

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 3](#_Toc28352348)

[Database 3](#_Toc28352349)

[Relational Part 3](#_Toc28352350)

[E-R DIAGRAM 4](#_Toc28352351)

[Key-Value Part 4](#_Toc28352352)

[The Data Stored 5](#_Toc28352353)

[Software Implementation 6](#_Toc28352354)

[Entities Annotations differences 6](#_Toc28352355)

[Key Value DB Manager 8](#_Toc28352356)

[Comment Manager KV 10](#_Toc28352357)

[Entity DB Manager Differences 10](#_Toc28352358)

# 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

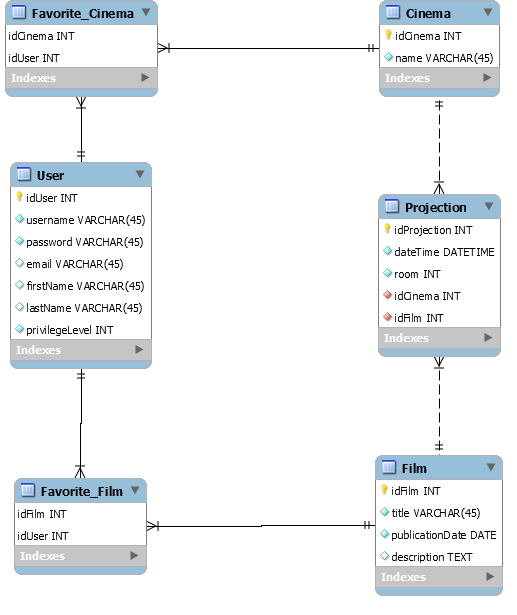
## Relational Part

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

1. comment
2. cinema\_has\_comment
3. film\_has\_comment

The data stored in the comment table will be entirely stored in the LevelDB store, whereas the information contained in the other two tables must be reconstructed with ad hoc java code whenever 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.

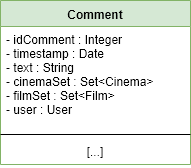
### The Data Stored

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:

1. settingsPresent
2. 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:

1. Different annotations in the entities involved with comments
2. A new super class to manage operations on LevelDB called KeyValueDBManager
3. CommentManager completely replaced by CommentManagerKV
4. A slightly different implementation of all Database managers having to deal with CommentManagerKV

Let’s examine all of them.

## 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.

// file Comment.java

public class Comment implements Serializable {

    private static final long serialVersionUID = 1L;

    private Integer idComment;

    private Date timestamp;

    private String text;

    private Set<Cinema> cinemaSet = new LinkedHashSet<>();

    private Set<Film> filmSet = new LinkedHashSet<>();

    private User user;

// Getters and Setters

}

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.

// file Film.java

@Entity

@Table(name = "Film")

public class Film implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    @Basic(optional = false)

    @Column(name = "idFilm")

    private Integer idFilm;

    @Basic(optional = false)

    @Column(name = "title")

    private String title;

    @Basic(optional = false)

    @Column(name = "publicationDate")

    @Temporal(TemporalType.DATE)

    private Date publicationDate;

    @Lob

    @Column(name = "description")

    private String description;

    @JoinTable(name = "Favorite\_Film", joinColumns = {

        @JoinColumn(name = "idFilm", referencedColumnName = "idFilm")}, inverseJoinColumns = {

        @JoinColumn(name = "idUser", referencedColumnName = "idUser")})

    @ManyToMany(fetch = FetchType.EAGER)

    private Set<User> userSet = new LinkedHashSet<>();

    @Transient // 🡸====||

    private Set<Comment> commentSet = new LinkedHashSet<>();

    @OneToMany(mappedBy = "idFilm", fetch = FetchType.EAGER, cascade = CascadeType.ALL)

    private Set<Projection> projectionSet = new LinkedHashSet<>();

// List of methods not shown…

}

## Key Value DB Manager

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

public class KeyValueDBManager {

    protected static DB KeyValueDB;

    private static final Options options = new Options();

    DateFormat dateFormat = new SimpleDateFormat("dd:MM:yyyy HH.mm.ss");

    public static DB getKVFactory(){

        if(KeyValueDB == null){

            start();

        }

        return KeyValueDB;

    }

    public static void start() {

        try {

            KeyValueDB = factory.open(new File("KeyValueDB"), options);

        } catch (IOException ex) {

            System.out.println("Errore non si è aperto il keyValueDB");

            Logger.getLogger(KeyValueDBManager.class.getName()).log(Level.SEVERE, null, ex);

        }

    }

    public static void stop() {

            try {

                KeyValueDB.close();

            } catch (IOException ex) {

                Logger.getLogger(KeyValueDBManager.class.getName()).log(Level.SEVERE, null, ex);

            }

        }

    protected void settings() {

        String value = get("settingsPresents");

        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:

   //file CinemaManagerKV.java

@Override

    public Cinema getById(int cinemaId, boolean retreiveComments) {

        Cinema cinema = null;

        try {

            entityManager = factory.createEntityManager();

            entityManager.getTransaction().begin();

            cinema = entityManager.find(Cinema.class, cinemaId);

            if(retreiveComments){

                cinema.setCommentSet(CommentManagerKV.getIstance().getCommentsCinema(cinema.getIdCinema()));

            }

        } catch (Exception ex) {

            System.out.println(ex.getMessage());

            ex.printStackTrace(System.out);

            System.out.println("A problem occurred in retriving a film!");

        } finally {

            if(entityManager.isOpen())

                entityManager.close();

        }

        return cinema;

    }

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.