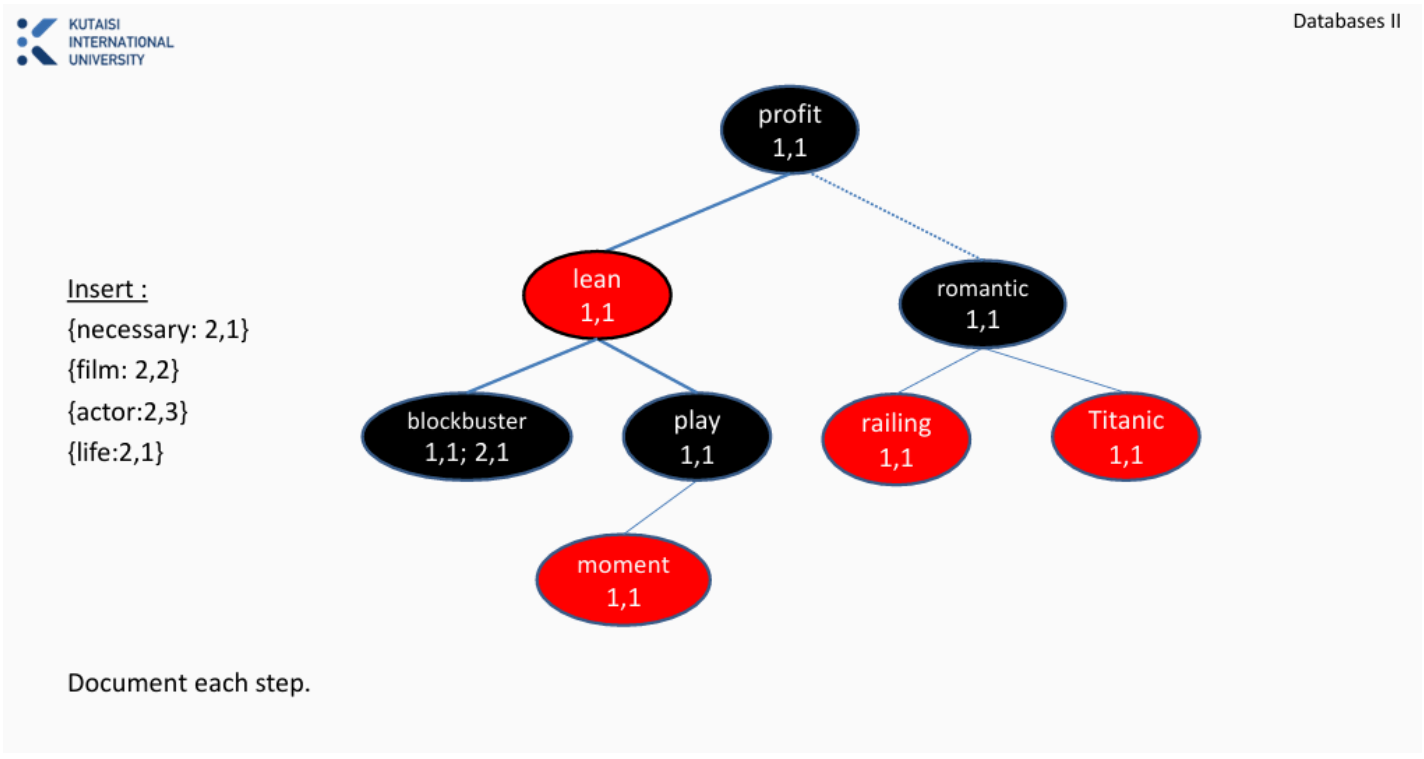
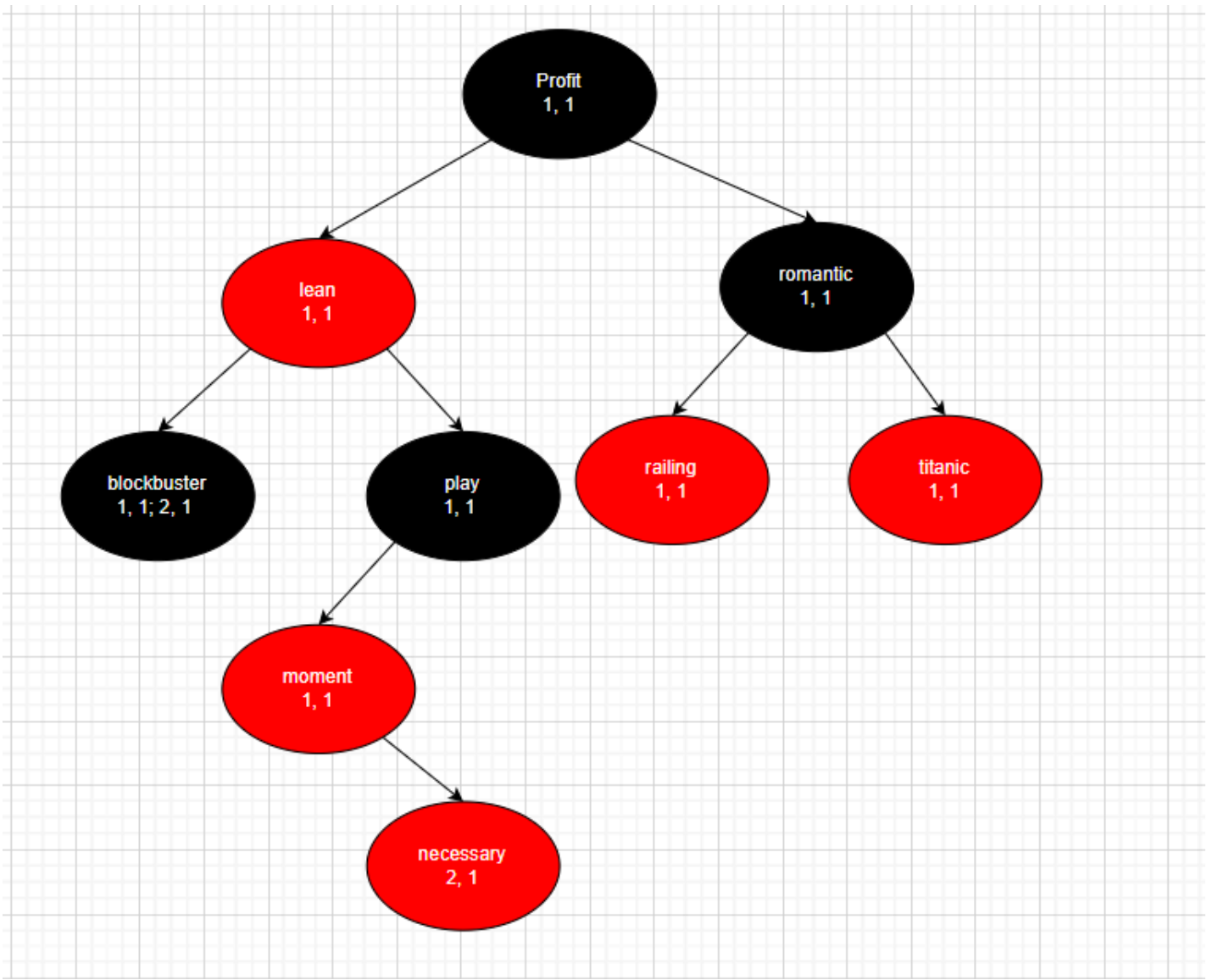
A8 + A9

