEW.age < 0

THEN SET NEW.age = 0;

END IF;

čia yra trigger'is kuris bus paleistas kiekvieną kartą kai nauja INSERT užklausa bus paleista ant Person stalo. Šis trigeris patikrins ar amžius įrašomo asmens nėra mažiau nei 0 ir jei yra, amžius bus nustatomas į 0 ir tik tada įrašas nutiks. Triggeriu sintaksė skiriasi nuo to ko norima pasiekti, bet iš esmės kiekvieno naujo trigerio užklausos pradžia bus CREATE TRIGGER <any\_trigger\_name> <..> Daugiau apie sintaksę galima sužinoti čia: <https://dev.mysql.com/doc/refman/5.7/en/trigger-syntax.html> Praktikoje triggerių reikėtų naudoti kuo mažiau, nes jie yra paleidžiami kiekvieną kartą kai kažkokia užklausa įvyksta ir taip lėtina duombazės darbą. Taigi trigerius stengiamsi naudoti tik labai rizikingose situacijose, akd padidinti saugumą arba tada kai tikimasi, kad labai dažnai reikės apdirbti naujas užklausas.

### Temporary Tables

Kartais būna situacijų kai mes norim kažką apskaičiuoti per kelias užklausas ir mums reikia kažkur laikyti informacija iš pirminių užklausų, kad jas patogiai paduoti į antrines užklausas. Tam į pagalbą ateina laikinosios lentelės. Laikinosios lentelės yra naudingos dirbant su tarpinėmis reikšmėmis (reikšmėmis kurios gautos atlikus kažkokią operaciją, bet jad dar reikės apdoroti, nes tai nėra ko reikia rezultate).  
Laikinosios lentelės visada bus ištrintos kai prisijungimo sesija baigsis ir SHOW TABLES užklausa jų nerodo.  
Sintaksė tokių lentelių yra tokia pat kaip CREATE TABLE sintaksė, tik šiuo atvėju reikia nurodyti, kad tai yra laikina lentelė tokiu būdu: CREATE TEMPORARY TABLE <table\_name> (<column\_definitions>)

### Functions and procedures

MySQL turi nemažai funkcijų kurias galima naudoti užklausose. Funkcijos kurios yra jau duombazėje paruoštos naudojimui vadinamos native funkcijomis. Apie kelias jau tikriausiai žinote, tokias kaip SUM(), MAX() ir taip toliau. Funkcijų sąrašą ir ką jos daro galite rasti čia: <https://www.w3schools.com/sql/sql_ref_mysql.asp> Taip pat egzistuoja procedūros. Tai yra operacijos kurios atlieką kažkokį veiksmą tačiau negali gražinti reikšmes stulpely, ne taip kaip funkcijos. Procedūros dar gali manipuliuoti pačia duombaze, tuo tarp funkcijos dirba su data.

CREATE FUNCTION sintksė yra ganėtinai sudėtinga ir ją galite rasti čia: <https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html> Pavizdys mūsų duombazėje galėtų atrodyti taip:

CREATE FUNCTION addHello (s CHAR(20))

RETURNS CHAR(50) DETERMINISTIC

RETURN CONCAT('Hello, ',s,'!');

Tuomet toks panaudojimas:

SELECT addHello(name) FROM Person;

Grąžintų:

\----------------/

| addHello(name) |

------------------

| Hello, Jonas! |

| Hello, Tomas! |

| Hello, Egle! |

| Hello, Toma! |

| Hello, Ieva! |

/----------------\

Tuo tarpu CREATE PROCEDURE pavyzdys atrodytų taip:

CREATE PROCEDURE setEntryNum(OUT param1 INT)

BEGIN

SELECT COUNT(name) INTO param1 FROM employee;

END;

Čia yra procedūra kuriai yra paduodamas kintamasis ir ji į tą kintamajį įrašo nustatytą reikšme, šiuo atvejų name įrašų skaičių iš Employee lentelės Patestuoti kaip veikia galima taip:

call setEntryNum(@a); -- < naujas kintamas @a

select @a -- < pasirenkama kitamojo `a` reikšme kuri tiesiog grąžintų "SELECT COUNT(name) INTO param1 FROM employee" rezultatą

\-----/

| @a |

-------

| 7 |

/-----\

Yra įmanoma kurti savo kompiliuotas funkcijas ir įkelti į duombazę, tokios funkcijos vadinas UDF - user defined functions. Tačiau jos rašomos C arba C++ kalba ir tuo užsiima pagrinde tik duombazių ekspertai. Daugiau apie tai jei įdomu: <https://dev.mysql.com/doc/refman/8.0/en/adding-udf.html>

### Hibernate <https://www.tutorialspoint.com/hibernate/orm_overview.htm>

* Configuration
* Entity
* Entity Mapping
* Persistence
* HQL
* Criteria
* Caching

### Configuration

Gali būti daug konfiguracijos būdų:

The Configuration object provides two keys components −

* **Database Connection** − This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.
* **Class Mapping Setup** − This component creates the connection between the Java classes and database tables.

## Hibernate with MySQL Database

MySQL is one of the most popular open-source database systems available today. Let us create **hibernate.cfg.xml** configuration file and place it in the root of your application's classpath. You will have to make sure that you have **testdb** database available in your MySQL database and you have a user **test**available to access the database.

The XML configuration file must conform to the Hibernate 3 Configuration DTD, which is available at <http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd>.

