DBMS PROJECT REPORT

Sneha Hegde

PES1201801157

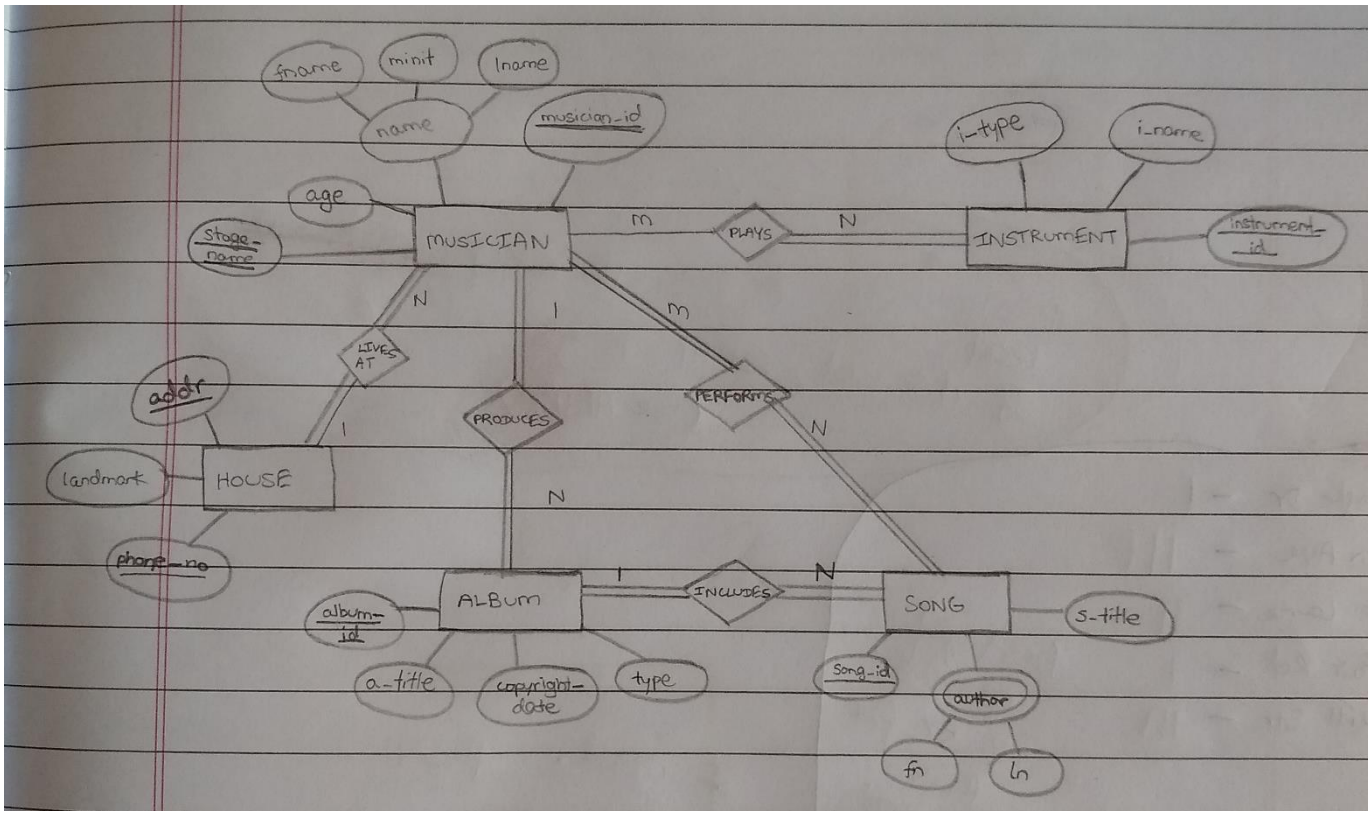
- Universe of Discourse chosen: Record Label Company

- Requirements:

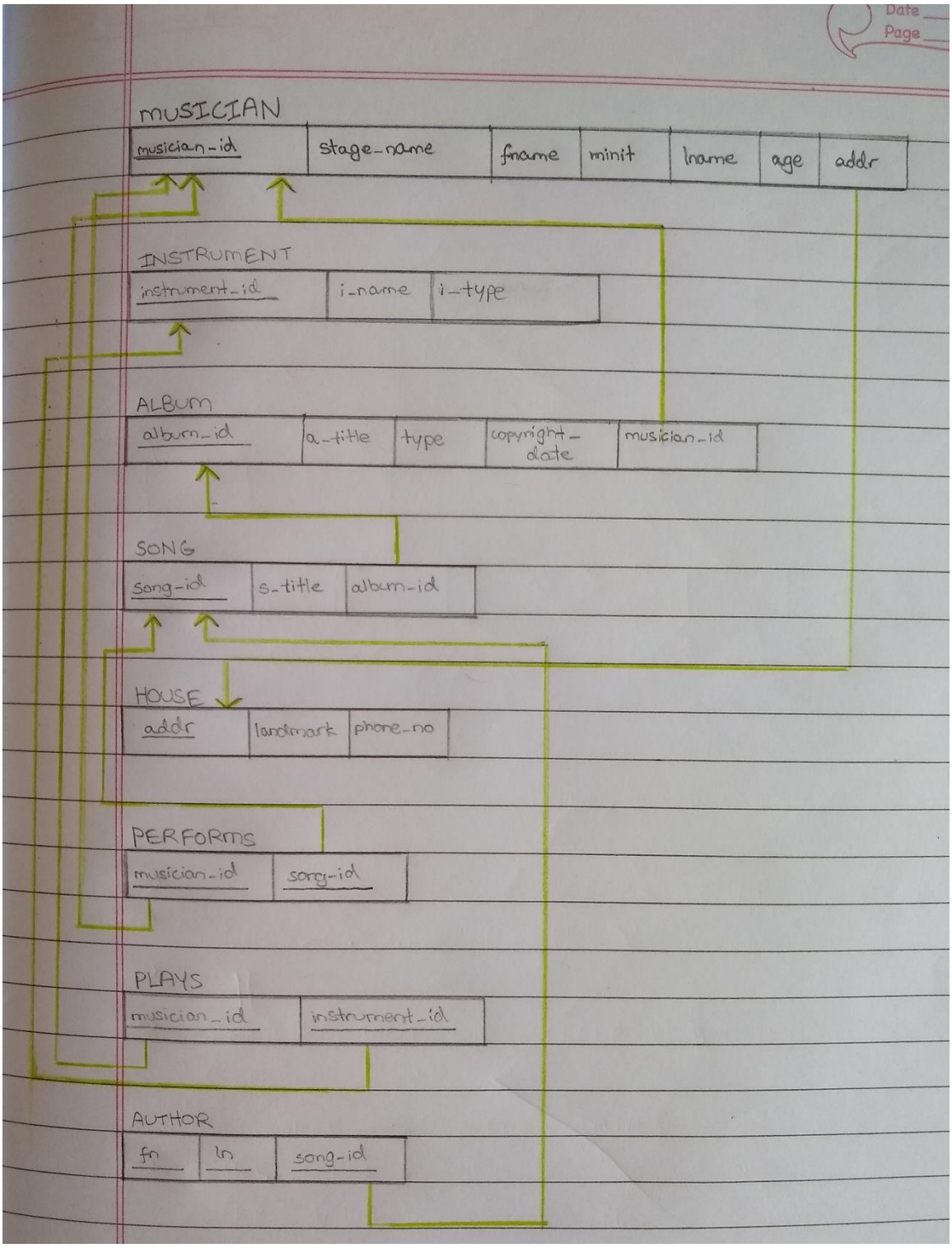
  Our record label company, Spin It, was founded on April 12, 1973, and has been bringing captivating music out into the world ever since.

  - ✓ Each musician that records at this company has a unique id, a name (first name, middle initial (optional), last name), a stage name, an age and a house address.
    No two musicians may have the same stage name.
    Musicians may share a house address, but no more than 3 of our musicians may live in the same house.
    Every house has exactly one phone, and a landmark to identify the location of the address.
  - ✓ Each instrument in these recorded songs has a unique instrument id, a name and an instrument type.
  - ✓ Each recorded album has a unique album id, a title, a copyright date and an album type.
  - ✓ Each song recorded at this company has a unique song id, a title and a set of authors.
  - ✓ Each musician may play several instruments (or none at all), and a given instrument may be played by several musicians.
  - ✓ Each album has a number of songs on it, but no song may appear on more than one album.
  - ✓ Each song is performed by one or more musicians, and a musician may perform a number of songs.
  - ✓ Each album has exactly one musician who acts as its producer, and a musician may produce several albums.