Type	Homework



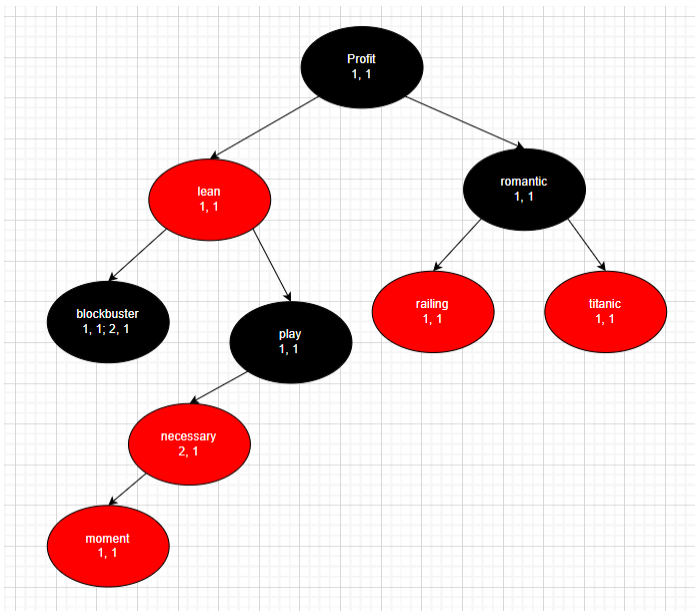
Step 1:

add {necessary: 2,1}.



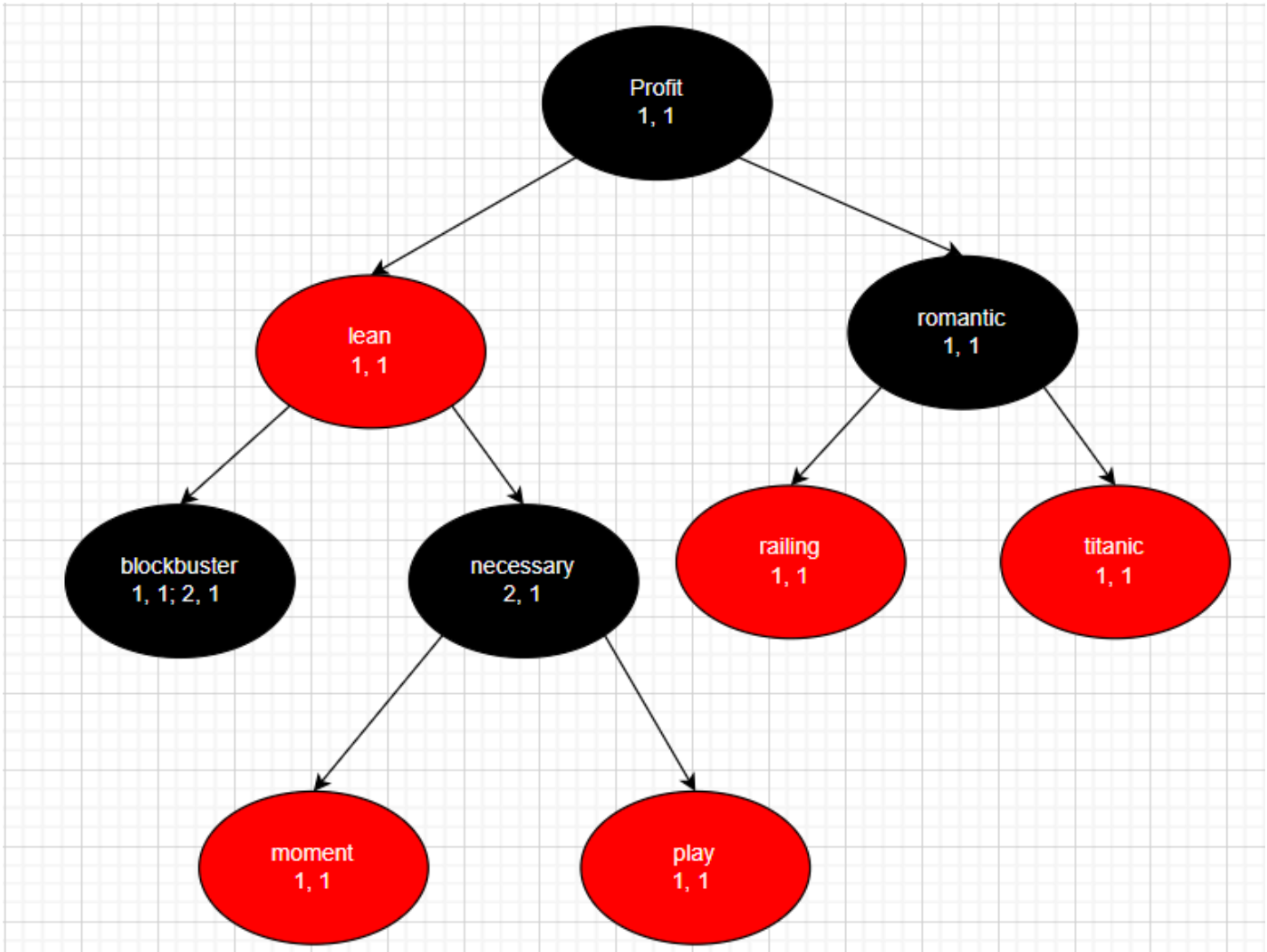
Step 2:

necessary has a red parent and black uncle(null), it is a LR (triangle) configuration so we rotate parent (moment) left (in the opposite direction of uncle)



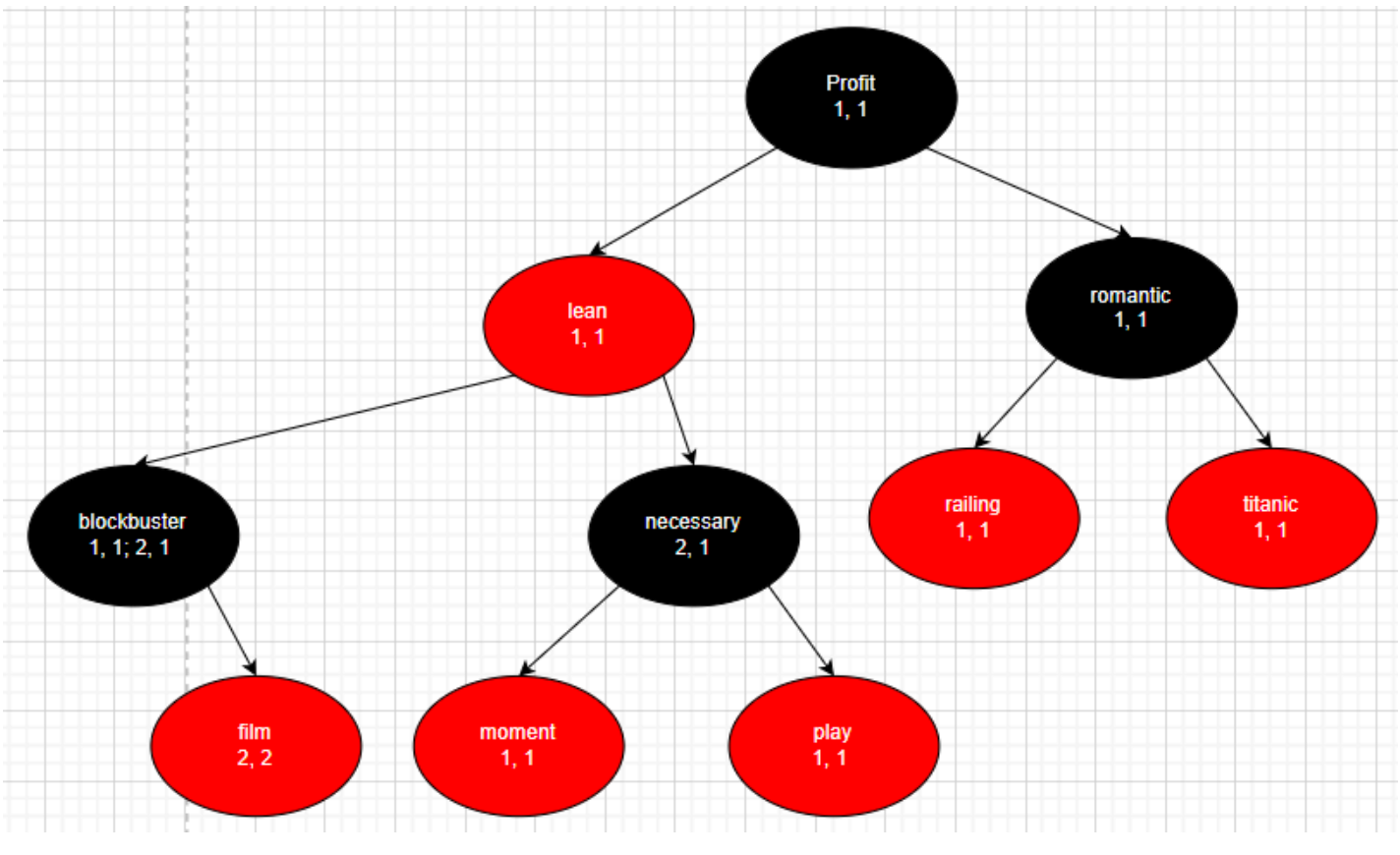
Step 3:

now it is a LL (line) configuration, so rotate gp right (in the opposite direction of x), recolor: former p (now root of subtree): black, former gp (now right child: red).



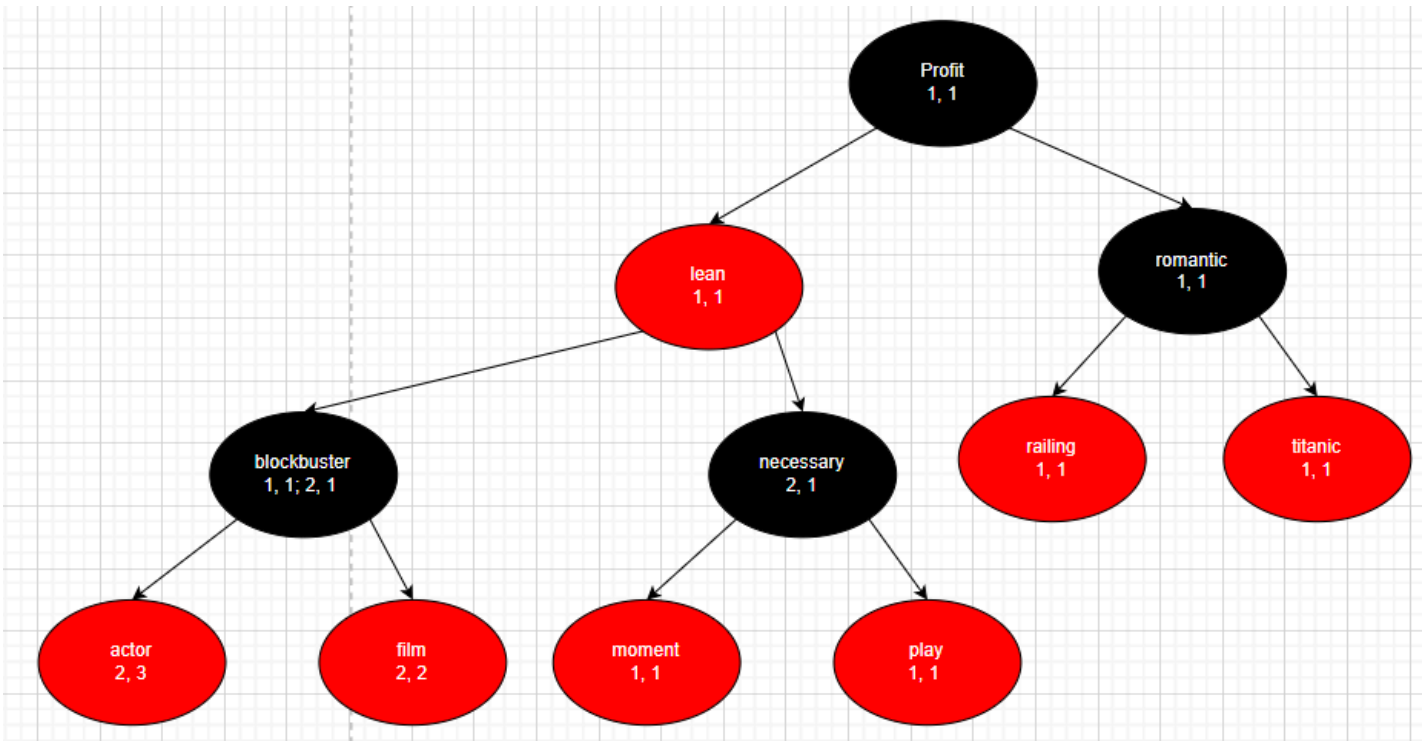
Step 4:

so now add {film: 2,2}.



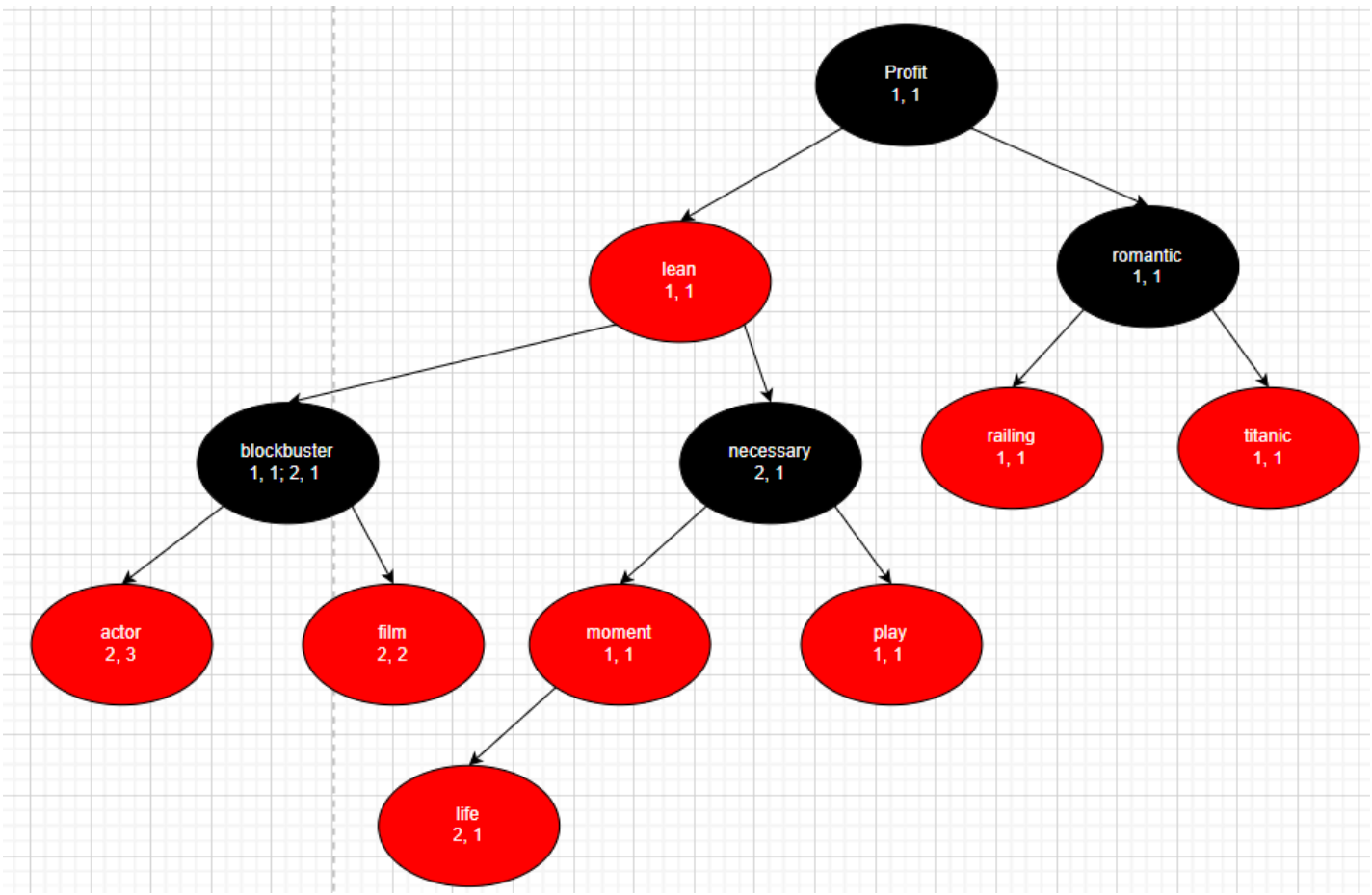
Step 5:

now add {actor:2,3}.



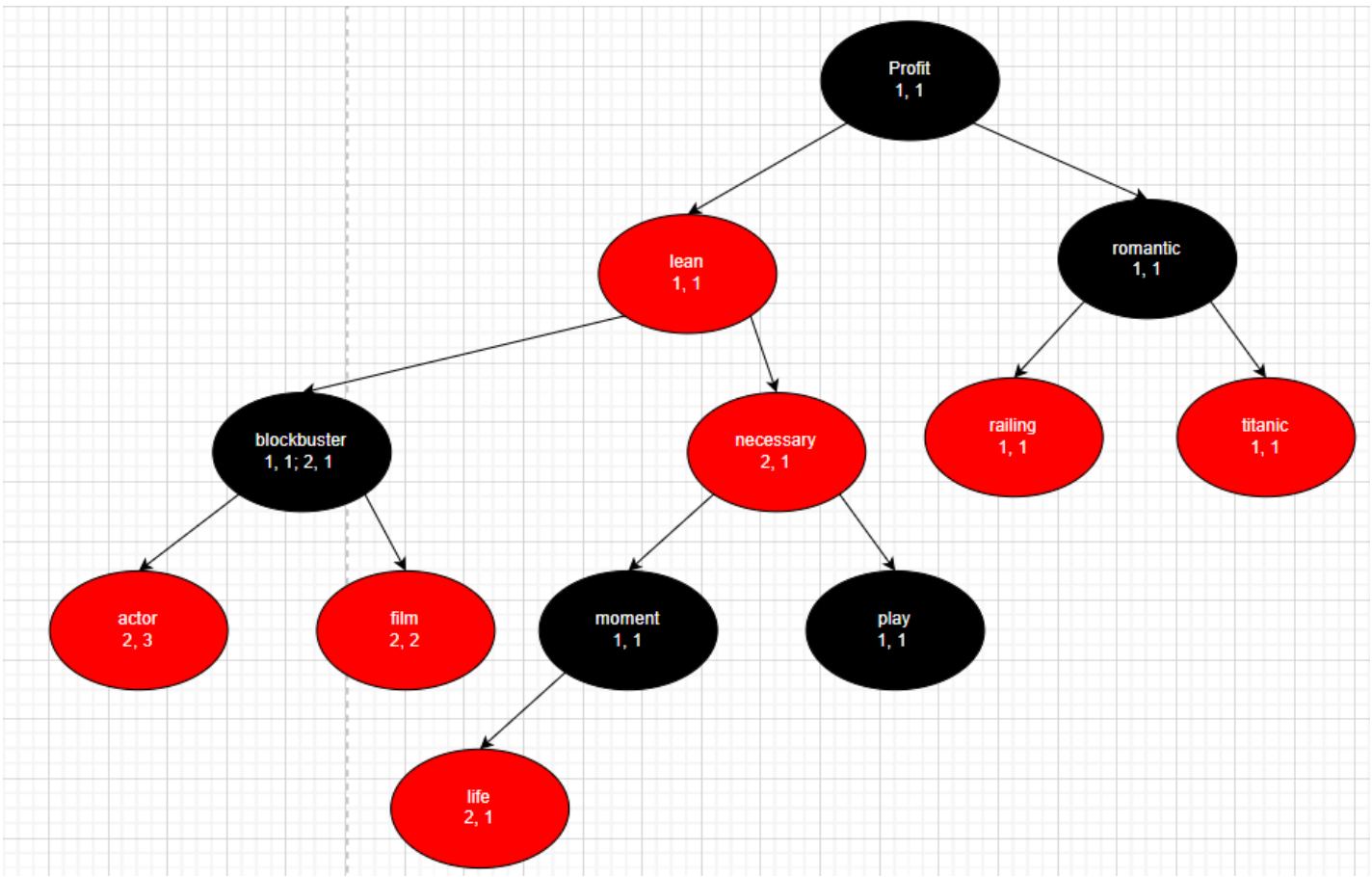
Step 6:

add {life:2,1}.



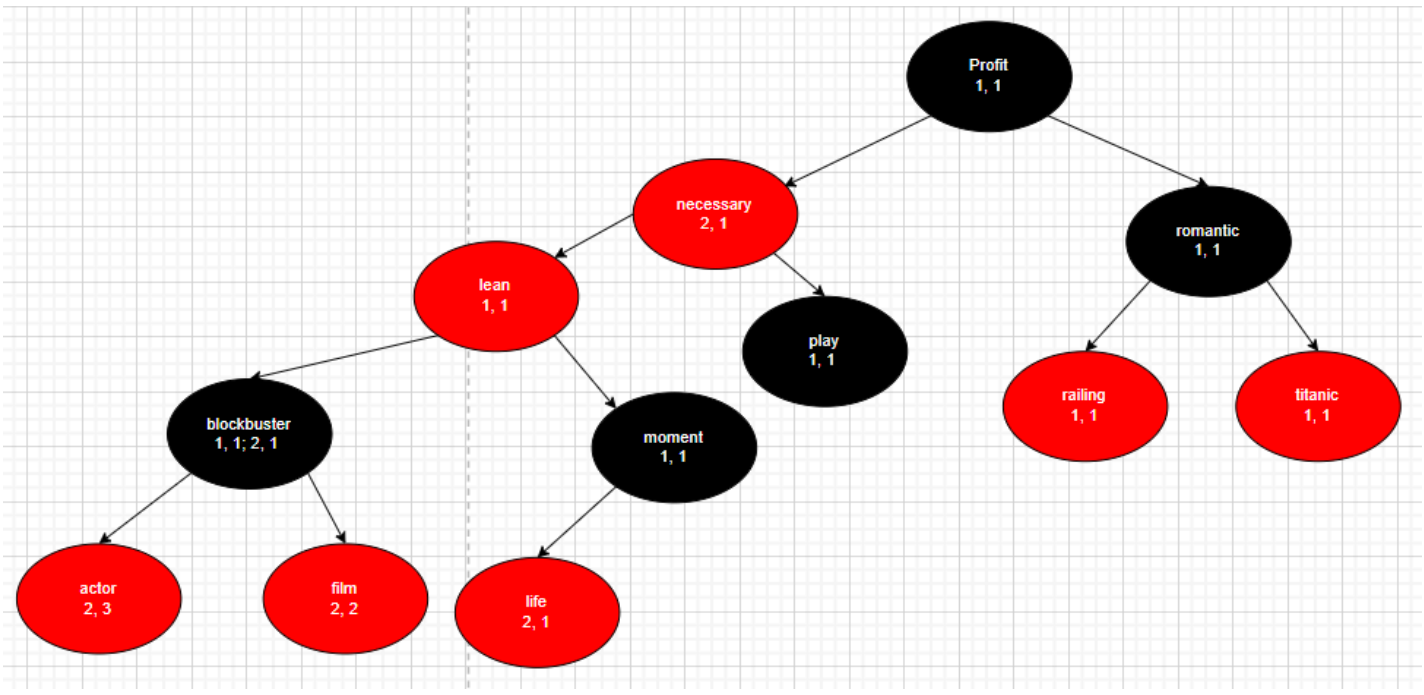
Step 7:

parent and uncle is red so recolor parent and uncle black, recolor grandparent as red.



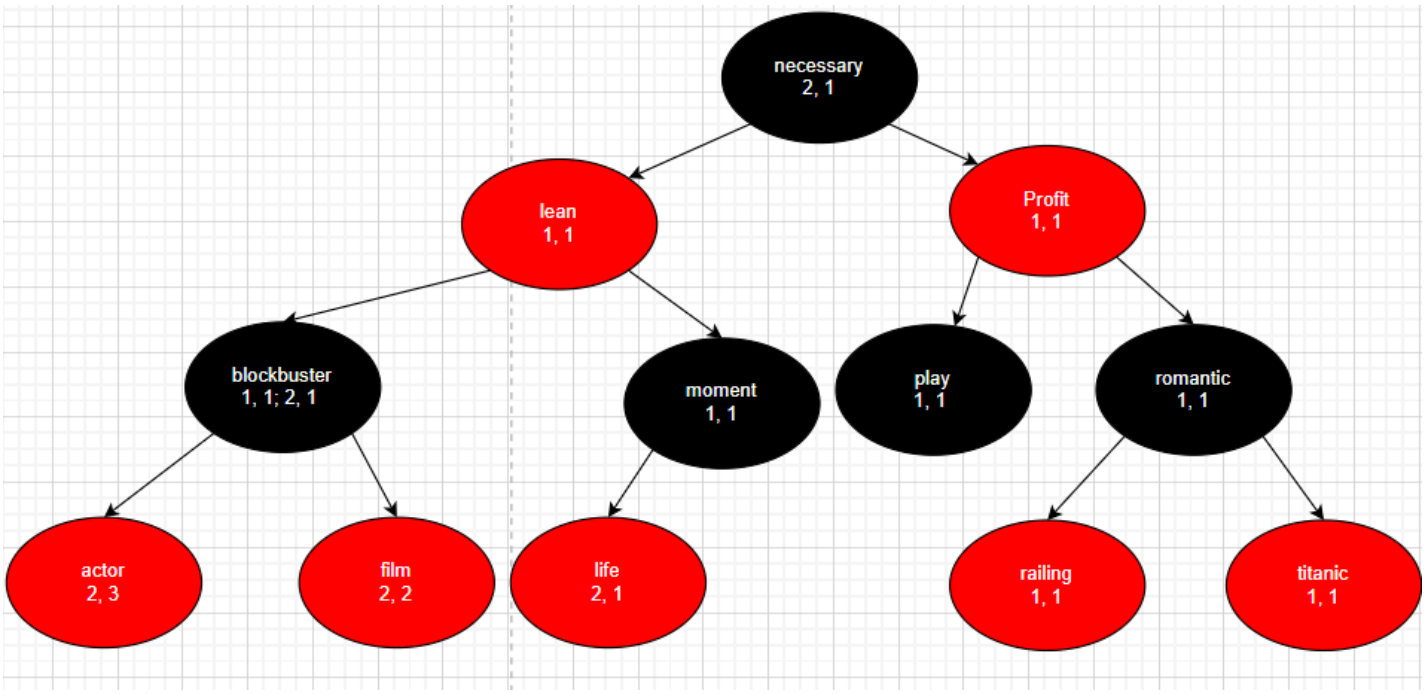
Step 8:

necessary has a red parent, black uncle it is a LR (triangle) configuration so rotate parent left (in the opposite direction of x).

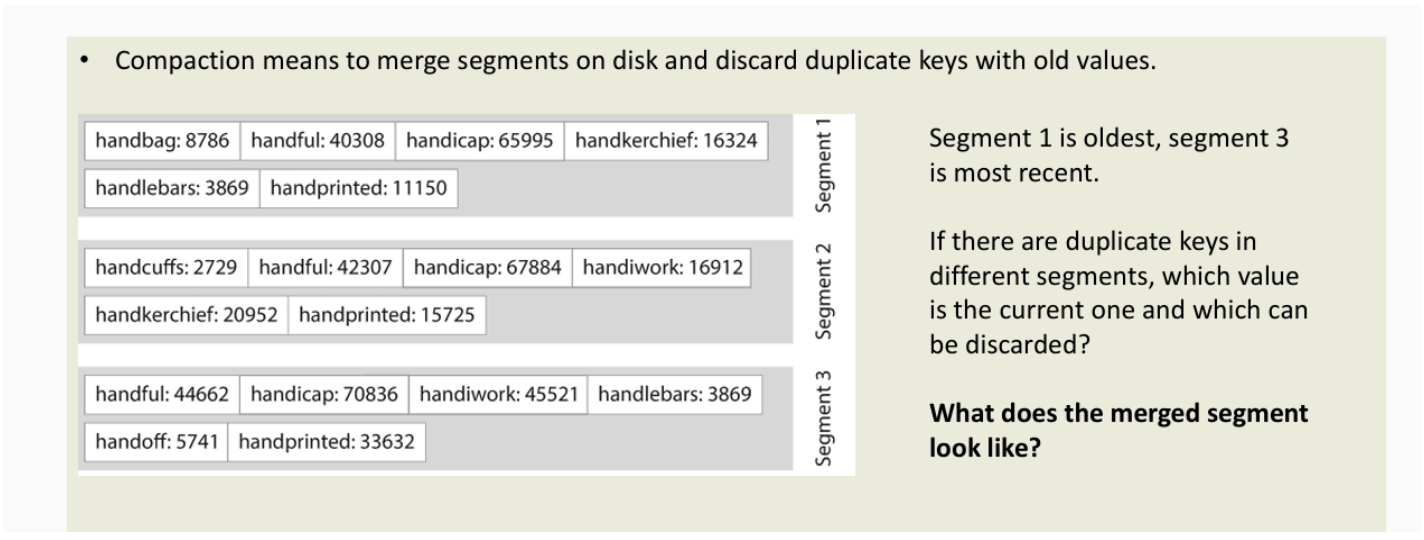


Step 9:

lean has a red parent and black uncle it is an LL(line) configuration so: rotate gp right (in the opposite direction of x), recolor: former p (now root of subtree): black, former gp (now right child: red).