<?xml version = "1.0" encoding = "utf-8"?>

<!DOCTYPE hibernate-configuration SYSTEM

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name = "hibernate.dialect">

org.hibernate.dialect.MySQLDialect

</property>

<property name = "hibernate.connection.driver\_class">

com.mysql.jdbc.Driver

</property>

<!-- Assume test is the database name -->

<property name = "hibernate.connection.url">

jdbc:mysql://localhost/test

</property>

<property name = "hibernate.connection.username">

root

</property>

<property name = "hibernate.connection.password">

root123

</property>

<!-- List of XML mapping files -->

<mapping resource = "Employee.hbm.xml"/>

</session-factory>

</hibernate-configuration>

1. JPA konfiguracija. Ši konfiguracija skirta Java Persitance API, tačiau pakeitus <provider> tag'o reikšmę galima nustatinėti Hibernate konfiguraciją.  
   Tai padaryti classpath reikia turėti direktoriją/failą: META-INF/persistance.xml

<https://www.javatpoint.com/example-to-create-hibernate-application-in-eclipse-ide>

<?xml version="1.0" encoding="UTF-8" ?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence

http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd" version="1.0">

<persistence-unit name="my-pu">

<description>My Persistence Unit</description>

<provider>org.hibernate.ejb.HibernatePersistence</provider>

<properties>

<property name="hibernate.dialect"

value="org.hibernate.dialect.MySQLDialect"/>

<property name="javax.persistence.jdbc.url"

value="jdbc:mysql://localhost:3306/playground"/>

<property name="javax.persistence.jdbc.user" value="admin"/>

<property name="javax.persistence.jdbc.password" value="admin"/>

</properties>

</persistence-unit>

</persistence>

|  |
| --- |
| * **hibernate.cfg.xml**.   <?xml version="1.0" encoding="UTF-8"?> |
|  | <!DOCTYPE hibernate-configuration PUBLIC |
|  | "-//Hibernate/Hibernate Configuration DTD 3.0//EN" |
|  | "http://hibernate.org/dtd/hibernate-configuration-3.0.dtd"> |
|  | <hibernate-configuration> |
|  | <session-factory> |
|  | <property name="connection.driver\_class">org.h2.Driver</property> |
|  | <property name="connection.url">jdbc:h2:mem:test;DB\_CLOSE\_ON\_EXIT=FALSE;AUTO\_RECONNECT=TRUE</property> |
|  | <property name="dialect">org.hibernate.dialect.H2Dialect</property> |
|  | <property name="hibernate.current\_session\_context\_class">thread</property> |
|  | <property name="hibernate.cache.use\_structured\_entries">true</property> |
|  | <property name="hibernate.javax.cache.missing\_cache\_strategy">create</property> |
|  |  |
|  | <!-- POOL MANAGEMENT --> |
|  | <property name="hibernate.c3p0.min\_size">1</property> |
|  | <property name="hibernate.c3p0.max\_size">2</property> |
|  |  |
|  | <!----> |
|  | <property name="hbm2ddl.auto">create</property> |
|  | <!--<property name="hibernate.hbm2ddl.import\_files">sql/tables.sql</property>--> |
|  | </session-factory> |
|  | </hibernate-configuration> |

1. **hibernate.properties**  **hibernate.connection.url**=**jdbc:mysql://localhost:3306/Pavyzdys?serverTimezone=UTC  
   hibernate.connection.username**=**Akvile  
   hibernate.connection.password**=**akvile  
   hibernate.c3p0.min\_size**=**5  
   hibernate.c3p0.max\_size**=**20  
   hibernate.c3p0.timeout**=**1800  
   hibernate.c3p0.max\_statements**=**50  
   hibernate.dialect**=**org.hibernate.dialect.MySQLDialect  
   hibernate.show\_sql**=**true**
2. Kitas variantas yra turėti hibernate.cfg.xml failą savo classpath'e.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/playground?serverTimezone=UTC</property>

<property name="hibernate.connection.username">medardas</property>

<property name="hibernate.connection.password">pass</property>

<property name="hibernate.connection.pool\_size">1</property>

<property name="hibernate.current\_session\_context\_class">thread</property>

<property name="hibernate.show\_sql">true</property>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

</session-factory>

</hibernate-configuration>

Maven build settings

<https://bintray.com/hibernate/artifacts/hibernate-orm>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-agroal</artifactId>

<version>5.4.2.Final</version>

<type>pom</type>

</dependency>

<https://www.tutorialspoint.com/hibernate/hibernate_environment.htm>

Tuomet jungiantis prie duombazės sukurti Configuration objektą iš kurio galima gauti Session objektą skirtą manipuliuoti Entity objektais duombazėje:

Configuration config = new Configuration().configure(); //pridėti klases visu sukurtu Entities;

Session session = config.buildSessionFactory().getCurrentSession();

Komentare //pridėti klases visu sukurtu Entities; turima omeny pridėti visass sukurtas entities klases per 'addAnnotatedClass(Class class)' Pvz:

Configuration config = new Configuration()

.addAnnotatedClass(Person.class)

.configure();

Atreipkite dėmesį, kad configure() pridedama po anotuotų klasių pridėjimo.

Session session = factory.openSession();

Transaction tx = null;

try {

tx = session.beginTransaction();

// do some work

...

tx.commit();

}

catch (Exception e) {

if (tx!=null) tx.rollback();

e.printStackTrace();

} finally {

session.close();

}

Configuration cfg = **new** Configuration()  
 .addAnnotatedClass(Person.**class**)  
 .addAnnotatedClass(Salary.**class**);  
SessionFactory sf = cfg.buildSessionFactory();  
Session session = sf.openSession();  
  
  
Query query = session.createQuery(**"from Person"**);  
  
List<Person> list = query.list();  
**for** (**int** i = 0; i < list.size(); i++) {  
 System.***out***.println(list.get(i).getId() + **" "** + list.get(i).getName() + **" "** + list.get(i).getAge());  
}  
  
session.disconnect();

Čia yra standartinis Hibernate configuracijos variantas. Sukurti Session galima lygiai tokiu pačiu būdu kaip priš tai.

### Entity

Enitity yra Java objektas kuris turi tiesioginį sąryšį savo field'ais su lentelės stulpeliais duombazėje. Taigi jei tarkim egzistuoja lentelė:

\-------------------/

| Person |

---------------------

| ID | Name | age |

--------------------

| 1 | Jonas | 10 |

| 2 | Ieva | 18 |

/-------------------\

Jos Entity bū java objektas su anotacija @Entity

@Entity

@Data

@NoArgsConstructor

public class Person {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String name;

private int age;

}

Čia Hibernate framework'as išsiaiškina kokios yra stulpelių reikšmės pagal tai kokie Entity objekte yra field'ų pavadinimai: id, name, age. Taip pat išsiaiškina stalo pavadinimą.