❖ Conceptual Model, aka ER Diagram:



❖ Relational Schema (spin_it_db):



*Knowledge that ‘phone_no’ and ‘stage_name’ are unique keys is kept for later use in the design.

❖ Functional Dependencies and Keys:

- The FDs of the schema:

  Let us try to find all the FDs (including trivial FDs) of the SONG relation, with respect to our database requirements.

  We need to start by thinking of all possible combinations of the number of attributes in the LHS and RHS. After that, we need to find all the FDs which fall into each category.
  In this instance, we have 3 attributes. Therefore, the categories are 1 attribute on the LHS and 1 on the RHS (1 → 1), 1 attribute on the LHS and 2 on the RHS (1 → 2), and so on. We need to find all the possible FDs which fall into all these categories.

  - 1 → 1: {song_id} → {song_id}, {song_id} → {s_title}, {song_id} → {album_id}
  - 1 → 2: {song_id} → {s_title, album_id}, {song_id} → {song_id, s_title},
  - {song_id} → {song_id, album_id}
  - 1 → 3: {song_id} → {song_id, s_title, album_id}
  - 2 → 1: {song_id, s_title} → {song_id}, {song_id, s_title} → {s_title}, {song_id, s_title} → {album_id}, {song_id, album_id} → {song_id}, {song_id, album_id} → {s_title}, {song_id, album_id} → {album_id}
  - 2 → 2: {song_id, s_title} → {song_id, s_title}, {song_id, s_title} → {s_title, album_id}, {song_id, s_title} → {song_id, album_id}, {song_id, album_id} → {song_id, s_title}, {song_id, album_id} → {s_title, album_id}, {song_id, album_id} → {song_id, album_id}
  - 2 → 3: {song_id, s_title} → {song_id, s_title, album_id}, {song_id, album_id} → {song_id, s_title, album_id}
  - 3 → 1: {song_id, s_title, album_id} → {song_id}, {song_id, s_title, album_id} → {s_title}, {song_id, s_title, album_id} → {album_id}
  - 3 → 2: {song_id, s_title, album_id} → {song_id, s_title}, {song_id, s_title, album_id} → {s_title, album_id}, {song_id, s_title, album_id} → {song_id, album_id}
  - 3 → 3: {song_id, s_title, album_id} → {song_id, s_title, album_id}

  Therefore, the above are all the FDs derived from the SONG relation.

  Note: If a certain set of FDs was already given to us, along with the relation, we could've used the inference rules to determine the closure for that set. We would've ended up with the same answer.

We can use the same algorithm to find all the FDs of the other relations. Some other examples of FDs in the schema are:

- {musician_id} → {stage_name, fname, minit, lname, age, addr},
- {stage_name} → {musician_id, fname, minit, lname, age, addr},
- {instrument_id} → {i_name, i_type},
- {album_id} → {a_title, type, copyright_date, musician_id},
- {addr} → {landmark, phone_no},
- {phone_no} → {addr, landmark},
- {musician_id, song_id} → {stage_name, fname, minit, lname, age, addr, s_title, album_id},
- {musician_id, instrument_id} → { stage_name, fname, minit, lname, age, addr, i_name, i_type},
- {fn, ln, song_id} → {s_title, album_id}

We can derive more FDs from the above list by using inference rules.

- The keys of the schema:

We try to find the keys of the relations using the FDs of that relation.
Let us test it for the relation SONG, assuming that we have no prior knowledge about the keys, and that only the relation has been given to us. We need to detect the key(s) of this relation.

Let us first assume that all the attributes form the key (default superkey). Let's remove attributes one by one, and see if they still form keys.
When we remove album_id, we're left with song_id and s_title. Looking at our set of functional dependencies for this relation, we can see that song_id and s_title are sufficient to uniquely determine all the attributes. Therefore, it is still a superkey. Let's now try to remove song_id. We're left with s_title. As we can see, only s_title is not enough to uniquely determine any of the attributes, as several songs may have the same title. In other words, it cannot appear on the LHS of a FD on its own. So, s_title is not a key. Let us bring back song_id. We have song_id and s_title, which we know is a superkey. Let's remove s_title now, so that we're left with just song_id. As we can see, song_id is enough to uniquely determine all the other attributes. It can appear as a standalone in the LHS of a functional dependency. Hence, song_id is the key of the SONG relation.

Note: A key can appear on the LHS of any FD as a stand-alone, as it is enough to uniquely identify other attributes.