LSM-Storage: 3. Compaction / Merge



if there are duplicate keys in different segments new values (most recent ones) are the current one and old values (from older segments) are discarded.

merged segment:

handbag: 8786, handlebars: 3869, handcuffs: 2729, handful: 44662, handiwork: 45521,
handicap: 70836, handkerchief: 20952, handoff: 5741, handprinted: 33632

LSM-Storage: Delete Operations

Assume that the key-value pair handprinted: 33632 needs to be deleted.
The deletion process in LSM works with a tombstone as value for the key to be deleted.

1. **Explain: What is a tombstone process?**
   A tombstone is a special marker used in Log-Structured Merge (LSM) trees to indicate that a key has been deleted. Instead of immediately removing the key from storage, the system inserts a new entry with the same key and a special tombstone value (often
   `null` or a deletion flag). This marks the key as logically deleted while keeping it in the storage layers temporarily.
2. **Why is a tombstone process necessary in LSM storage? Give a concrete example.**

   LSM storage is write-optimized and relies on immutable, sorted files (SSTables) written in levels. When a key is deleted, it may still exist in older SSTables. A tombstone ensures that when reading or compacting, the system knows not to return or retain older versions of that key.

   Example:

   Suppose key `33632` exists in an old SSTable with value `"handprinted"`. When you want to delete it:

   - A tombstone (`33632 → null`) is written to the **memtable**, then flushed to the **newest SSTable**.
   - If a read query occurs, it will find the tombstone first and **ignore** the older value.
   - During **compaction**, the system will merge files and discard the old `"handprinted"` value based on the tombstone.

   Without the tombstone, the old value could mistakenly reappear after compaction.
3. **When can the tombstone key-value pair be finally be deleted itself?**

   The tombstone can be physically removed during compaction, but only after all older SSTables (which might still contain the deleted key) have been compacted and the key is guaranteed to no longer exist in any level.

   This ensures the deletion is safely propagated throughout the LSM structure before the tombstone is discarded.

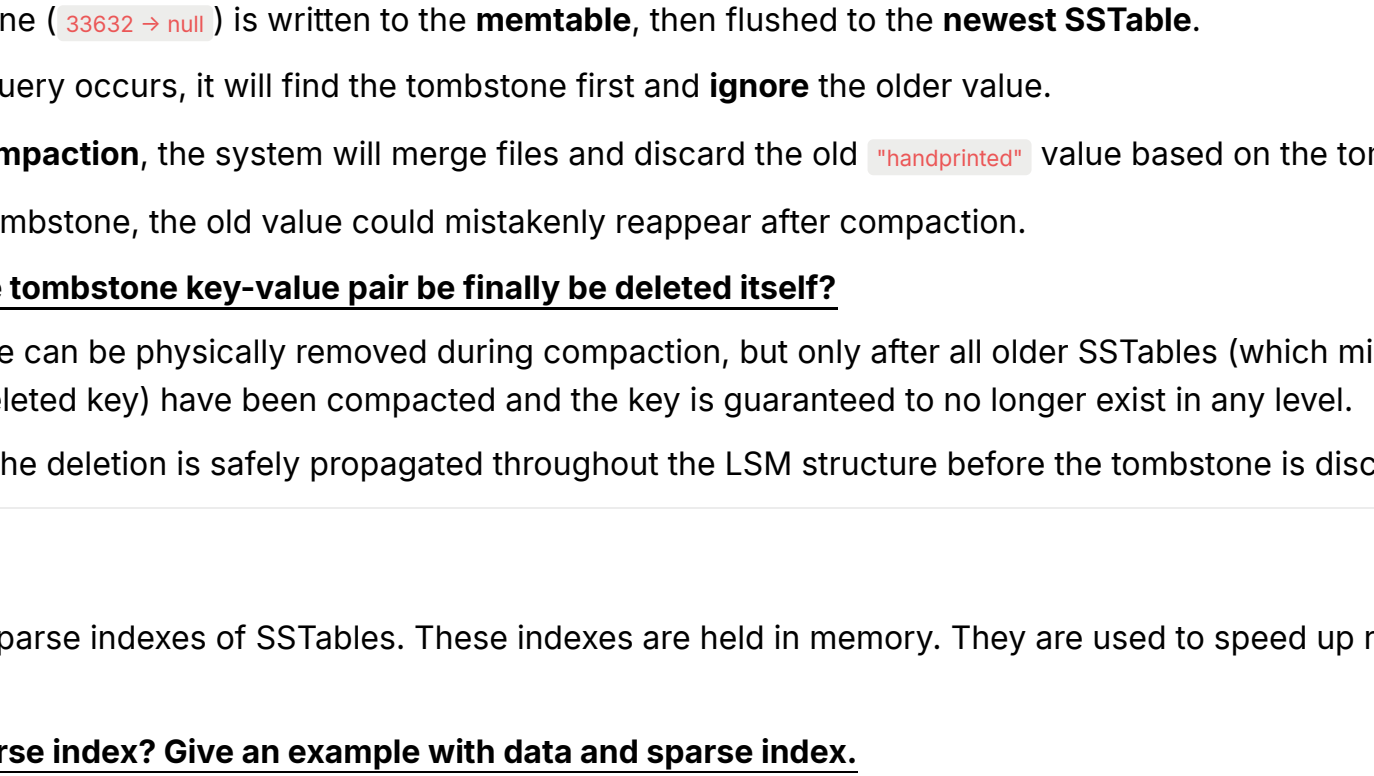
Sparse Index

LSM works with sparse indexes of SSTables. These indexes are held in memory. They are used to speed up read operations

1. **What is a sparse index? Give an example with data and sparse index.**

   A sparse index is an index that holds only a subset of keys from an SSTable (e.g., every N-th key), rather than every key. It points to approximate locations within the SSTable to guide searches efficiently.

   Example:



only some of the keys in a segment are indexed, They point to offsets within the segment files.

2. **Why is it not necessary to hold a full index in memory for LSM?**

   Holding a full index consumes a large amount of memory, especially with large datasets. LSM-trees are optimized for write-heavy workloads, and the actual SSTable files are immutable and sorted.

   Because SSTables are sorted, once the sparse index narrows down the region where a key might exist, a short scan or binary search in that segment is sufficient. Thus, only a partial (sparse) index is enough to locate data efficiently, keeping memory usage low.

3. **How can a sparse index speed up a search request? Give a concrete example.**

   Let's say a query asks for key `1073`.

   The sparse index might contain entries like:

   ```
   1050 → offset A
   1100 → offset B
   ```

   From the index, we know `1073` lies between 1050 and 1100 → So we scan only the small range between offsets A and B (not the entire SSTable).

   This is **much faster** than scanning from the beginning. Even though it's not a direct lookup, it significantly narrows the search space and improves performance.

MongoDB database terminology

RDBMS	Key-Value	Document
database	cluster	database
table	bucket	collection
tuple	key-value	document
PK / rowID	key	objectID

Data Import

my mongosh location:

```
C:\Users\barba\AppData\Local\Programs\mongosh\mongosh.exe
```

1. **Create db lesson in MongoDB**

   // Switches to your "lesson" database or creates it if it does not exist.