* @Id nusako, kad tai yra PRIMARY KEY reikšmė.
* @GeneratedValue(strategy = GenerationType.IDENTITY) nusako, kad ši reikšmė yra auto-generuojama.

### Entity mapping

Viena sudėtingiausių dalių dirbant su hibernate yra teisingai sumappinti java objektus su lentlėmis. Tačiau teisingai sukūrus šį sąryšį galima labai pasilengivnti darbą su savo sistema. Taigi, jei egzituoja antra lentelė kur PersonId yra Foreign Key į Person(id).

\------------------/

| Salary |

--------------------

| PersonId | Pay |

--------------------

| 2 | 1000 |

| 1 | 1200 |

/------------------\

Kad ją galėtume surišti su Person objektu mūsų naujas Entity gali atrodyti taip:

@Entity

@Data

@NoArgsConstructor

public class Salary {

@Id

private int personId;

private int pay;

@OneToOne(cascade = CascadeType.ALL)

@PrimaryKeyJoinColumn(name = "personId", referencedColumnName = "id")

private Person person;

}

O Person turėtų gauti nauja field'ą:

@OneToOne(mappedBy = "person")

private Salary salary;

Šis detales hibernate supranta taip:

* Hiberate žino, kad Person Primary key yra Id
* Jis taip pat žino, kad primary key pas Salary yra personId
* Sukurtas naujas field'as salary Objekte Person nurodo, kad Person turi Salary ir jų ryšys yra abiejų objektų Primary Key, tai yra Person.id = Salary.personId
* cascade = CascadeType.ALL reiškia, kad jei rekšmė pasikeis šiame objekte, ji taip pat pasikeis ir Person objekte ir atitinkamais pakeitimais duombazėje bus pasirūpinta.

### Persistence

Session session = hibernateCfg.buildSessionFactory().openSession();

session.getTransaction().begin();

Person person = new Person();

person.setName("Petras");

person.setAge(30);

session.save(person);

Salary salary = new Salary();

salary.setPay(1000);

salary.setPersonId(person.getId());

session.save(salary);

session.getTransaction().commit();

session.close();

Šiame kodo gabaliuke pirmiausia yra sukuriama sesijos Session objektas iš konfiguracijos kuri būvo nusakyta ankstesniame skyriuje. Tuomet norint pradėti darbą su Enity objektais ir jų įrašymu į duombazę reikia atidaryti transakcija. Transakcija yr atidaroma su session.getTransaction().begin(); ir baigiama su session.getTransaction().commit(); Taigi norint daryti update, insert ir delete komandas - jas būtina daryti dar atidarytos transakcijos. Select tipo statement'am tai nebūtina, galima tiesiog naudoti funkcijas tokias kaip session.find(class, id)

Atsijungimas nuo duombazės visiškai yra tvarkomas kaip sesijos uždarymas: session.close();;

### HQL - Hibernate Query Language

Tai yra į objektus orentuota užklaušų kalba (object-oriented query language) kuri yra panaši į SQL tačiau vietoj lentelių ir stulpelių ji dirba su objektais ir jų field'ais. Hibernate išverčia šiuos objektus į standartinę SQL užklausą ir siunčia tai į duombazę. IntelliJ netgi gali debuginti šitas užklausas java failuose.

Nors per Hibernate galima leisti ir Native SQL užklausas, naudodami HQL mes galime naudotis hibernate caching galimybėmis ir šiek tiek mažiau nerimauti apie teisingas užklausas.

Lentelių ir field'u pavadinimai yra case sensitive, tačiau SQL raktažodžiai nėra.

String hql = "FROM Employee";

Query query = session.createQuery(hql);

List<Employee> results = query.list();

Tokia programa pirmoje eilutėje nusako su kokiu objektu reikia dirbti (Employee), tuomet antroje - sukurti SQL užklausą iš hibernate sesijos ir paskutinis dalykas yra iškviesti list() metoda kuris gražina paprastą SQL (SELECT \* FROM Employee) rezultatą kaip sąrašą.

Lygiai toks pats rezultatas būtu su:

String hql = "FROM com.codeacademy.hibernatetutorial.model.Employee";

Query query = session.createQuery(hql);

List<com.codeacademy.hibernatetutorial.model.Employee> results = query.list();

Tokiu būdu galima sukurti praktiškai betkokią HQL užklausą visom CRUD operacijoms. Reikia atsiminti, kad tai yra tas pats kas SQL tik dirbant su objektais ir jų field'ais reikia galvoti apie juos kaip apie stalus ir stulpelius tiesiogiai.

##### Named parameters

HQL užklausose galima nustatinėti parametrus:

String hql = "FROM Employee E WHERE E.id = :employee\_id";

Query query = session.createQuery(hql);

query.setParameter("employee\_id",10);

List results = query.list();

Tokiu būdu sugalvojame parametro pavadinimą ir idedame jį su dvidatškiu, kad hibernate suprastu tai kaip parametrą (:employee\_id) Tuomet nustatyti parametro reikšmę galima programiniu būdu query.setParameter("employee\_id",10);

### Criteria

Criteria yra objektas kuris gali per programines funkcijas sukurti kompleksią užklausą kuri po to išverčiama į SQL kalbą. Hibernate į inicijuoti kriterijų galima taip:

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);

Root<Employee> employeeRoot = criteriaQuery.from(Employee.class);

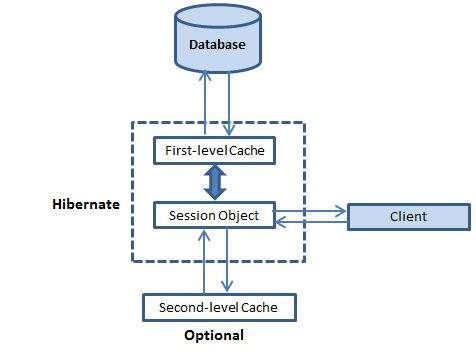
criteriaQuery.where(criteriaBuilder.equal(employeeRoot.get("name"), "Jooohn"));

List<Employee> employees = session.createQuery(criteriaQuery).getResultList();

