PISA UNIVERSITY

TASK 1
LARGE-SCALE AND MULTI-STRUCTURED DATABASES

KEY VALUE IMPLEMENTATION REPORT

ACADEMIC YEAR 2019-2020

STEFANO PETROCCHI, ANDREA TUBAK, FRANCESCO RONCHIERI, ALESSANDRO MADONNA

SUMMARY

INTRODUCTION

This document is meant to be a guide to understand our implementation of a hybrid solution for **Task 1 *PisaFlix***.

It is strongly advised to read the **Feasibility Study** before going forward with this document.
As already discussed in the Feasibility Study the only entity suited for the implementation of a key value solution is the *Comment* entity.

We decided to use ***LevelDB*** to store the information of all the comments made by users on Cinemas and Films.

Much of the software remained untouched, therefore we are going to discuss just the **main differences** with respect to the purely relational solution.

DATABASE

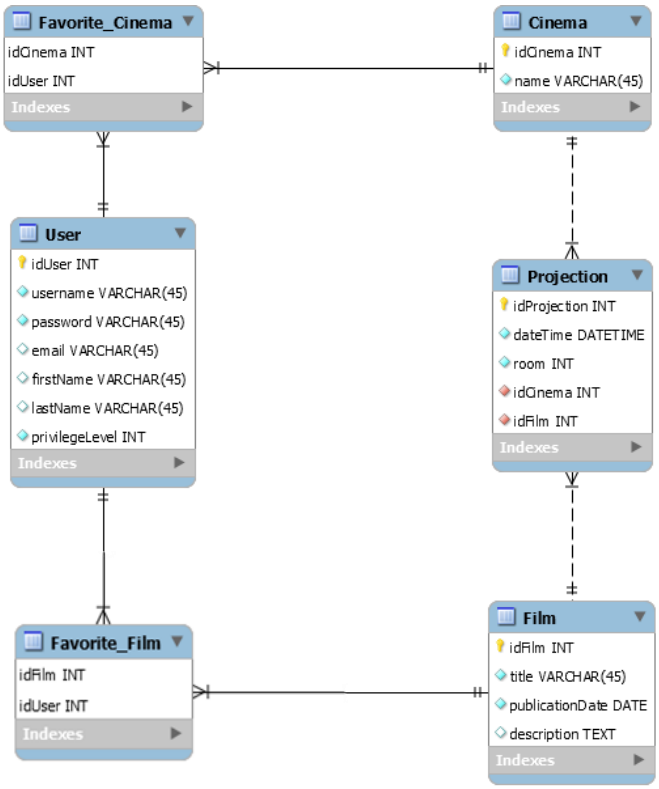
RELATIONAL PART

The relational part of our database differs from the original design just for the absence of 3 tables:

- ***comment***
- ***cinema_has_comment***
- ***film_has_comment***

The data that was stored in *comment* table will be now entirely stored in the *LevelDB* store and the information contained in the other two tables must be reconstructed with ***ad hoc*** java code when a cinema or a film is retrieved.

E-R DIAGRAM



KEY-VALUE PART

This part of our database does not have a defined structure; it is basically just a container where you can store data associated to a string that acts as a *key*. Given this characteristic we have used *LevelDB* to store all the data necessary to replace the relational implementation of the *Comment* entity.

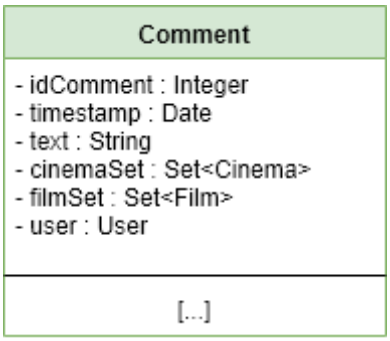
THE DATA STORED

In the first time on which *LevelDB* is opened, a couple of *key-value* pairs are added as part of an **initialization**. The keys are:

- ***settingsPresent***
- ***setting:lastCommentKey***

The *first* one will contain ***true*** if the initialization is **already been done**, the *second* one will contain the **id of the last comment** being inserted, so that whenever a new comment is created, we can retrieve the right id to insert it. Obviously, this id will be **incremented** at each insert.

The next data we are going to discuss is comment itself. For clarity's sake let us remind the **structure** of the *Comment* entity.