We can use the same algorithm to determine the keys of the rest of the relations. Combining that with the knowledge of the database requirements, we get the following information-

- Primary Keys:
  - ✓ musician_id (MUSICIAN)
  - ✓ instrument_id (INSTRUMENT)
  - ✓ album_id (ALBUM)
  - ✓ song_id (SONG)
  - ✓ addr (HOUSE)
  - ✓ musician_id, song_id (PERFORMS)
  - ✓ musician_id, instrument_id (PLAYS)
  - ✓ fn, ln, song_id (AUTHOR)

- Secondary Candidate Keys:
  - ✓ stage_name (MUSICIAN)
  - ✓ phone_no (HOUSE)

- Foreign Keys:
  - ✓ addr (MUSICIAN)
  - ✓ musician_id (ALBUM)
  - ✓ album_id (SONG)
  - ✓ musician_id, song_id (PERFORMS)
  - ✓ musician_id, instrument_id (PLAYS)
  - ✓ song_id (AUTHOR)

❖ Normalisation of relations:

Normalisation is an integral part of relational schema optimisation. It gets rid of errors in the relational schema due to possible errors in the conceptual schema, or incorrect conceptual to relational schema mapping.

- First Normal Form:

  It is based on disallowing multivalued, composite and complex attributes. The domain of an attribute in a relation must contain only atomic values and the value of the attribute must be a single value from the domain. We cannot have relations within a relation as a result of a complex attribute.

  There are several ways to overcome this and bring the relations to 1NF:

  - We can decompose the relation by creating a new relation with the attribute which violates 1NF and the primary key of the original relation. The resulting PK is the combination of these two.
  - We can break down the violating attribute within the same relation, allowing it to take only one value at a time, hence, splitting each tuple into multiple tuples. The new PK would be a combination of the original PK and the violating attribute.
  - If we know the minimum and maximum number of values the violating attribute can take, we can split it up into new

attributes within the same relation, where each new attribute takes a single value.

Out of these three, the first one is the best solution. The second one has a high amount of redundancy, and a higher chance of data inconsistency. The third one is inefficient when it comes to querying, and also has a high scope for many NULL values. Hence, it makes the most sense to go ahead with the first solution.

Looking at our schema, we can see that this has already been done during the mapping stage, from the conceptual to the relational schema.

Name attribute in the MUSICIAN entity is composite. In the relational schema, we have broken it down into its atomic attributes, i.e., fname, minit and lname.

Author in the SONG entity is a complex attribute. We have followed solution number one, and we've created a new relation for author, where we've broken it down into fn and ln, including them in the new relation. We've also included the PK of the parent relation, i.e., song_id from SONG, and the PK of the new relation is the combination of all these three attributes.

In this way, we've brought our schema to 1NF.

- Second Normal Form:

  It is based on the concept of full FD. A FD is said to be a full FD if, when we remove even one attribute from the LHS set, the FD doesn't hold good (if it does, it is called a partial dependency).
  For a relation to be in 2NF, every non-prime attribute must be fully functionally dependent on the whole of the PK.
  In other words, in a relation, if the PK consists of more than one attribute, and if a non-prime attribute can be identified using not all of the PK attributes, the relation is not in 2NF.
  (It separates all the FDs where the LHS is a proper subset of the PK)

  To overcome a violation of 2NF, we decompose the original relation and create a new relation, which consists of the violating attribute, along with the part of the PK on which it is fully functionally dependent.

  In our schema, it is the case that the relation has only one PK, or all the attributes in the relation form the PK (no non-prime attributes). Therefore, the schema is already in 2NF, and further tests can't be done. We can extend this to candidate keys, and not just PKs, and we'll get the same result.

- Third Normal Form:

  It is based on the concept of transitive dependency.
  A FD $X \rightarrow Z$ is said to be transitive, if there exists a Y (which is neither a candidate key nor a subset of any key), such that $X \rightarrow Y$ and $Y \rightarrow Z$ both hold good.

  A relation is in 3NF if:

  - it is in 2NF
  - no non-prime attribute is transitively dependent on the PK

  (It gets rid of all the FDs where the LHS is a non-key attribute)

  To overcome a violation of 3NF, where we have $X \rightarrow Y$, $Y \rightarrow Z$ and $X \rightarrow Z$, where Y is neither a candidate key nor a subset of a key, we split the relation, by retaining Y in the original and removing Z, and creating a new relation for Y and Z, where Y becomes the PK of the new relation.