Čia pirmiausia sukuriamas CriteriaBuilder objektas kuris gali sukurti užklausas ir jose lyginti loginius paramtrus. Užklausos yra kuriamos (CriteriaQuery) remiantis kažkokia lentele, kadangi mes norim sužinoti apie Employee, mes nurodome atitinkamos Enity klase ir sukuriam SELECT užklausą (krierijai gali būti tik užklausose SELECT, DELETE ir UPDATE. INSERTlogiškai neturi kriterijų) su kriterijais link Employee entity. Prie šio objekto veliau bus pridedami papildomi kriterijai užklausai.  
Kitas objektas kurį sukuriam yra Employee objektas kaip užklausos dalis. tai reiškia šį objektą galima naudoti kai norima nustatyti kokiam stulpeliui kriterijus turi tikti. Stulpelį galima nurodyti iškviečiant employeeRoot.get(<column\_name>), tai yra objektas prie kurio yra kabinami kriterijai. Kuomet baigiama krterijų objetkas - jis paleidžiamas kaip užklausa naudojant session.createQuery(criteriaQuery).getResultList();

### Caching

Caching (kešavimas) yra mechanizmas kuris šiek tiek pagreitina užklausų rezultatų gavima. Tai yra sistema kuri egzistuoja tarp aplikacijos ir duombazės ir saugo naujausias užklausas siųstas iš kliento. Jei nepraeina nustatytas laikas rezultatai į tokias pačias užklausas yra gražinami tiesiai iš kešo ir nėra ištikrųjų siunčiami į duombazę taip sutaupant daug laiko.

[](https://camo.githubusercontent.com/98a29a3740929793e2ce88148369cc09f2aabe29/68747470733a2f2f692e696d6775722e636f6d2f41507446674e5a2e6a7067)

##### Kaip kešavimas veikia?

Hibernate framework'e yra du kešavimo lygiai: 1st level cache ir 2nd level cache. Pirmas kešavimo lygis yra įjungtas automatiškai, antrą lygį reikia sukonfiguruoti. Jei abi kešavimo sistemos įjungtos ENtity paieška veikia taip:

* Kiekvieną kartą kai hibernate sesija bando užkrauti Entity pirmas dalykas kur ji žiuri tai yra pirmo lygio kešas kuris tvarkomas Session objekte.
* Jei toks enity egzistuoja pirmame lygije tuomet tas jis yra tiesiog gražinamas kaip rezultatas.
* Jei to enity nėra pirmo lygio keše tuomet yra žiūrima į antro lygio kešą.
* Jei šis enity egzistuoja antro lygio keše - jis gražinamas kaip rezultatas. Tačiau prieš tai padarant objektas taip pat yra išaugotas pirmo lygio keše, kad nereikėtų daryti vėl paieškos antrą kartą.
* Jei Enity neegzisutoja neiviename keše, užklausa į duombazę yra išsiunčiama. Gautas rezultatas yra įrašomas į abu kešo lygius.
* Antro lygio kešavimo sistema atnaujinta pakeistą objektą, jei pakeitimas buvo daromas naudojant hibernate sesijos api.
* Jei įrašai duombazėje pasikteitė per laiką kol neišseko enity kešavimo nustatymas timeToLiveSeconds - kešavimo sistemai nėra kaip žinoti apie tą pasikeitimą ir tiesiog senas rezultatas bus gražintas iš kešo.

If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.

##### 1. First-level cache

Šis cache lygis yra Session objekto lygije. Keli dalykai žinoti apie 1st level cache:

1. Hibernate First Level Cache yra įjungta automatiškai, tam konfiguracijos nereikia.
2. Šis cache yra specifiškas sesijai. Pavyzdžiui dėl to jei du kartus iš eilės padarysim tą patįsession.find() antrą kartą užklausa nebus išsiųsta į duombazę, tačiau jei naujoje sesijoje paleisim tą pačia užklausa iškarto - ji bus išsiųsta.
3. Hibrenate First Level Cache glai turėti senų reikšmių.
4. Galima naudoti šiuos sesijos metodus, kad manipuliuoti cache:
   * evict(Object) - išmesti tam tikrą objektą.
   * clear() - išvalyti viską.
   * contains(Object) - patikrina ar tam tikras objektas yra cache. Gražina boolean.
5. Kadangi hibernate saugo visas užklausas darant daug pakeitimų, bulk updates, kai masiškai keičiami įrašai - būtina pravalyti cache dėl galimų atminties problemų.

##### 2. Second-level cache

Antro lygio cache (kešas) galima sukonfiguruoti laikyti tik tam tikro Entity detales, užklausas, taip pat pasirūpinti , kad cache būtų išsaugotas Session objekte net ir išnaujo prisijungus.

Prieš įjungiant second-level caching pirmiausia turim nutarti kelis dalykus. Pirmiausia reikia išsirinkti Concurrency Strategytai yra algoritmas kurį pasirinksim manipuliuoti objektams egzistuojantiems keše:

* read-only - tokio tipo kešas gerai tinka situacijoms kur dirbama su objektais kuriuos dažnai skaitom bet nemodifikuojam. Ši strategija yra ganėtinai paprasta ir greita.
* nonstrict read-write - kaikurios aplikacijos tik kartais turi modifikuoti informaciją. Dėl to sitiuacijų kur du skirtingi sistemos komponentai bandytų pakeisti tą patį objektą vienu metu nelabai tikėtinos. Taigi tokiom situacijom tinka nonstrict read-write strategija.
* read-write - tokia kešinmo strategija yra geriausia kai programa dažnai turi keisti informaciją.
* transactional - tokia strategija palaiko tam tikrus kešo privider'ius tokius kaip JBossTreeCache kuie turi savo distinktyviū informacijos kešinimo aspektų.

