**Реализация демо проекта на Spring Boot для отработки механизмов работы с реляционной базой данных на примере PostgreSQL**

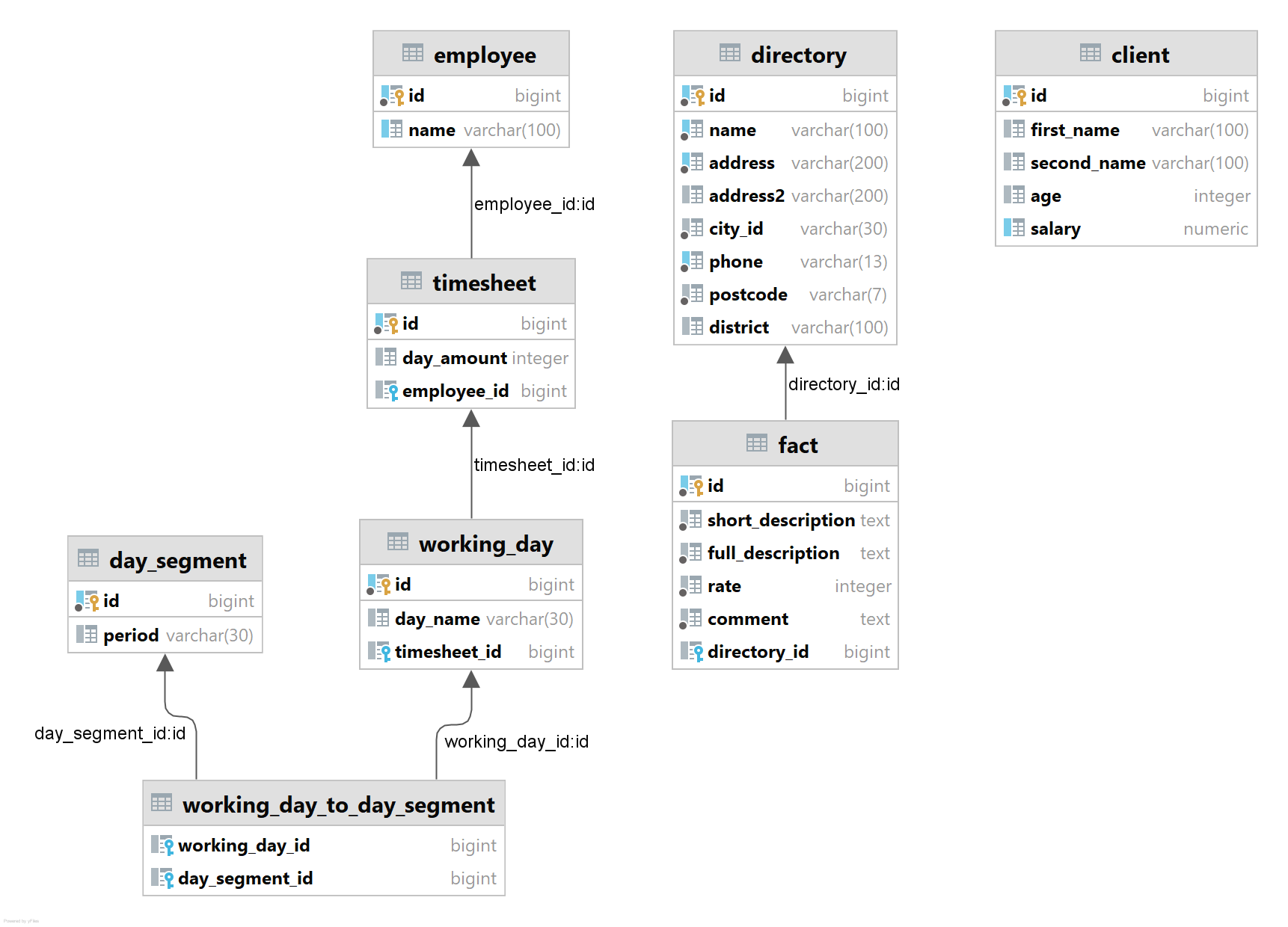
Требования

1. Для работы с базой данных использовать JPA + Hibernate.
2. Для отработки сценариев использовать Unit-tests.

Таблица 1 – Доменные модели, соответствующие им таблицы и типы связей между таблицами

|  |  |  |  |
| --- | --- | --- | --- |
| **Типы связей** | **Доменная модель** | | **Таблица** |
| One-to-one | Работник | | employee |
| Табель | One-to-many | timesheet |
| Many-to-many | Рабочий день | working\_day |
| Промежуток/сегмент дня | | day\_segment |
| One-to-many | Справочник | | directory |
| Факт | | fact |
| One-to-one | Студент | | student |
| Контактная информация | | contact\_info |
| – | Клиент | | client |

**ERD схема базы данных:**



Задачи

1. Реализовать DML операций загрузки, отображения, изменения и удаления (CRUD):
2. для одиночной таблицы без связей:

ClientRepositoryImplTest\_TaskA1

1. для двух связанных таблиц вида «один к одному»: StudentLazyRepositoryImplTest\_TaskA2
2. для двух связанных таблиц вида «один ко многим»: DirectoryRepositoryImplTest\_TaskA3

Для связанных таблиц сравнить работу в жадном и ленивом режимах:

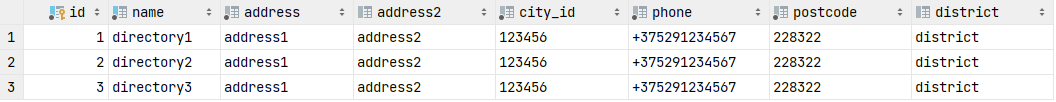
В связи **«один к одному»** выполняется одинаковый запрос в любом из fetch режимов и не выбрасывается LazyInitializationException. Для такого типа связи не важно, какой fetch type выбирать.

В связи **«один ко многим»** работает таким образом:

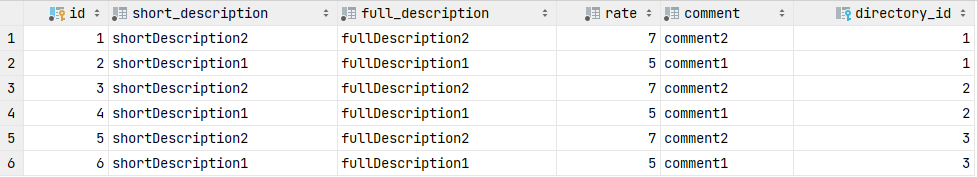
EAGER – выполняется запрос SELECT для получения сущностей Directory. После этого для каждой сущности Directory выполняется SELECT для получения зависимых от неё сущностей Facts.