   ```
   use lesson
   ```

2. **Create teacher and student collection in MongoDB.**

   // Create and insert a document into the teacher collection

   ```
   db.createCollection("teacher")
   ```

   // Create and insert a document into the student collection

   ```
   db.createCollection("student")
   ```

   If you want to see tables run code:

   ```
   show collections
   ```

3. **Export your teacher and student tables in .json format out of Postgres lesson db**

   firstly, run psql as an administrator, then run the following commands:

```
\c agency
SET client_encoding = 'UTF8';
```

create teacher.json and student.json in the downloads and then run the following command in psql.

```
\copy (SELECT row_to_json(t) FROM teacher t) TO 'C:\Users\barba\Downloads\teacher.json';
```

```
\copy (SELECT row_to_json(t) FROM student t) TO 'C:/Users/barba/Downloads/student.json';
```

4. **Import into MongoDB. You can use Compass GUI to do this.**
   - Open **Compass**.
   - Connect to your MongoDB server.
   - Choose the `lesson` database.
   - Select `teacher` collection → click **Import Data** → upload your `teacher.json`.
   - Do the same for `student` collection.
5. **Each tuple becomes a document.**

   run the following command in mogosh

   ```
   db.teacher.find().pretty()
   ```

```
> db.teacher.find().pretty()
< {
    _id: ObjectId('680b6b54514adfb09a1292de'),
    t_id: 8,
    t_name: 'Duerr',
    t_mail: 'duerr@galopp.xx',
    t_postalcode: 4600,
    t_dob: '2002-06-01',
    t_gender: 'm',
    t_education: 'HighSchool',
    t_remark: null,
    t_payment: 0
  }
  {
    _id: ObjectId('680b6b54514adfb09a1292df'),
    t_id: 9,
    t_name: 'Schumann',
    t_mail: 'schumann@galopp.xx',
    t_postalcode: 4600,
    t_dob: '1994-10-17',
    t_gender: 'f',
    t_education: 'Bachelor',
    t_remark: null,
    t_payment: 0
}
{
    _id: ObjectId('680b6b54514adfb09a1292e0'),
```

each **row** (tuple in relational DB) becomes a **document** inside the collection. This is how a row is now stored as a JSON document.

6. **What does MongoDB add to each document?**
   MongoDB automatically adds a field called:

```
{
  _id: ObjectId('680b6b54514adfb09a1292de'),
```

   This `_id` is a unique identifier for each document, similar to a primary key in relational databases.

7. **What index does MongoDB create on each collection automatically?**

   MongoDB automatically creates a **unique index on the `_id` field** for every collection. So every document is uniquely identified by `_id`, and it is indexed for fast lookup.

Mongo DB Syntax

1. Commands start with: **db.**
2. Followed by the **name of the collection** (to which the operation refers).
3. Then followed by the **method()**

Query commands refer to one collection

Examples:

db.<collectionname>.countDocuments()	Count documents of a collection
db.<collectionname>.find({})	Display all documents of a collection

db.collection.find(query, projection, options)

find() returns a cursor to the documents that match the specified query criteria. Per default, it returns the whole document. If you only want to get certain fields back, you need to add a corresponding projection.

Basic Queries

Counting the number of documents:

```
db.<collectionname>.count()                 #depreciated
db.<collectionname>.countDocuments()        #new
```

Show all documents:

```
db.<collectionname>.find({})
```

Pretty output format: (line breaks, one value per line):

```
db.<collectionname>.find({}).pretty()
```

Return documents, sorted by name, limit number of returned documents:

```
db.teacher.find({}).sort({name:-1}).limit(5)
```

Return only documents that match a query condition, return only specific fields (projection):

```
db.teacher.find({gender: "m", },{name:1, gender:1})
```

Return documents that match a query condition, return all fields except DoB:

```
db.teacher.find({education: "Master", },{dob:0})
```

Basic Queries Get some practice

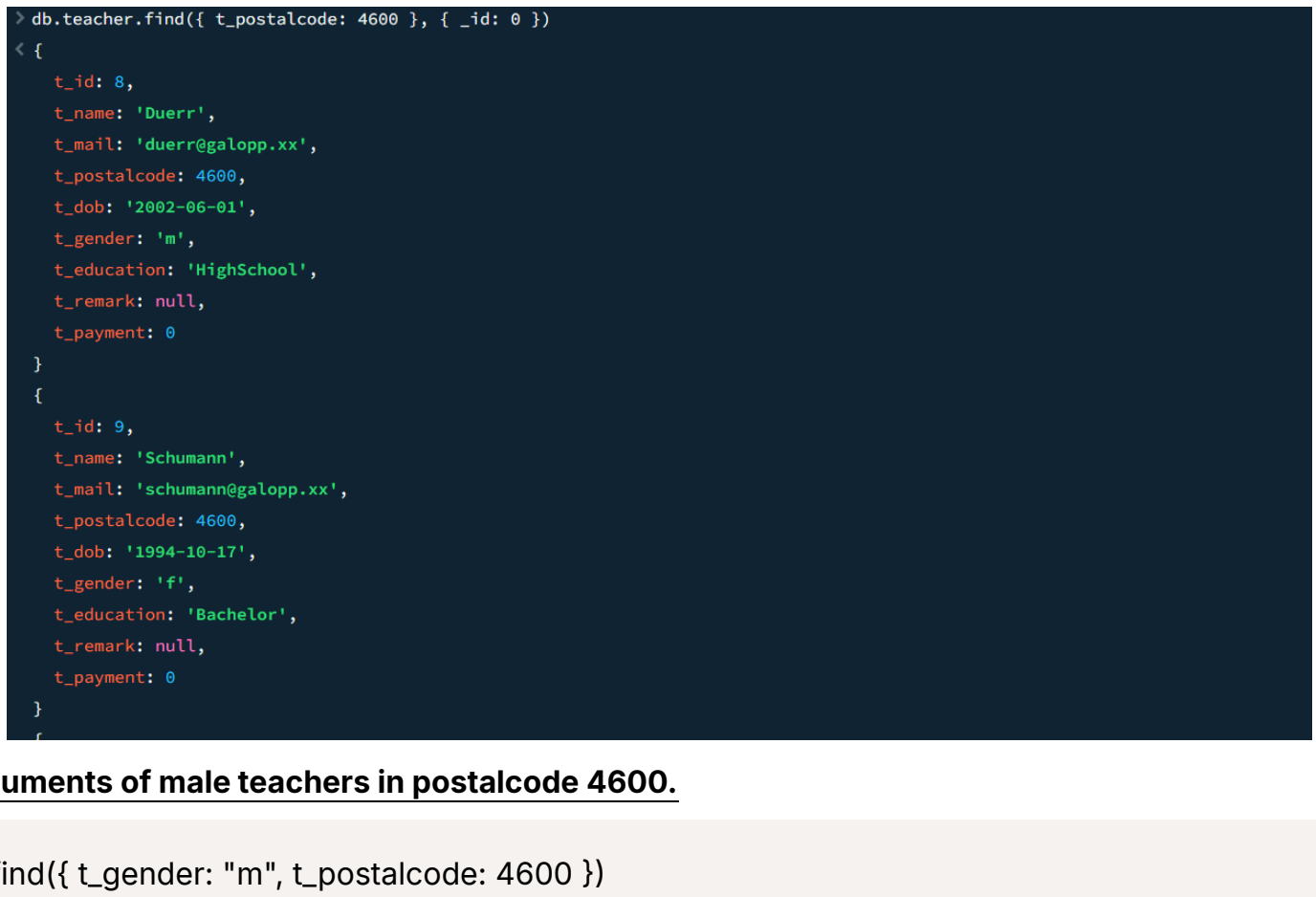
1. **How many male teachers are in teacher collection?**

```
db.teacher.countDocuments({ t_gender: "m" })
```

```
> db.teacher.countDocuments({ t_gender: "m" })
< 10
```