Taigi jei norėtume pridėti Read-Write strategijos second-level cache prie Entity objekto darytume taip:

@Entity

@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)

public class Employee{<..>

Kitas dalykas kurį reikia išsirinkti yra Cach Provider. Tai yra klasė kuri pasirūpina pačiu "kešavimu" (chaching), tai yra saugo atmintyje užklausas ir gražina jų rezultatus jei nauja užklausa pakankamai nauja. Yra keli cache providers:

* EHCache
* OSCache
* SwarmCache
* Jboss Cache

Ne kiekvienas provideris palaiko visas concurrency strategijas. Iš išvardintų tik EHCache ir OSCache palaiko Read-Write. Taigi norit pridėti EHCache klasę į konfiguracijos (hibernate.cfg.xml) failą pridedam tokiąreikšmę:

<property name = "hibernate.cache.provider\_class">

org.hibernate.cache.EhCacheProvider

</property>

arba į hibernate.properties: hibernate.cache.provider\_class=org.hibernate.cache.EhCacheProvider

Tokiu būdu mes įjungėm second-level caching valdoma EHCache providerio su Employee objektu skaitymu tvarkomo read-write concurrency strategy tipo algoritmu.

##### Keletas pagrindinių funkcijų

* session.flush() - sinchronizuoja visus objektus užkešuotus pirmo lygio kešinge su duombaze.
* session.evict(Object) - išima objektą iš pirmo lygio kešo.
* session.clear() - išmeta visus objektus iš pirmo lygio kešo.
* session.getSessionFactory().getCache().evictQueryRegion( "<region\_name>" ) išmeta užklausas pagal duota regiona

# Mongo DB

Mongo Database yra multiplatforminė, non-relationl ir orentuota į dokumentus duombazės sistema. Šį non-relational (arba NoSQL) modelį naudoja daug skirtingų duombazių, tačiau Mongo DB dirba būtent su dokumentais kurie yra reprezentuojami JSON formatu.

### What is Non-relational database?

Non-relational duombazė nenaudoja tradicinio įrašų saryšio taisyklių kokias turi RDBMS tokie kaip MySQL, PostgreSQL ir t.t

Populiarėjančios ne reliacinės duombazės buvo sukurtos siekiant įveikti BigData poreikius. Big Data yra duomenys kurie auga pernelyg sparčiai ir yra pernelyg įvairiai struktūrizuti, kad įprastos technologijos susitvarkytu.

Egzistuoja visokios NoSQL technologijos ir jos stipriai skirias tarpusavy, bet tokį modelį naudojančios duombazės tipiškai lagviau plečiamos (scaling) ir yra bendrai lankstesnės. Tai yra non-relationl duombazių evoliucijos (iš relational) pasekmė. Ši evoliucija remiasi tam tikromis savybėmis:

* Data models: Skirtingai nuo relational, schemos NoSQL duombazėje yra daug lankstesnės. To rezultatas yra daug paprastesnė įrašų manipuliacija ir duombazės prisitaikymas priklausomai nuo sistemos reikalavimų.
* Data structure: Non-relational duombazių dizainas leidžia tokiom duombazės laikyti nestrukturizuotus įrašus kurių nebutinai įmanoma gražiai sudėti į lenteles ir stulplelius. Nestrukturizuotos informacijos suvokimas yra svarbi savybė nes dauguma informacijos sugeneruota šiais laikais yra nestrukturizuota.

### Savybės bei Funkcijos

Minimalus duombazės paleidimo ir pirmo naudotojo sukūrimo procesas yra nusakytas kurso skaidrėse (prasideda maždaug ties 42 skaidre).

##### Kolekcijos (Collections)

Kaip minėta MongoDB duombazė savyje laiko dokumentus JSON formatu. Šių dokumentų rinkinys vadinasi kolekcija - collection. Kolekciją galima prilyginti lentelėms SQL tipo duombazėje, tuo tarpu JSON dokumentai būtų artimiausia analogija įrašams lentelėje. Toks būdas tvarkyti informacijai yra bendra daugumai NoSQL duombazių.

Taigi, kad į MongoDB įdėti įraša pirmiausia reikia sukurti kolekciją:

db.createCollection('firstCollection');

##### Dokumentai

MongoDB gali laikyti beveik betkokį JSON dokumentą betkokioje kolekcijoje, o jo įrašymas per konsolę atrodytų taip:

db.firstCollection.insertOne({name:'Medardas',skill:'not MongoDB'});

Kai dokumentas yra įdedamas į duombazę, jam yra sugeneruojamas unikalus ID kurį nuo šiol matysit suradę dokumentą.

##### Find

Patikrinti įrašus šioje kolekcijoje galima taip:

db.firstCollection.find();

Ši funkcija find() taip pat gali priimti argumentus, argumentai nusakytų salygas pagal kurias reikia rasti objektus. Pavyzdžiui:

db.firstCollection.find({name:'Medardas'});

Apie šia funkciją galima galvoti kaip apie SELECT užklausą SQL duombazėje kur kaikuriuos WHERE loginius lyginimus mes perduodam kaip argumentus. Taigi šiuo atvėju funkcija ieškotų visų dokumentų kolekcijoje firstCollection ir gražintų visus kur yra laukas name su reikšme Medardas. Beja paieška yra case sensitive.

Laukų skaičius pagal kuriuos ieškoti neturi ribų, taigi iš principo jūsų paieškos objektas gali būti tiek didelis kiek reikia.

###### Projection

Antras funkcijos parametras vadinasi projekcija. Jis priima JSON objektą kuris nusako kuriuos laukus rezultate norime matyti plius objekto \_id (nos \_id grąžinimą irgi galima išjungti specifiškai tai nustačius). Šio JSON objekto formatas yra {<field\_name>: <reikšmė>} Reikšmės gali būti 1 arba 0 kas atitinkamai reiškia norime matyti ir nenorime matyti lauko

Norint nustatyti, kad negrąžintų \_id fieldo ir grąžintų skill galima daryti taip:

db.firstCollection.find({name:"Medardas"},{\_id: 0, skill:1});

Gražintų:

{ "skill" : "not MongoDB" }

###### Regex

Find funkcija tiap pat supranta paieškos objektus su regex'ais. Taigi norint rasti visus dokumentus kur egzistuoja skill'as turintis "Mongo" savyje galyma rašyti taip:

db.firstCollection.find({skill:/.\*Mongo.\*/})

###### Loginės operacijos

Loginės operacijos Mongo db yra nusakomos naudojant operatorius JSON objektuose kuriuos naudojat kaip paieškos parametrus.