  In our schema, all the relations are already in 2NF and there is no Y for any of the relations, unless it's a candidate key, which is not allowed. Hence, no transitive dependencies are formed. Therefore, our schema is in 3NF. We can extend this to candidate keys, and not just PKs, and we'll get the same result.

- Boyce-Codd Normal Form:

  For a relation to be in BCNF, all the FDs (not including trivial FDs) have to be of the form $X \rightarrow Y$, where X is a superkey and Y is a non-prime attribute.
  BCNF adds on to 3NF. Therefore, all relations in BCNF are already in 3NF, but not always the other way around.

  To bring a relation R to BCNF, we identify the FD which causes the violation in the relation. Let it be $X \rightarrow A$. We split the relation into R – A and XA and test them using the Nonadditive Join Test for Binary Decompositions. If there is still a violation, we repeat.

  In our schema, we can see that all the nontrivial FDs of each relation have only superkeys on the LHS and non-prime attributes on the RHS. Therefore, the schema is in BCNF.

- Fourth Normal Form:

  It is based on the concept of multivalued dependencies.

It arises as a consequence of 1NF. When we try to bring a relation to 1NF using the second method, we end up with multivalued dependencies (where a set of tuples may determine several different values of a multivalued attribute). They may also arise as a result of mixing two independent 1:N relationships.

While mapping our conceptual schema onto our relational schema, we've taken care to not merge any of the relationships, and we've taken care of the multivalued attributes by creating new relations for them. Hence, we don't have any multivalued dependencies in one schema. Therefore, our schema is in 4NF.

- Fifth Normal Form:

  It is based on the concept of join dependency. It is hard to detect and rarely done in practice. A simple way to check if a schema is 5NF, disregarding join dependencies and only considering FDs, is to make sure it's in 3NF and if each key consists of a single attribute. If yes, the schema is in 5NF.

  Therefore, our schema is in 5NF.

❖ Test for Lossless/Nonadditive Join Property:

When a relation is decomposed, we need to be able to join them back together to get the original, without generating spurious tuples. The Nonadditive Join Property ensures this. But it can't happen when there's no common attribute(s) between the relations, where the common attribute is the PK in one relation, and a FK in the other.

We can test if the decomposed relations $\{R_1, R_2, \ldots, R_n\}$ of a universal relation R have the NJP, when we know the attributes in each child relation and when we have a set of FDs.
The steps of the NJP test are as follows:

1. Create an initial matrix S, where each row corresponds to a child relation, and each column corresponds to an attribute in R.
2. Fill each cell of S with b_{ij}, where i and j are the row and column numbers, respectively.
3. For each child relation (row) ,
   for each attribute (column),
   if that attribute is a part of that relation, fill that cell with a_j, where j is the column number.
4. For each FD $X \rightarrow Y$ in the given set,
   find all the rows ROWS which have the same symbol(s) in the columns corresponding to the attribute(s) in X (equality of symbols includes their numbers, not just their letters. Ex- $b_{13} \neq b_{42}$).
   If no such rows, move on to the next FD and repeat.

For each attribute in Y,

check if any of the rows in ROWS has an *a* symbol for that attribute.

If yes,

set the other rows in ROWS to that same *a* symbol for that attribute.

If no,

choose any one *b* symbol for that particular attribute from any row in ROWS and set the other rows in ROWS to that same *b* symbol for that attribute.

5. At the end, if there is any row with all ‘a’s, the decomposition has NJP.

Now, we need to test for NJP on our relations. Let us consider 4 sets of cases, where the sets are independent. (Refer to the relations on page 3)

- Set one is the universal relation consisting of attributes from the relations MUSICIAN and HOUSE.
  The associated FDs are
  F=
  {
  {musician_id} → {stage_name, fname, minit, lname, age, addr},
  {stage_name} → {musician_id, fname, minit, lname, age, addr},
  {addr} → {landmark, phone_no},
  {phone_no} → {addr, landmark}
  }

  Since it is a binary decomposition, we can use the NJB test.
  The NJB test says that if the universal relation R is decomposed into R1 and R2 with respect to a set of FDs F, they satisfy the property only when
  the FD ((R1 ∩ R2) → (R1 − R2)) is in F^+, or
  the FD ((R1 ∩ R2) → (R2 − R1)) is in F^+.

  In this case, MUSICIAN ∩ HOUSE = addr.