LAZY – выполняется один запрос с использованием LEFT OUTER JOIN. Применяется опция DISTINCT для того чтобы из получившегося, JOIN-а извлечь уникальные данные. Допустим у нас в БД 3 записи Directory и у каждой из них по 2 зависимые записи в таблице Facts:

*Directory*



*Fact*



С операцией DISTINCT извлечётся 3 уникальных записи сущности Directory, без неё 6. Потому что в таблице Fact содержится две записи, относящиеся к одной сущности Directory, и с учетом особенностей работы LEFT OUTER JOIN, запрос делает выборку по всем записям, связанным с записями в таблице Fact. То есть, если две записи Fact связаны с одной записью Directory, то вернётся две записи Directory.

Можно поймать LazyInitializationException если сессия Hibernate закрыта, но нам надо отобразить в консоль/изменить связные записи в типе связи один ко многим. Для того, чтобы обработать данное исключение, я прочитал статьи:

* <https://vladmihalcea.com/the-best-way-to-handle-the-lazyinitializationexception/>
* <https://www.baeldung.com/hibernate-initialize-proxy-exception>
* <https://stackoverflow.com/questions/22821695/how-to-fix-hibernate-lazyinitializationexception-failed-to-lazily-initialize-a>

Лучшее решение, описанное в них, и на других просторах интернета, это использование операции JOIN FETCH. В примерах ниже описан код, в котором я предотвратил LazyInitializationException используя Criteria API и HQL.

**Пример на сущностях Directory-Fact, метод findById:**

**Запрос в Criteria API (EAGER):**

**try** (Session session = **entityManagerFactory**.unwrap(SessionFactory.**class**).openSession()) {  
 CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();  
  
 CriteriaQuery<DirectoryEager> criteriaQuery = criteriaBuilder.createQuery(DirectoryEager.**class**);  
 Root<DirectoryEager> root = criteriaQuery.from(DirectoryEager.**class**);  
 Predicate idPredicate = criteriaBuilder.equal(root.get(**"id"**), id);  
 criteriaQuery.where(idPredicate);  
  
 TypedQuery<DirectoryEager> query = session.createQuery(criteriaQuery);  
 **return** Optional.*ofNullable*(query.getSingleResult());  
}

**Запрос, сформированный Hibernate (EAGER):**

Hibernate:

select

directorye0\_.id as id1\_3\_,

directorye0\_.address as address2\_3\_,

directorye0\_.address2 as address3\_3\_,

directorye0\_.city\_id as city\_id4\_3\_,

directorye0\_.district as district5\_3\_,

directorye0\_.name as name6\_3\_,

directorye0\_.phone as phone7\_3\_,

directorye0\_.postcode as postcode8\_3\_

from

public.directory directorye0\_

Hibernate:

select

facteagers0\_.directory\_id as director6\_5\_0\_,

facteagers0\_.id as id1\_5\_0\_,

facteagers0\_.id as id1\_5\_1\_,

facteagers0\_.comment as comment2\_5\_1\_,

facteagers0\_.directory\_id as director6\_5\_1\_,

facteagers0\_.full\_description as full\_des3\_5\_1\_,

facteagers0\_.rate as rate4\_5\_1\_,

facteagers0\_.short\_description as short\_de5\_5\_1\_

from

public.fact facteagers0\_

where

facteagers0\_.directory\_id=?

Hibernate:

select

facteagers0\_.directory\_id as director6\_5\_0\_,

facteagers0\_.id as id1\_5\_0\_,

facteagers0\_.id as id1\_5\_1\_,

facteagers0\_.comment as comment2\_5\_1\_,

facteagers0\_.directory\_id as director6\_5\_1\_,

facteagers0\_.full\_description as full\_des3\_5\_1\_,

facteagers0\_.rate as rate4\_5\_1\_,

facteagers0\_.short\_description as short\_de5\_5\_1\_

from

public.fact facteagers0\_

where

facteagers0\_.directory\_id=?

Hibernate:

select

facteagers0\_.directory\_id as director6\_5\_0\_,

facteagers0\_.id as id1\_5\_0\_,

facteagers0\_.id as id1\_5\_1\_,

facteagers0\_.comment as comment2\_5\_1\_,

facteagers0\_.directory\_id as director6\_5\_1\_,

facteagers0\_.full\_description as full\_des3\_5\_1\_,

facteagers0\_.rate as rate4\_5\_1\_,

facteagers0\_.short\_description as short\_de5\_5\_1\_

from

public.fact facteagers0\_

where

facteagers0\_.directory\_id=?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Запрос в Criteria API (LAZY):**

**try** (Session session = **entityManagerFactory**.unwrap(SessionFactory.**class**).openSession()) {  
 CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();  
 CriteriaQuery<DirectoryLazy> criteriaQuery = criteriaBuilder.createQuery(DirectoryLazy.**class**);  
  
 Root<DirectoryLazy> root = criteriaQuery.from(DirectoryLazy.**class**);  
 root.fetch(**"factLazies"**,JoinType.***LEFT***);  
 criteriaQuery.select(root).distinct(**true**);  
 Predicate idPredicate = criteriaBuilder.equal(root.get(**"id"**), id);  
 criteriaQuery.where(idPredicate);  
  
 Query<DirectoryLazy> query = session.createQuery(criteriaQuery);  
 **return** Optional.*ofNullable*(query.getSingleResult());  
}

**Запрос в HQL (LAZY):**

Query<DirectoryLazy> query = session.createQuery(  
 **"SELECT DISTINCT dl FROM DirectoryLazy dl LEFT JOIN FETCH dl.factLazies WHERE dl.id=:id"**,  
 DirectoryLazy.**class**)  
 .setParameter(**"id"**, id);

**Запрос сформированный Hibernate (LAZY):**

Hibernate:

select

distinct directoryl0\_.id as id1\_3\_0\_,

factlazies1\_.id as id1\_5\_1\_,

directoryl0\_.address as address2\_3\_0\_,

directoryl0\_.address2 as address3\_3\_0\_,

directoryl0\_.city\_id as city\_id4\_3\_0\_,

directoryl0\_.district as district5\_3\_0\_,

directoryl0\_.name as name6\_3\_0\_,

directoryl0\_.phone as phone7\_3\_0\_,

directoryl0\_.postcode as postcode8\_3\_0\_,

factlazies1\_.comment as comment2\_5\_1\_,

factlazies1\_.directory\_id as director6\_5\_1\_,

factlazies1\_.full\_description as full\_des3\_5\_1\_,

factlazies1\_.rate as rate4\_5\_1\_,

factlazies1\_.short\_description as short\_de5\_5\_1\_,

factlazies1\_.directory\_id as director6\_5\_0\_\_,

factlazies1\_.id as id1\_5\_0\_\_

from

public.directory directoryl0\_

left outer join

public.fact factlazies1\_

on directoryl0\_.id=factlazies1\_.directory\_id

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Реализовать:
2. получение набора данных *из больше чем двух связанных между собой таблиц* (как пример: Табель, День табеля, Сегмент дня табеля)
3. получение данных для набора таблиц *с разными типами связей между ними*

Реализация подпункта 1 и 2 находятся в классе EmployeeRepositoryImplTest\_TaskB, методы findById и findAll. Модель Employee, Timesheet и связанные с ними таблицы содержат разные типы связей (см. таблицу 1) и количество связанных таблиц при этом – 4.