Sintaksė:

{ $<operator>: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }

Taigi surasti žmogu su skill'u "MongoDB" **arba** skill'u panašiu į HR galima taip:

db.firstCollection.find({$or: [{skill:'MongoDB'},{skill:/.\*HR.\*/}]});

Logical operatoriai yra tokie:

* and - turi tikti visi išsireiškimai
* not - turi išsireiškimas
* nor - turi netikti neivienas išsireiškimas
* or - gali tikti betkuris išsireiškimas
* Installing server
* Basic configuration
* Starting server
* Basic functions
* More about MongoDB design

Installing on Linux Debian systems

* Java client implementation
* **$** echo "deb [ arch=amd64 ] https://repo.mongodb.org/apt/ubuntu bionic/mongodb-org/4.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.0.list
* **$** gpg --keyserver hkp://keyserver.ubuntu.com:80 --recv-key 68818C72E52529D4
* **$** sudo apt-get update && sudo apt-get upgrade
* **$** sudo apt install -y mongodb-org

Setting up server

* Create director to store database data:

**$** cd /  
**$** sudo mkdir data  
**$** sudo mkdir db

* Give owning permission to current user

**$** cd /data/db  
**$** sudo chown -R <your\_UBUNTU\_user\_id> .

Starting server

Start server with default configuration run following:   
**$** mongod

Or start it as a fork process:  
**$** mongod --fork --logpath /logs/mongodb/mongodb.log

To stop the server:

**$** mongod –shutdown

Downloading client and Creating mongodb user

Download MongoDB Compass - GUI for mongodb client:  
<https://www.mongodb.com/download-center/compass?jmp=hero>Notes: windows users should select platform option “Windows 64-bit (7+) (not MSI or ZIP).

To create user:  
1. In your terminal connect to mongo “tutorial” database using mongo command line client:  
**$** mongo tutorial  
Above command will connect to new unconfigrued mongo database as admin user.

Below command creates new user to work with:

$ use admin  
**$** db.createUser({user:"<user\_name>", pwd:"<password>", roles:[{role:"root", db:"admin"}]})

{

"id": 21,

"first\_name": "Vita",

"last\_name": "Tedahl",

"email": "vtedahlk@gmpg.org",

"gender": "Female",

"ip\_address": "236.250.108.184",

"work": {

"name": "IBM",

"working\_since": "01-01-2000"

}

}  
db.firstCollection.insertOne({"id": 21,"first\_name":"Jonas","last\_name":"Jonauskas","email":"jonjon@gmpg.org","gender":"Male","ip\_address":"135.90.108.184","work":{"name": "IBM","working\_since":"01-01-2000","salary": {"pay": 1000,"payday": 29}}})

SQL Basics

* Create database

CREATE DATABASE playground;

In order to work with a database it must be selected as working database in the session:  
  
USE playground;

* Create table

CREATE TABLE person (id INT(10), name VARCHAR(40), age INT(3));

SQL CRUD operations

* *Create*: Add entries to the table

INSERT INTO person (name, age) VALUES('Jonas', 30);

* *Read:* Select entries from the table

SELECT \* FROM person;

* *Update:* Update entries

UPDATE person SET id=0 WHERE `name`='vardas';

* *Delete:* Delete Entries

DELETE FROM person WHERE id=0;

<repositories>

<repository>

<id>projectlombok.org</id>

<url>https://projectlombok.org/edge-releases</url>

</repository>

</repositories>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>edge-SNAPSHOT</version>

<scope>provided</scope>

</dependency>

Ctrl+alt+s plugins +lombok

Kad nereiktu ideti get set

@Data  
**public class** Person **implements** Comparable<Person> {

Pries klase tada nereiks get ir set

Some of The Most Important SQL Commands

* **SELECT** - extracts data from a database
* **UPDATE** - updates data in a database
* **DELETE** - deletes data from a database
* **INSERT INTO** - inserts new data into a database
* **CREATE DATABASE** - creates a new database
* **ALTER DATABASE** - modifies a database
* **CREATE TABLE** - creates a new table
* **ALTER TABLE** - modifies a table
* **DROP TABLE** - deletes a table
* **CREATE INDEX** - creates an index (search key)
* **DROP INDEX** - deletes an index
* **SELECT** \* **from** employee;
* **SELECT** \* **from** currency;
* **SELECT** \* **from** time\_sheet;
* **SELECT**
* employee.id,
* employee.**NAME**,
* employee.valiuta,
* employee.hourly
* \* (**SELECT** time\_sheet.hours **FROM** time\_sheet **WHERE** time\_sheet.employee1\_id = employee.id)
* **FROM** employee
* **JOIN** time\_sheet
* **ON** employee.id=time\_sheet.employee1\_id;
* **SELECT**
* employee.id,
* employee.**name**,
* employee.valiuta **AS** kuras,
* (employee.hourly
* \* (**SELECT** time\_sheet.hours **FROM** time\_sheet **WHERE** time\_sheet.employee1\_id = employee.id)
* \* (**SELECT** rate **FROM** currency **WHERE** currency = 'EUR')) **AS** salary
* **FROM** employee
* **JOIN** time\_sheet
* **ON** employee.id=time\_sheet.employee1\_id
* **ORDER** **BY** salary **DESC**
* **LIMIT** 1;
* **SELECT**
* employee.id,
* employee.**name**,
* employee.valiuta (**INSERT** **INTO** employee(valiuta) **SELECT** currency **FROM** currency **WHERE** 'EUR') **AS** kuras,
* (employee.hourly
* \* (**SELECT** time\_sheet.hours **FROM** time\_sheet **WHERE** time\_sheet.employee1\_id = employee.id)
* \* (**SELECT** rate **FROM** currency **WHERE** currency = 'EUR')
* \* (**SELECT** **FROM** currency **WHERE** currency = 'EUR' ) **AS** salary
* **FROM** employee
* **JOIN** time\_sheet
* **ON** employee.id=time\_sheet.employee1\_id
* **ORDER** **BY** salary **DESC**
* **LIMIT** 1;