  HOUSE – MUSICIAN = landmark, phone_no.

  {addr} → {landmark, phone_no} is a part of F^+.

  Therefore, this set satisfies NJB.

- Set two is the universal relation consisting of attributes from the relations MUSICIAN, ALBUM, SONG and PERFORMS.
  The associated FDs are
  F=

{

{musician_id} → {stage_name, fname, minit, lname, age, addr},
{stage_name} → {musician_id, fname, minit, lname, age, addr},
{album_id} → {a_title, type, copyright_date, musician_id},
{song_id} → {s_title, album_id},
{musician_id, song_id} → {stage_name, fname, minit, lname, age, addr, s_title, album_id}

}

Following the steps of the NJP algorithm,

- Steps 1 and 2:

	musician_id	stage_name	fname	minit	lname	age	album_id	a_title	type	Copyright_date	song_id	s_title	addr
MUSICIAN	b11	b12	b13	b14	b15	b16	b17	b18	b 19	b1 10	b1 11	b1 12	b1 13
ALBUM	b21	b22	b23	b24	b25	b26	b27	b28	b29	b2 10	b2 11	b2 12	b2 13
SONG	b31	b32	b33	b34	b35	b36	b37	b38	b39	b3 10	b3 11	b3 12	b3 13
PERFORMS	b41	b42	b43	b44	b45	b46	b47	b48	b49	b4 10	b4 11	b4 12	b4 13

- Step 3:

	musician_id	stage_name	fname	minit	lname	age	album_id	a_title	type	Copyright_date	song_id	s_title	addr
MUSICIAN	a1	a2	a3	a4	a5	a6	b17	b18	b19	b1 10	b1 11	b1 12	a13
ALBUM	a1	b22	b23	b24	b25	b26	a7	a8	a9	a10	b2 11	b2 12	b2 13
SONG	b31	b32	b33	b34	b35	b36	a7	b38	b39	b3 10	a11	a12	b3 13
PERFORMS	a1	b42	b43	b44	b45	b46	b47	b48	b49	b4 10	a11	b4 12	b4 13

- Step 4:
  - Apply {musician_id} → {stage_name, fname, minit, lname, age, addr}

	musician_id	stage_name	fname	minit	lname	age	album_id	a_title	type	Copyright_date	song_id	s_title	addr
MUSICIAN	a1	a2	a3	a4	a5	a6	b17	b18	b19	b1 10	b1 11	b1 12	a13
ALBUM	a1	~~b22~~ a2	~~b23~~ a3	~~b24~~ a4	~~b25~~ a5	~~b26~~ a6	a7	a8	a9	a10	b2 11	b2 12	~~b2 13~~ a13
SONG	b31	b32	b33	b34	b35	b36	a7	b38	b39	b3 10	a11	a12	b3 13
PERFORMS	a1	~~b42~~ a2	~~b43~~ a3	~~b44~~ a4	~~b45~~ a5	~~b46~~ a6	b47	b48	b49	b4 10	a11	b4 12	~~b4 13~~ a13

ROWS={MUSICIAN, ALBUM, PERFORMS}

  - Apply {stage_name} → {musician_id, fname, minit, lname, age, addr}

    ROWS={MUSICIAN, ALBUM, PERFORMS}

    Applying this FD gives the same table as the previous one, as the LHSs of both FDs are keys of the same relation.

- Apply {album_id} → {a_title, type, copyright_date, musician_id}

	musician_id	stage_name	fname	minit	lname	age	album_id	a_title	type	Copyright_date	song_id	s_title	addr
MUSICIAN	a1	a2	a3	a4	a5	a6	b17	b18	b19	b1 10	b1 11	b1 12	a13
ALBUM	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	b2 11	b2 12	a13
SONG	~~b31~~ a1	b32	b33	b34	b35	b36	a7	~~b38~~ a8	~~b39~~ a9	~~b3 10~~ a10	a11	a12	b3 13
PERFORMS	a1	a2	a3	a4	a5	a6	b47	b48	b49	b4 10	a11	b4 12	a13

ROWS={ALBUM, SONG}

- Apply {song_id} → {s_title, album_id}

	musician_id	stage_name	fname	minit	lname	age	album_id	a_title	type	Copyright_date	song_id	s_title	addr
MUSICIAN	a1	a2	a3	a4	a5	a6	b17	b18	b19	b1 10	b1 11	b1 12	a13
ALBUM	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	b2 11	b2 12	a13
SONG	a1	b32	b33	b34	b35	b36	a7	a8	a9	a10	a11	a12	b3 13
PERFORMS	a1	a2	a3	a4	a5	a6	~~b47~~ a7	b48	b49	b4 10	a11	~~b4 12~~ a12	a13