Each comment being stored in LevelDB will be divided in all its components, so there will be a *key-value* pair of each field of a comment being stored. The main idea is to use the *key* ***comment:x:field*** for each field of a comment, where **x** is the **id of the comment** to be stored. The keys to each field are:

- **comment:x:timestamp**
- **comment:x:text**
- **comment:x:cinema or comment:x:film**
- **comment:x:user**

As an example, let's say we want to store a new comment that has "*awesome film*" as text. In this case we will get the id of this new comment by retrieving the id of the last comment being inserted (e.g. "*5*"), and then we will save this *key-value* pair: ***comment:6:text***, "*awesome film*".

The last kind of data stored in LevelDB are hand-made **indexes** that store the list of ids of all the comments associated to a given Film or Cinema. To achieve this, whenever a comment is created, it is also **added to the list of ids linked to that Cinema or Film**.

As an example, the *key* to access the list of ids linked to a cinema or a film with id *8* would be ***cinema:8:comments***, for the former, and ***film:8:comments*** for the latter. What we will find is a string of concatenated ids separated by a colon symbol.

This feature has been implemented to make the retrieve operation of all the comments associated to a Film/Cinema more efficient. The heart of the problem is that we would have to go through all the comments stored in the *LevelDB* store just to see which one match with the Film/Cinema we want to visualize. Given the fact that most of the load on this database will be associated to the visualization of comments, the hassle of implementing these indexes is easily justified.

SOFTWARE IMPLEMENTATION

The main differences in this area are:

- Different **annotations** in the entities involved with comments
- A new super class to manage operations on LevelDB called ***KeyValueDBManager***
- CommentManager completely replaced by ***CommentManagerKV***
- A slightly different implementation of all database managers having to deal with *CommentManagerKV*

ENTITIES ANNOTATIONS DIFFERENCES

Given the fact that the ***Comment*** entity is completely out from our relational database, this entity will not have any kind of directives for *Hibernate*.

```
1.  // file Comment.java
2.  public class Comment implements Serializable {
3.      private static final long serialVersionUID = 1L;
4.      private Integer idComment;
5.      private Date timestamp;
6.      private String text;
7.      private Set<Cinema> cinemaSet = new LinkedHashSet<>();
8.      private Set<Film> filmSet = new LinkedHashSet<>();
9.      private User user;
10.
11.     // Getters and Setters
12. }
```

Other entities like **Cinema** and **Film** have a `Set<Comment>` which contains the set of comments associated to them. In this case we just replaced all the directives with `@Transient`, which tells *Hibernate* to **ignore** that field. We will have to manage those fields in the ***EntityDBManager*** associated to each entity.

```
1.  // file Film.java
2.  @Entity
3.  @Table(name = "Film")
4.  public class Film implements Serializable {
5.      private static final long serialVersionUID = 1L;
6.
7.      @Id
8.      @GeneratedValue(strategy = GenerationType.IDENTITY)
9.      @Basic(optional = false)
10.     @Column(name = "idFilm")
11.     private Integer idFilm;
12.
13.     @Basic(optional = false)
14.     @Column(name = "title")
```

```
15.     private String title;
16.
17.     @Basic(optional = false)
18.     @Column(name = "publicationDate")
19.     @Temporal(TemporalType.DATE)
20.     private Date publicationDate;
21.
22.     @Lob
23.     @Column(name = "description")
24.     private String description;
25.
26.     @JoinTable(name = "Favorite_Film", joinColumns = {
                    @JoinColumn(name = "idFilm", referencedColumnName = "idFilm")},
                    inverseJoinColumns = {
                    @JoinColumn(name = "idUser", referencedColumnName = "idUser")})
27.     @ManyToMany(fetch = FetchType.EAGER)
28.     private Set<User> userSet = new LinkedHashSet<>();
29.
30.     @Transient // |====||
31.     private Set<Comment> commentSet = new LinkedHashSet<>();
32.
33.     @OneToMany(mappedBy = "idFilm", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
34.     private Set<Projection> projectionSet = new LinkedHashSet<>();
35.
36. // List of methods not shown…
37. }
```