**CREATE** **TABLE** employee1 (**NAME** **VARCHAR**(40), hourly **INT**(10), **position** **VARCHAR**(40), currency **VARCHAR**(40),on\_vacation **BOOLEAN**);

**SELECT**

**insert** **into** employee1 (**name**, hourly, **position**, currency, on\_vacation)

**values** ('Jonas', 24.0, 'developer', 'USD', **FALSE**);

**SELECT**

**insert** **into** employee1 (**name**, hourly, **position**, currency, on\_vacation)

**values** ('Ieva', 24.0, 'HR', 'EUR', **TRUE**);

**SELECT**

**insert** **into** employee1 (**name**, hourly, **position**, currency, on\_vacation)

**values** ('Mantas', 1903.4, 'HR', 'RUB', **FALSE**);

**SELECT**

**insert** **into** employee1 (**name**, hourly, **position**, currency, on\_vacation)

**values** ('Egle', 21.0, 'ceo', 'USD', **FALSE**);

**SELECT**

**insert** **into** employee1 (**name**, hourly, **position**, currency, on\_vacation)

**values** ('Baxteris', 0.99, 'office dog', 'EUR', **FALSE**);

**SELECT**

**insert** **into** employee1 (**name**, hourly, **position**, currency, on\_vacation)

**values** ('Matas', 45.0, 'developer', 'EUR', **TRUE**);

**SELECT**

**insert** **into** employee1 (**name**, hourly, **position**, currency, on\_vacation)

**values** ('Matew', 36.0, 'developer', 'USD', **false**);

**create** **table** Address

(

id **int** **primary** **key** **AUTO\_INCREMENT**,

street **varchar**(40),

city **varchar**(30)

);

**create** **table** Company

(

id **int** **primary** **key** **AUTO\_INCREMENT**,

**name** **varchar**(40),

address **int**,

**CONSTRAINT** fk\_company\_address **FOREIGN** **KEY** (address) **REFERENCES** Address (id)

);

**create** **table** Employee

(

id **int** **primary** **key** **AUTO\_INCREMENT**,

**name** **varchar**(40),

**position** **varchar**(30),

company **int**,

address **int**,

**CONSTRAINT** fk\_Employee\_address **FOREIGN** **KEY** (address) **REFERENCES** Address (id),

**CONSTRAINT** fk\_Company **FOREIGN** **KEY** (company) **REFERENCES** Company (id)

);

**create** **table** Contact

(

id **int** **primary** **key** **AUTO\_INCREMENT**,

ref\_id **int**,

contact\_type **varchar**(30),

**value** **varchar**(40),

**CONSTRAINT** fk\_contact\_Person **FOREIGN** **KEY** (ref\_id) **REFERENCES** Employee (id)

);

**create** **table** Address

(

id **int** **primary** **key** **AUTO\_INCREMENT**,

street **varchar**(40),

city **varchar**(30)

);

**create** **table** Company

(

id **int** **primary** **key** **AUTO\_INCREMENT**,

**name** **varchar**(40),

address **int**,

**CONSTRAINT** fk\_company\_address **FOREIGN** **KEY** (address) **REFERENCES** Address (id)

);

**create** **table** Employee

(

id **int** **primary** **key** **AUTO\_INCREMENT**,

**name** **varchar**(40),

**position** **varchar**(30),

company **int**,

address **int**,

**CONSTRAINT** fk\_Employee\_address **FOREIGN** **KEY** (address) **REFERENCES** Address (id),

**CONSTRAINT** fk\_Company **FOREIGN** **KEY** (company) **REFERENCES** Company (id)

);

**create** **table** Contact

(

id **int** **primary** **key** **AUTO\_INCREMENT**,

ref\_id **int**,

contact\_type **varchar**(30),

**value** **varchar**(40),

**CONSTRAINT** fk\_contact\_Person **FOREIGN** **KEY** (ref\_id) **REFERENCES** Employee (id)

);

Prisijungimas prie duonbazės:

1. sudo apt-get update
2. sudo apt -y install mysql-server
3. sudo service mysql status
4. sudo service mysql start

**public class** Main {  
 **public static void** main(String[] args) {  
 Configuration cfg = **new** Configuration()  
 .addAnnotatedClass(Address.**class**)  
 .addAnnotatedClass(Company.**class**);  
 SessionFactory sf = cfg.buildSessionFactory();  
 Session session = sf.openSession();  
 Address address = **new** Address(**"Gedimino pr."**, **"Vilnius"**);  
 Company company = **new** Company();  
 company.setName(**"Microsoft"**);  
 company.setAddress(address);  
 session.getTransaction().begin();  
 session.save(address);  
 session.save(company);  
 session.getTransaction().commit();  
 session.disconnect();