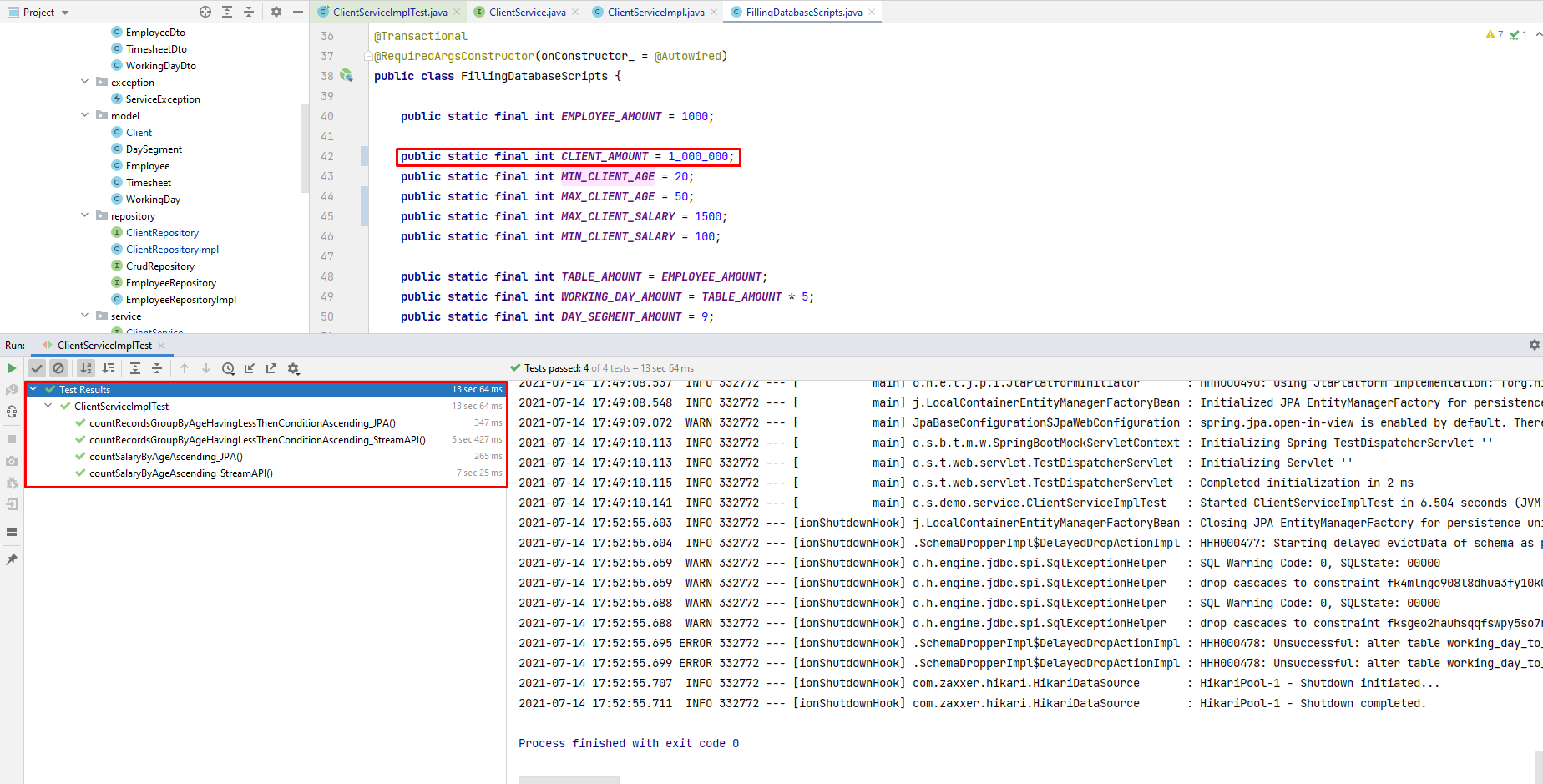
1. получение *сгруппированных данных*. Сравнить удобство и производительность с получением полного набора данных и группировкой уже на уровне java при помощи Stream API.

**Сравнение удобства:**

Случай с **countSalaryByAgeAscending** оказался проще в Hibernate, т.к. в Stream API пришлось долго думать над логикой и разворачивать большую вложенность структур данных в более простую, а затем в нужную мне – List<Client>. Нужно было оперировать разными структурами данных (да и просто дольше гуглил как реализовать задачу). При использовании Hibernate запрос получился визуально и интуитивно более понятным и не выглядит перегруженным логикой, его проще прочитать и понять.

В случае с **countRecordsGroupByAgeHavingLessThenConditionAscending** получилось наоборот: в Hibernate необходимо было использовать интерфейс Tuple (Кортеж), чтобы подсчитать количество записей с группировкой по параметру age. Я долго искал как это можно было сделать (доходили мысли до создания новой модели и отдельных запросов, пока не нашёл объект Кортеж. С ним было намного лучше). В Stream API напротив группировка получилось минималистичной и решение нашлось достаточно быстро, он выглядит намного более читабельным и понятным.

**Сравнение производительности:**



Объём данных – 1\_000\_000 строк в таблице Clients

Таблица 2 – Результаты unit-tests

|  |  |  |  |
| --- | --- | --- | --- |
| Метод | JPA | Stream API | Разница в пользу JPA, во сколько раз |
| countRecordsGroupByAgeHavingLessThenConditionAscending | 347ms | 5427ms | **15,6** |
| countSalaryByAgeAscending | 265ms | 7025ms | **26,5** |

1. получение данных из базы *с пагинацией*

Реализация подпункта 4 находятся в классе EmployeeRepositoryImplTest\_TaskB, методы findAllPagination\_startPage0\_pageSize10 и findAllPagination\_startPage10\_pageSize10.

1. Наполнить созданные таблицы тестовыми данными с учетом их предназначение (справочники - 100 записей, таблицы с фактами - 100 000) и протестировать проект на производительность. Обратить внимание на необходимость использования соответствующих индексов.