KEY VALUE DB MANAGER

KeyValueDBManager is a super class which regulates all the basic operations executed on our LevelDB store.

```
1.  public class KeyValueDBManager {
2.
3.      protected static DB KeyValueDB;
4.      private static final Options options = new Options();
5.
6.      DateFormat dateFormat = new SimpleDateFormat("dd:MM:yyyy HH.mm.ss");
7.
8.      public static DB getKVFactory(){
9.
10.         if(KeyValueDB == null){
11.             start();
12.         }
13.         return KeyValueDB;
14.     }
15.
16.     public static void start() {
17.         try {
18.             KeyValueDB = factory.open(new File("KeyValueDB"), options);
19.         } catch (IOException ex) {
20.             System.out.println("Errore non si è aperto il keyValueDB");
21.             Logger.getLogger(KeyValueDBManager.class.getName()).log(Level.SEVERE, null, ex
    );
22.         }
23.     }
24.
25.     public static void stop() {
26.             try {
27.                 KeyValueDB.close();
28.             } catch (IOException ex) {
29.                 Logger.getLogger(KeyValueDBManager.class.getName()).log(Level.SEVERE, null
    , ex);
30.             }
31.         }
32.
33.     protected void settings() {
34.         String value = get("settingsPresents");
35.         if (value == null || "false".equals(value)) {
```

```
            put("settingsPresent", "true");
            put("setting:lastCommentKey", "0");
        }
    }

    protected void put(String key, String value) {
        getKVFactory().put(bytes(key), bytes(value));
    }

    protected void delete(String key) {
        getKVFactory().delete(bytes(key));
    }

    protected String get(String key) {
        byte[] value = getKVFactory().get(bytes(key));
        if (value != null) {
            return Iq80DBFactory.asString(value);
        } else {
            //System.out.println("Key not found");
            return null;
        }
    }

}
```

Every *EntityDBManager* that wants to use this operations has to extend ***KeyValueDBManager*** and to call `super.settings()` inside its constructor (this is done to ensure that the LevelDB store is properly opened and initialized).

An example of this is shown below:

```
// file FilmManagerKV.java
public class FilmManagerKV extends KeyValueDBManager
implements FilmManagerDatabaseInterface {

    private final EntityManagerFactory factory;
    private EntityManager entityManager;

    private static FilmManagerKV filmManager;

    public static FilmManagerKV getIstance() {
        if (filmManager == null) {
            filmManager = new FilmManagerKV();
        }
        return filmManager;
    }

    private FilmManagerKV() {
        factory = DBManager.getEntityManagerFactory();
        super.settings();
    }

// ALL OTHER METHODS NOT REPORTED…

}
```

Final note on the utilization of this class: while the **opening** of the *LevelDB* store is done automatically, the **closing** is not. Always remember to call *KeyValueDBManager.close()* at the end of the main method.

COMMENT MANAGER KV

CommentManagerKV contains all the code needed to manage comments in our application using just the *LevelDB* store. It implements all the methods in the ***CommentManagerDatabaseInterface***

in order to have a smooth transition from the relational implementation to this one. It also has some auxiliary methods to manage *indexes*, set of *comments retrieval* and *differentiation of the type* of comment (because given an id we don't know if that comment is associated to a Film or a Cinema).

For the sake of brevity, the code is not reported, the reader can consult it directly in our java project folder.

ENTITY DB MANAGER DIFFERENCES

Given the fact that we added `@Transient` on all the fields of the classes that have some kind of relationship with *Comment*, we will not get a complete *Cinema* or *Film* whenever we execute an operation like `getById()` the set of comments associated with a *Cinema*, for instance, will remain **empty**. Therefore, we had to add a **call for the retrieval** of those comments in all the methods which return entities related to *Comment*.

This is, for example, `getById()` in ***CinemaManager***:

```
1.  //file CinemaManagerKV.java
2.  @Override
3.  public Cinema getById(int cinemaId, boolean retreiveComments) {
4.      Cinema cinema = null;
5.      try {
6.          entityManager = factory.createEntityManager();
7.          entityManager.getTransaction().begin();
8.          cinema = entityManager.find(Cinema.class, cinemaId);
9.
10.
11.         if(retreiveComments){
12.             cinema.setCommentSet(CommentManagerKV.getIstance().getCommentsCinema(cinema.ge
    tIdCinema()));
13.         }
14.     } catch (Exception ex) {
15.         System.out.println(ex.getMessage());
16.         ex.printStackTrace(System.out);
17.         System.out.println("A problem occurred in retriving a film!");
18.     } finally {
19.         if(entityManager.isOpen())
20.             entityManager.close();
21.     }
22.     return cinema;
23. }
```

The only difference from the original code is that, after the retrieval of the cinema object, we check if the *boolean* ***retrieveComments*** is *true* and if that's the case we go on and call the method **`getCommentsCinema()`** and give the result to the comment set of the cinema object. The ***retrieveComments*** *boolean* is needed because, depending on the utilization of the cinema object, *we might not need the comments to be set*; saving us time.