*//import model.Address;  
//import model.Company;  
//import model.Contact;  
//  
//import model.Employee;  
//import org.apache.log4j.Logger;  
//import org.apache.log4j.xml.DOMConfigurator;  
//import org.hibernate.Session;  
//import org.hibernate.SessionFactory;  
//import org.hibernate.cfg.Configuration;  
//import org.hibernate.query.Query;  
//  
//import javax.persistence.criteria.CriteriaBuilder;  
//import javax.persistence.criteria.CriteriaQuery;  
//import javax.persistence.criteria.Root;  
//import java.util.List;  
//  
//import static com.mysql.cj.conf.PropertyKey.logger;  
//  
//public class Main {  
////  
////  
////  
//// public static void main(String[] args) {  
//  
//// Configuration cfg = new Configuration()  
//// .addAnnotatedClass(Address.class)  
//// .addAnnotatedClass(Company.class)  
//// .addAnnotatedClass(Employee.class)  
//// .addAnnotatedClass(Contact.class);  
//// SessionFactory sf = cfg.buildSessionFactory();  
//// Session session = sf.openSession();  
//// session.getTransaction().begin();  
//// Address address = new Address("Pilies g.", "Vilnius");  
////  
//// Company company = new Company();  
//// company.setName("Invega");  
//// company.setAddress(address);  
////  
//// Employee employee = new Employee();  
//// employee.setAddress(new Address("Pilies g.", "Vilnius"));  
//// employee.setCompany(company);  
//// employee.setName("Linas");  
//// employee.setPosition("IT");  
////  
//// Contact contact = new Contact();  
//// contact.setRef\_id(employee);  
//// contact.setContact\_type("phone");  
//// contact.setValue("867085236");  
////  
//// session.save(employee);  
////  
//// session.getTransaction().commit();  
//// session.disconnect();  
//  
//// employee\_list();  
//// }  
////  
//// public static void employee\_list() {  
//// Configuration cfg = new Configuration()  
//// .addAnnotatedClass(Address.class)  
//// .addAnnotatedClass(Company.class)  
//// .addAnnotatedClass(Employee.class)  
//// .addAnnotatedClass(Contact.class);  
//// SessionFactory sf = cfg.buildSessionFactory();  
//// Session session = sf.openSession();  
//// session.getTransaction().begin();  
//  
//// System.out.println("Employee with phone:");  
//// Query query = session.createQuery("from Contact where contact\_type='phone'");  
//// List<Contact> list = query.list();  
//// for (Contact c : list) {  
//// System.out.println(c.getEmployee().getName() + " " + c.getContact\_type() + " " + c.getValue());  
//// }  
//// System.out.println("Employee with 0 contacts:");  
//// Query query1 = session.createQuery("from Contact where contact\_type is null");  
//// List<Contact> list1 = query1.list();  
//// for (Contact c : list1) {  
//// System.out.println(c.getEmployee().getName() + " " + c.getContact\_type() + " " + c.getValue());  
//// }  
//// System.out.println("Employee with 1 contact:");  
////  
//// String q = "SELECT Employee.NAME, COUNT(Contact.employee) FROM Contact "  
//// + "GROUP BY Employee.NAME " + "HAVING COUNT( Contact.employee)=1";  
//// Query query2 = session.createQuery(q);  
//// List list2 = query2.list();  
//// System.out.println(list2);  
////  
//// session.disconnect();  
//// }  
//}*

**private static void** deleteEmployeesFromId (**int** delete) {  
 Configuration cfg = **new** Configuration()  
 .addAnnotatedClass(Address.**class**)  
 .addAnnotatedClass(Company.**class**)  
 .addAnnotatedClass(Employee.**class**)  
 .addAnnotatedClass(Contact.**class**);  
 SessionFactory sf = cfg.buildSessionFactory();  
 Session session = sf.openSession();  
 session.getTransaction().begin();  
  
 CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();  
 CriteriaDelete<Employee> employeeDelete = criteriaBuilder.createCriteriaDelete(Employee.**class**);  
 Root<Employee> root = employeeDelete.from(Employee.**class**);  
 employeeDelete.where(criteriaBuilder.equal(root.get(**"id"**), delete));  
 session.createQuery(employeeDelete).executeUpdate();  
 session.getTransaction().commit();  
  
 session.disconnect();

<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>8.0.15</**version**>  
</**dependency**>  
 <**dependency**>  
 <**groupId**>org.projectlombok</**groupId**>  
 <**artifactId**>lombok</**artifactId**>  
 <**version**>edge-SNAPSHOT</**version**>  
 <**scope**>provided</**scope**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>c3p0</**groupId**>  
 <**artifactId**>c3p0</**artifactId**>  
 <**version**>0.9.1.2</**version**>  
 </**dependency**>  
  
 *<!--LOGGING-->* <**dependency**>  
 <**groupId**>log4j</**groupId**>  
 <**artifactId**>log4j</**artifactId**>  
 <**version**>1.2.17</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.slf4j</**groupId**>  
 <**artifactId**>slf4j-log4j12</**artifactId**>  
 <**version**>1.7.25</**version**>  
 </**dependency**>  
 *<!--CACHING-->* <**dependency**>  
 <**groupId**>org.ehcache</**groupId**>  
 <**artifactId**>ehcache</**artifactId**>  
 <**version**>3.7.0</**version**>  
 </**dependency**>

Configuration cfg = **new** Configuration()  
 .addAnnotatedClass(Address.**class**)  
 .addAnnotatedClass(Company.**class**)  
 .addAnnotatedClass(Employee.**class**)  
 .addAnnotatedClass(Contact.**class**);  
SessionFactory sf = cfg.buildSessionFactory();  
Session session = sf.openSession();  
session.getTransaction().begin();  
  
  
session.getTransaction().commit();

CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();

CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);

Root<Employee> employeeRoot = criteriaQuery.from(Employee.class);

criteriaQuery.where(criteriaBuilder.equal(employeeRoot.get("name"), "Jooohn"));

List<Employee> employees = session.createQuery(criteriaQuery).getResultList();

createQuery - SELECT

* Alter table Employee to have name column as unique value
* Create method *deleteEmployeesFromId(N)* to delete all Employees and their addresses where Employee id >= N.
* Get first company by name from database. If there is none - create one with an address.
* Create method *createEmployeeInCompany(company)* which creates an Employee with address, randomly generated string (letters a-z lower case) as a name and saves it to database.
* Populate your database with 100.000 Employee entries and count how long it took
* Create a function to find an Employee with name similar to a given value.
* Implement 2nd level caching for Employee and Address
* Enable caching and determine how much faster it is retrieve any Employee for second time.
  + Print out first query time and compare it to second.

*Optional:*Concurrent retrieval of Employee.

Funkcija ikelenti duomenis i json

select \* from fn\_get\_population\_by('Abuja');

CREATE FUNCTION public.fn\_get\_population\_by(IN prm\_capital\_name character varying)

RETURNS json

AS

$BODY$

BEGIN

RETURN (

WITH \_city AS (

SELECT id, name AS capitalName,

countrycode AS capitalCode

FROM city

WHERE name = '' || prm\_capital\_name || ''),

\_data AS (

SELECT capitalName,

capitalCode,

c.population,

c2.language,

c2.percentage

FROM country c

JOIN \_city a on a.id = c.capital

JOIN countrylanguage c2 on c.code = c2.countrycode

WHERE TRUE)

SELECT json\_agg(r.\*) as json\_data

FROM \_data AS r);

END;

$BODY$

LANGUAGE 'plpgsql';