Для тестирования производительности проекта с использованием индексов и без них был выбран запрос (найти всех клиентов у кого зарплата равна минимальной в таблице и вывести с сортировкой фамилий по убыванию). Объём данных – 3\_000\_000 записей:

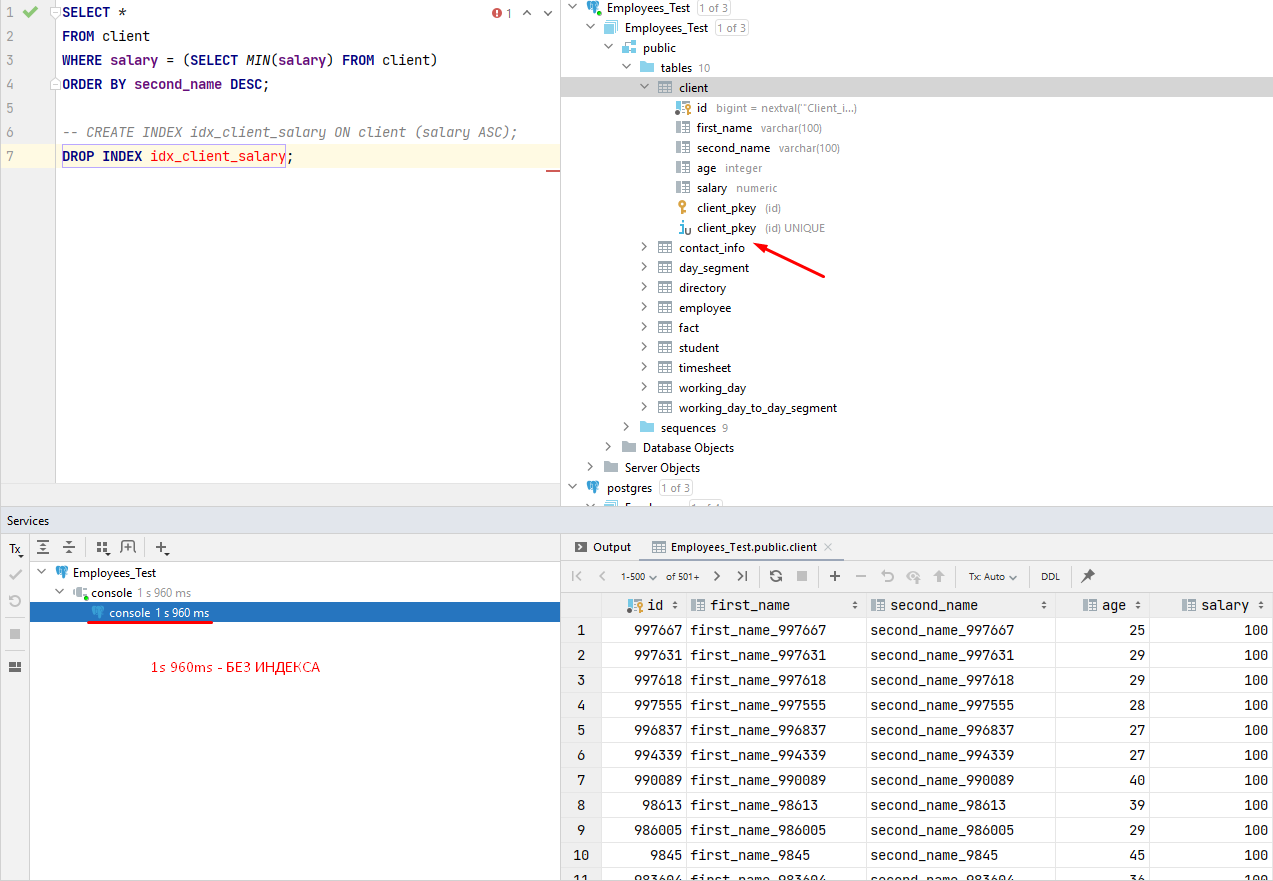
**SELECT** *\****FROM** client  
**WHERE salary** = (**SELECT** *MIN*(**salary**) **FROM** client)  
**ORDER BY second\_name DESC**;

И выбран индекс – зарплата сотрудника по убыванию:

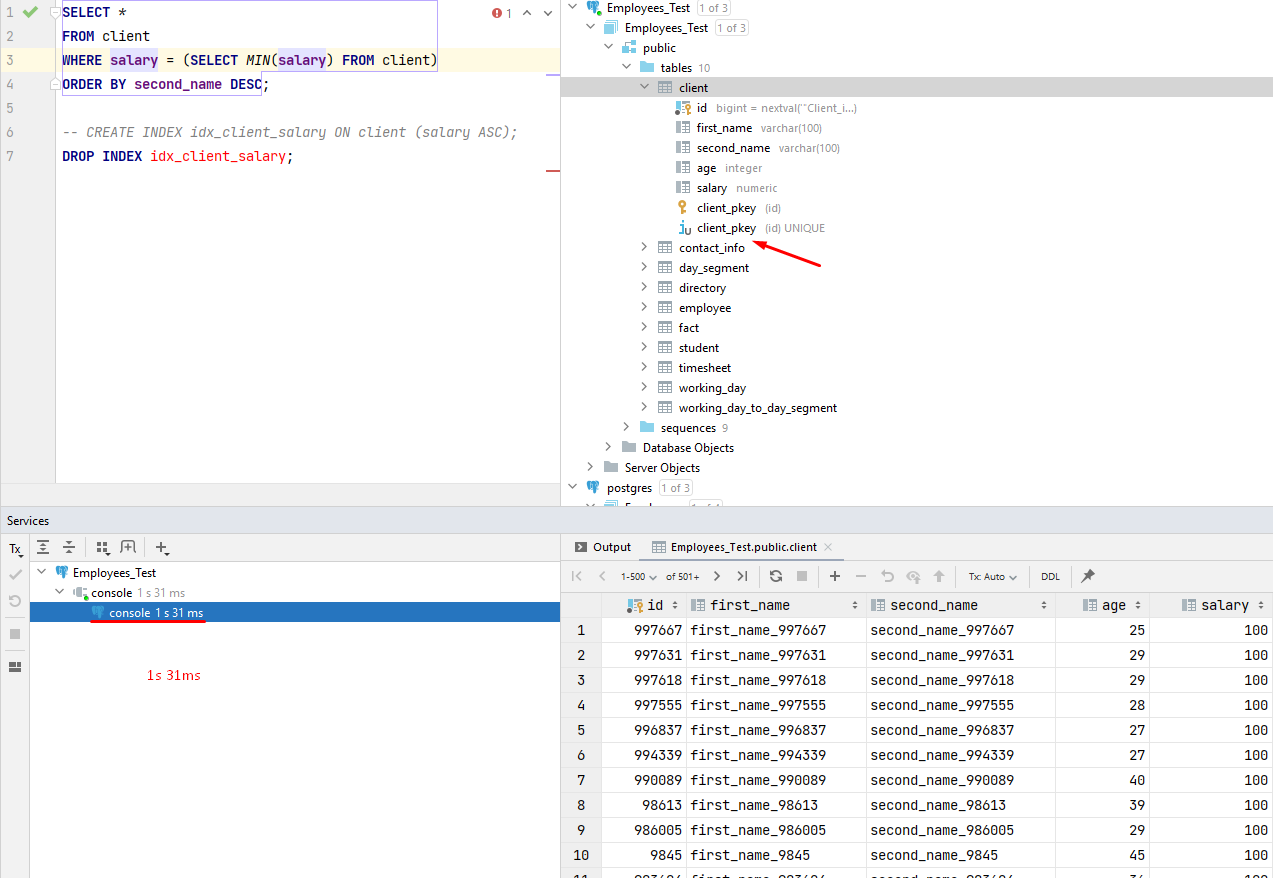
**CREATE INDEX** idx\_client\_salary **ON** client (**salary ASC**);