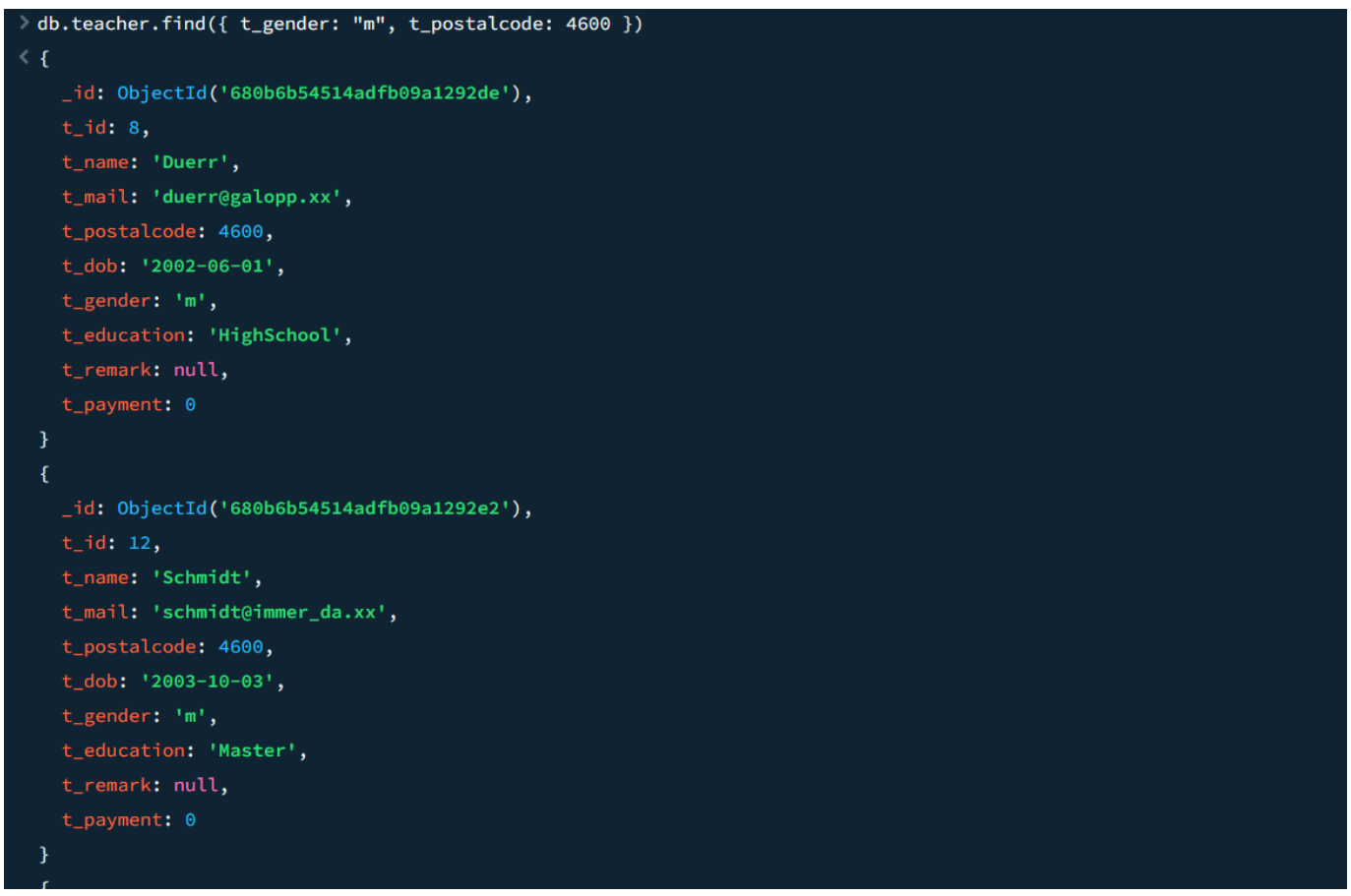
2. **Return documents of teachers that live in postalcode 4600. Do not show object_id.**

```
db.teacher.find({ t_postalcode: 4600 }, { _id: 0 })
```



3. **Return all documents of male teachers in postalcode 4600.**

```
db.teacher.find({ t_gender: "m", t_postalcode: 4600 })
```



4. **Return the documents of teachers Krawinkel and Kaiser (or condition on any other two teachers)**

```
db.teacher.find({ t_name: { $in: ["Krawinkel", "Kaiser"] } })
```

```
> db.teacher.find({ t_name: { $in: ["Krawinkel", "Kaiser"] } })
< {
    _id: ObjectId('680b6b54514adfb09a1292f0'),
    t_id: 2,
    t_name: 'Kaiser',
    t_mail: 'kaiser@circus.xx',
    t_postalcode: 4500,
    t_dob: '2001-11-30',
    t_gender: 'f',
    t_education: 'HighSchool',
    t_remark: 'The relational model was a theoretical proposal, and many people at the time doubted whether it could be implemented efficien
    t_payment: 6
  }
  {
    _id: ObjectId('680b6b54514adfb09a1292f1'),
    t_id: 1,
    t_name: 'Krawinkel',
    t_mail: 'krawinkel@we-are.xx',
    t_postalcode: 4500,
    t_dob: '1978-09-15',
    t_gender: 'f',
    t_education: 'Master',
    t_remark: 'In a complex application there may be more intermediary levels, such as APIs built upon APIs, but the basic idea is still the
    t_payment: 10
```

5. **Teacher Krawinkel teaches EN, DE and FR. Add the subjects to the document. (updateOne())**

```
db.teacher.updateOne(
  { t_name: "Krawinkel" },
  { $set: { subjects: ["EN", "DE", "FR"] } }
)
```

```
> db.teacher.updateOne(
    { t_name: "Krawinkel" },
    { $set: { subjects: ["EN", "DE", "FR"] } }
  )
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
```

```
> db.teacher.find({t_name: "Krawinkel"})
< {
    _id: ObjectId('680b6b54514adfb09a1292f1'),
    t_id: 1,
    t_name: 'Krawinkel',
    t_mail: 'krawinkel@we-are.xx',
    t_postalcode: 4500,
    t_dob: '1978-09-15',
    t_gender: 'f',
    t_education: 'Master',
    t_remark: 'In a complex application there may be more intermediary levels, such as APIs built upon APIs, but the basi
    t_payment: 10,
    subjects: [
      'EN',
      'DE',
      'FR'
    ]
}
```

6. **Teacher Alt teaches MA, PH and CH. Add the subjects to the doucment**

```
db.teacher.updateOne(
  { t_name: "Alt" },
  { $set: { subjects: ["MA", "PH", "CH"] } }
)
```

```
> db.teacher.updateOne(
    { t_name: "Alt" },
    { $set: { subjects: ["MA", "PH", "CH"] } }
  )
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
> db.teacher.find({t_name: "Alt"})
< {
    _id: ObjectId('680b6b54514adfb09a1292e7'),
    t_id: 18,
    t_name: 'Alt',
    t_mail: 'alt@neu.xx',
    t_postalcode: 4600,
    t_dob: '1965-05-03',
    t_gender: 'm',
    t_education: 'Master',
    t_remark: null,
    t_payment: 0,
    subjects: [
      'MA',
      'PH',
      'CH'
    ]
}
```