**Результаты**

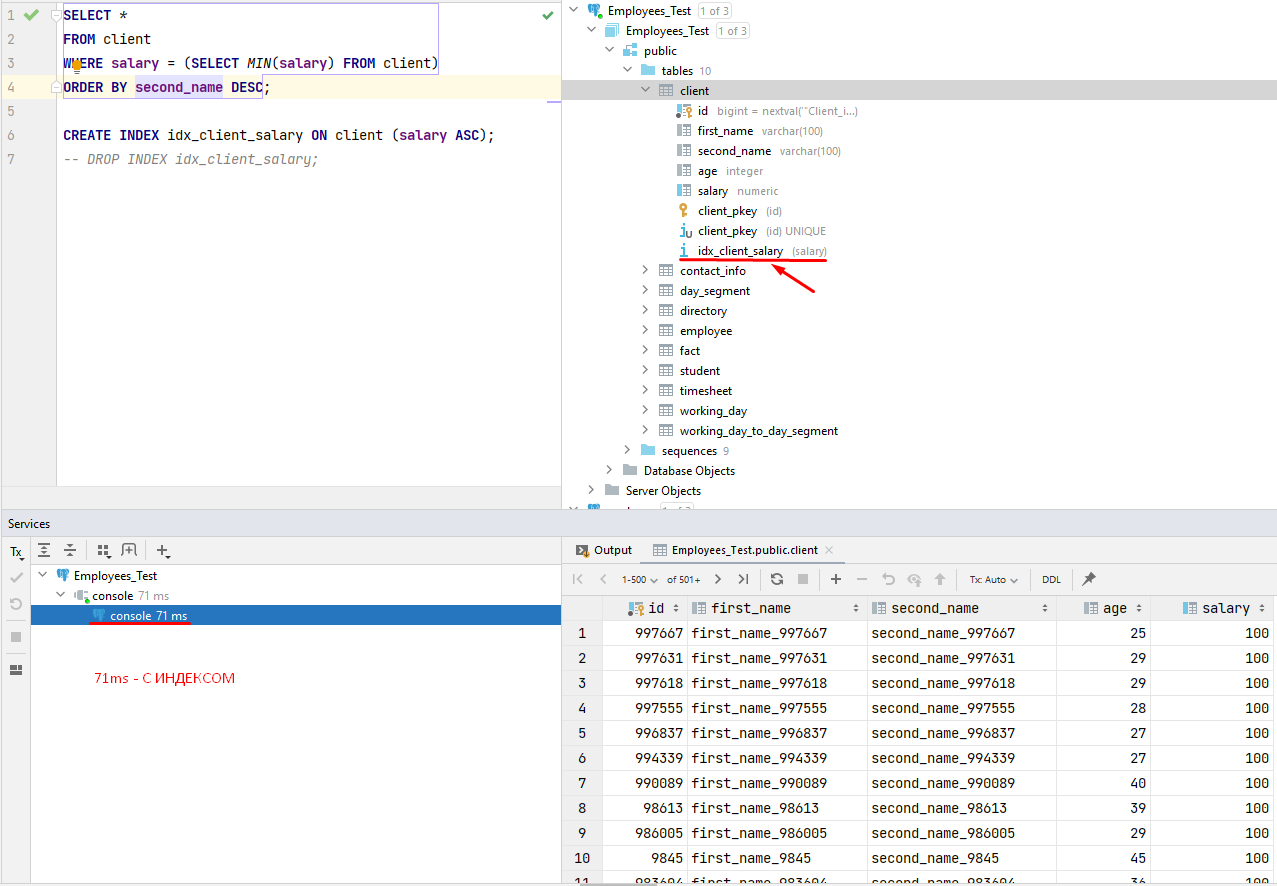
Без индекса:



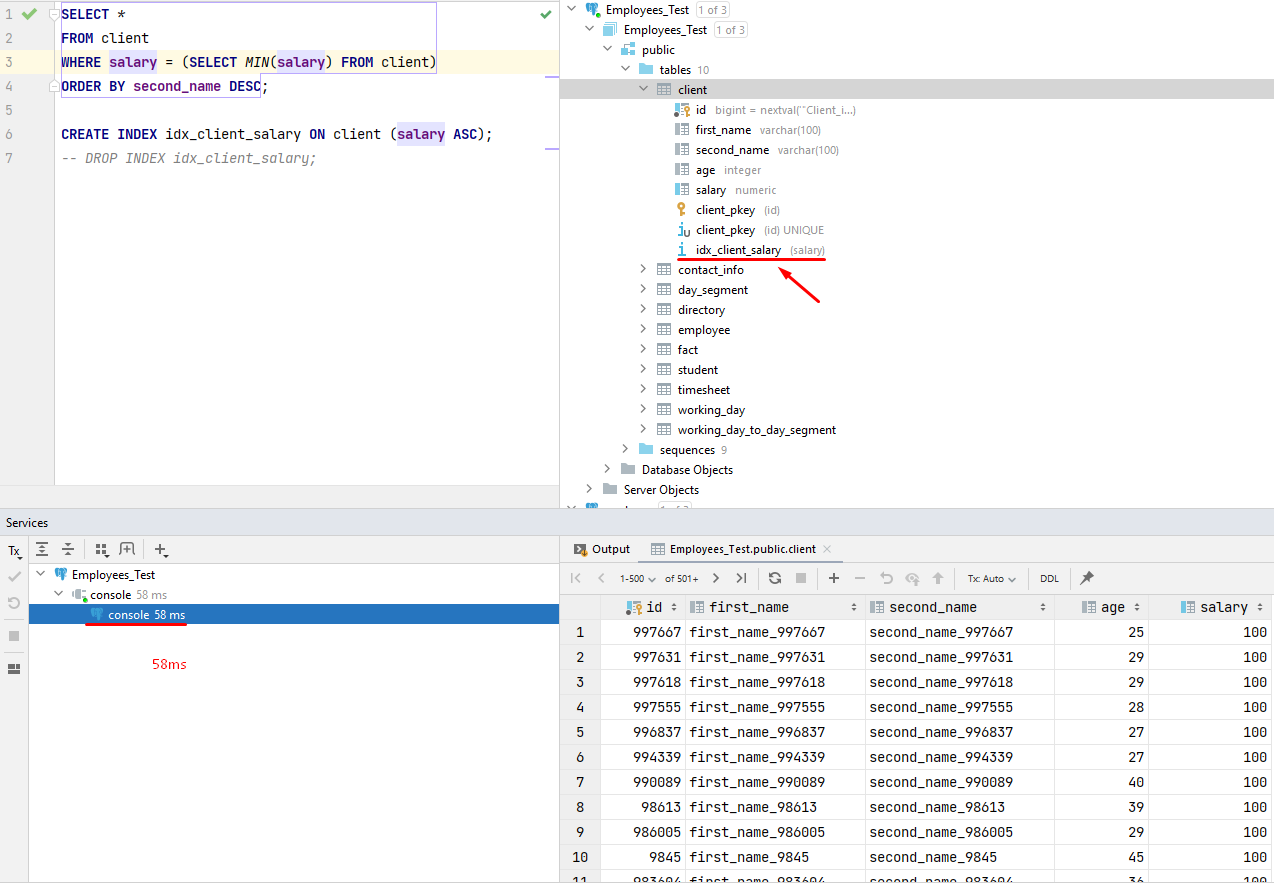
При повторном выполнении:



С индексом:

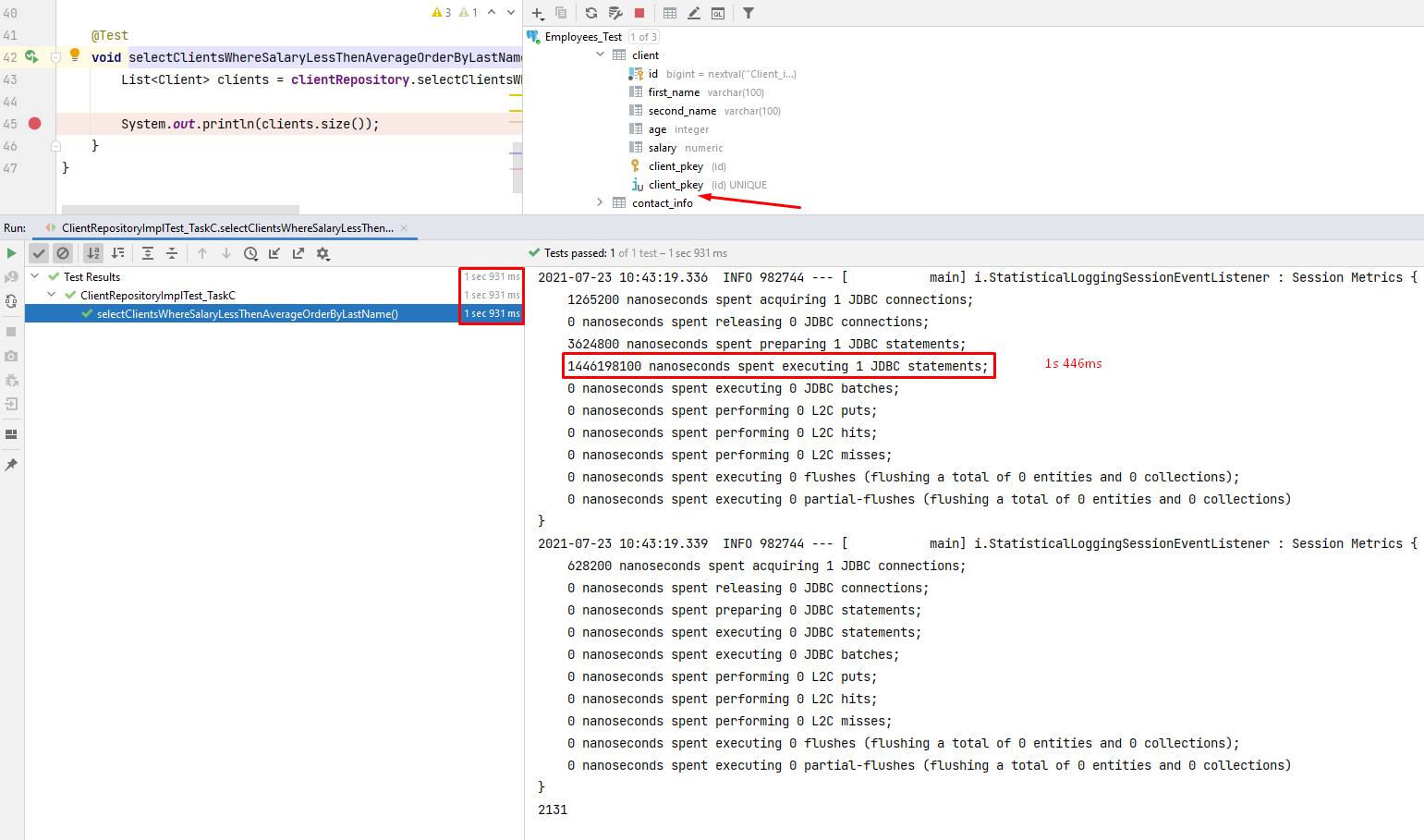


При повторном выполнении:

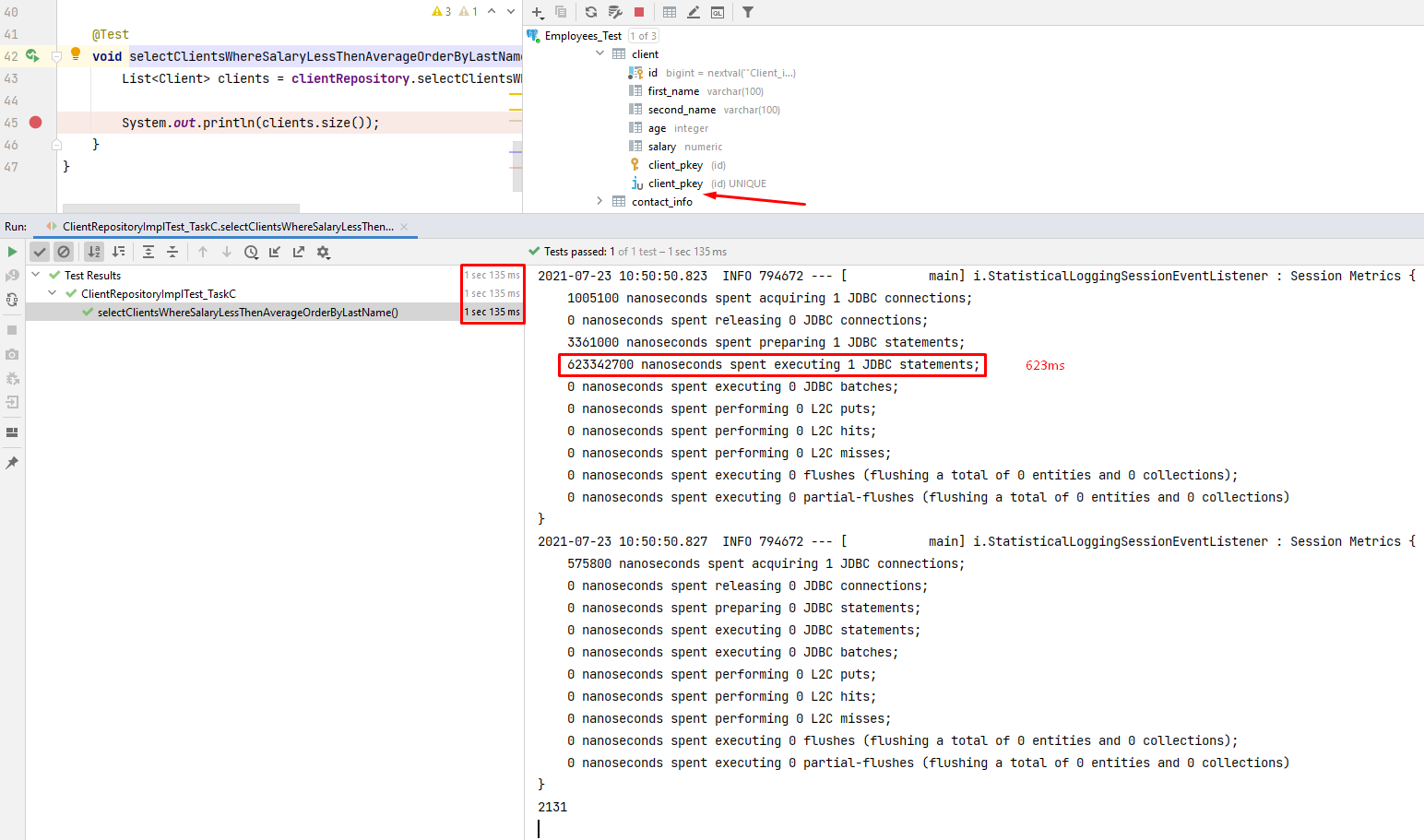


**Hibernate**

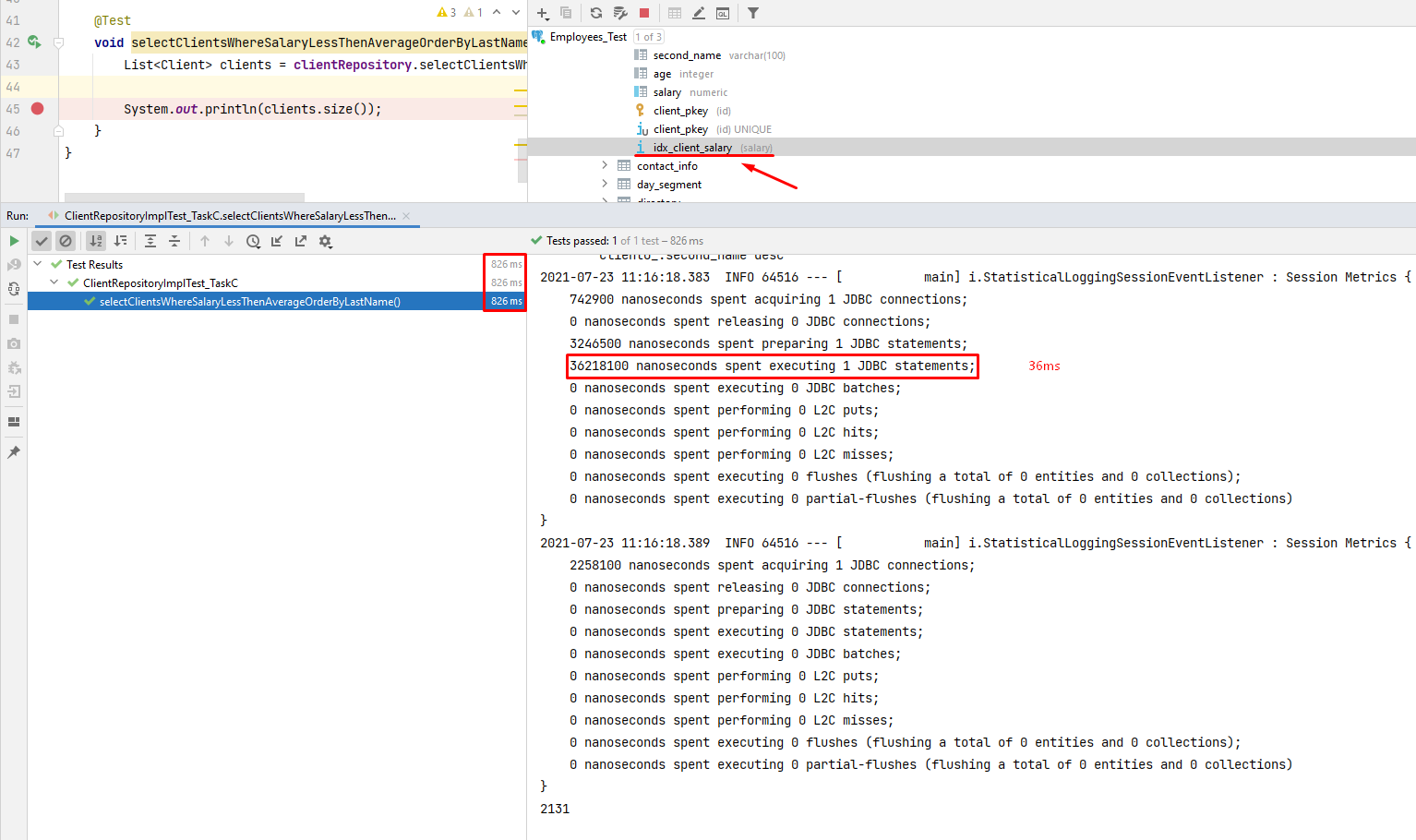
Без индекса:



При повторном выполнении:



С индексом:



При повторном выполнении:

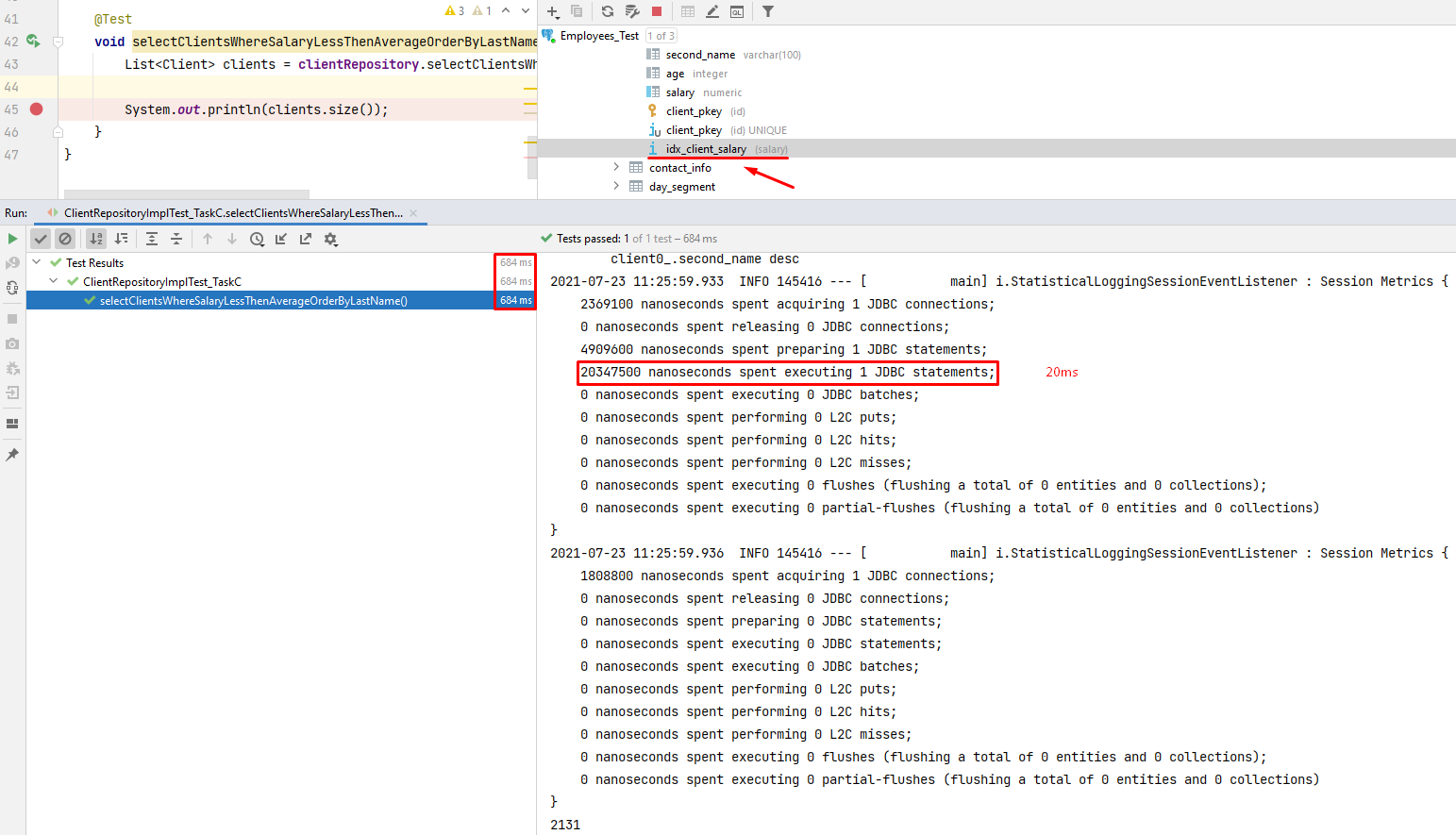


Таблица 3 – Выполнение запроса в Hibernate

|  |  |  |  |
| --- | --- | --- | --- |
| **Номер прогона теста** | **Без индекса** | **С индексом** | **Разница в пользу индекса, во сколько раз** |
| 1 | 1446ms | 36ms | **40,16** |
| 2 | 623ms | 20ms | **30,15** |

Таблица 4 – Выполнение unit-test

|  |  |  |  |
| --- | --- | --- | --- |
| **Номер прогона теста** | **Без индекса** | **С индексом** | **Разница в пользу индекса, во сколько раз** |
| 1 | 1931ms | 826ms | **2,34** |
| 2 | 1135ms | 684ms | **1,66** |

1. Проработать вопрос информационной безопасности данного проекта: какие существуют угрозы и как в проекте они сводятся к допустимому уровню.

Угрозы в проекте:

* Креды подключения к БД хранятся в application.properties. если кто-то получит доступ к файлу, то получит доступ к БД и может её испортить.

Решение – добавить файл в .gitignore чтобы он не попал в удалённый репозиторий, а также запаролить файл в файловой системе сервера, чтобы доступ к его изменению имел только админ.

* Подключение к БД идёт через супер-админа, а значит пользователь проекта может выполнить DDL операции и нарушить схему БД.

Решение – через супер-админа создать пользователя БД без возможности выполнять DDL операции, а только DML.

* Отсутствие код-ревью -> в коде могу быть баги или забыты проверки на NPE и другие непроверяемые исключения.

Решение – провести код-ревью.

* Скрипты для заполнения БД, написанные на   
  Java, могут содержать логические ошибки и не будут выполнены в БД, если задать неправильные исходные значения (количество клиентов, минимальная/максимальная зарплата клиента).

Решение – следить за исходными значениями и рассчитывать заранее сколько записей и с какими порядковыми номерами надо вставить в БД, проверять чтобы значения не были непротиворечивы (к примеру, минимальный возраст клиента больше максимального).

* В Hibernate можно словить LazyInitializationException.

Решение – писать код используя best practice по предотвращению данного типа исключений.

* Уязвимости в Hibernate: SQL-инъекции (<https://www.cvedetails.com/cve/CVE-2020-25638/>).

Решение – проводить код-ревью, писать код внимательнее, покрывать код интеграционными тестами.