ROWS={SONG, PERFORMS}

- Apply {musician_id, song_id} → {stage_name, fname, minit, lname, age, addr, s_title, album_id}

	musician_id	stage_name	fname	minit	lname	age	album_id	a_title	type	Copyright_date	song_id	s_title	addr
MUSICIAN	a1	a2	a3	a4	a5	a6	b17	b18	b19	b1 10	b1 11	b1 12	a13
ALBUM	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	b2 11	b2 12	a13
SONG	a1	~~b32~~ a2	~~b33~~ a3	~~b34~~ a3	~~b35~~ a5	~~b36~~ a6	a7	a8	a9	a10	a11	a12	~~b3 13~~ a13
PERFORMS	a1	a2	a3	a4	a5	a6	a7	b48	b49	b4 10	a11	a12	a13

ROWS={SONG, PERFORMS}

- Step 5: We can see that the row SONG consists of all ‘a’s.

Therefore, this set satisfies NJP.

♦ Set three is the universal relation consisting of attributes from the relations MUSICIAN, INSTRUMENT and PLAYS.
The associated FDs are
F=
{
{musician_id} → {stage_name, fname, minit, lname, age, addr},
{stage_name} → {musician_id, fname, minit, lname, age, addr},
{instrument_id} → {i_name, i_type },
{musician_id, instrument_id} → {stage_name, fname, minit, lname, age, addr, i_name, i_type }

}

Following the steps of the NJP algorithm,

- Steps 1 and 2:

	musician_id	stage_name	fname	minit	lname	age	instrument_id	i_name	i_type	addr
MUSICIAN	b11	b12	b13	b14	b15	b16	b17	b18	b19	b1 10
INSTRUMENT	b21	b22	b23	b24	b25	b26	b27	b28	b29	b2 10
PLAYS	b31	b32	b33	b34	b35	b36	b37	b38	b39	b3 10

- Step 3:

	musician_id	stage_name	fname	minit	lname	age	instrument_id	i_name	i_type	addr
MUSICIAN	a1	a2	a3	a4	a5	a6	b17	b18	b19	a10
INSTRUMENT	b21	b22	b23	b24	b25	b26	a7	a8	a9	b2 10
PLAYS	a1	b32	b33	b34	b35	b36	a7	b38	b39	b3 10

- Step 4:
  - Apply {musician_id} → {stage_name, fname, minit, lname, age, addr},

	musician_id	stage_name	fname	minit	lname	age	instrument_id	i_name	i_type	addr
MUSICIAN	a1	a2	a3	a4	a5	a6	b17	b18	b19	a10
INSTRUMENT	b21	b22	b23	b24	b25	b26	a7	a8	a9	b2 10
PLAYS	a1	~~b32~~ a2	~~b33~~ a3	~~b34~~ a4	~~b35~~ a5	~~b36~~ a6	a7	b38	b39	~~b3 10~~ a10

    ROWS={MUSICIAN, PLAYS}

  - Apply {stage_name} → {musician_id, fname, minit, lname, age, addr}

    ROWS={MUSICIAN, PLAYS}

    Applying this FD gives the same table as the previous one, as the LHSs of both FDs are keys of the same relation.

  - Apply {instrument_id} → {i_name, i_type }

	musician_id	stage_name	fname	minit	lname	age	instrument_id	i_name	i_type	addr
MUSICIAN	a1	a2	a3	a4	a5	a6	b17	b18	b19	a10

INSTRUMENT	b21	b22	b23	b24	b25	b26	a7	a8	a9	b2 10
PLAYS	a1	a2	a3	a4	a5	a6	a7	~~b38~~ a8	~~b39~~ a9	a10

ROWS={INSTRUMENT, PLAYS}

- Step 5: We can see that the row PLAYS consists of all ‘a’s.

Therefore, this set satisfies NJP.

♦ Set four is the universal relation consisting of attributes from the relations SONG and AUTHOR.
The associated FDs are
F=
{
{song_id} → {s_title, album_id},
{fn, ln, song_id} → {s_title, album_id}

}

Since it is a binary decomposition, we can use the NJB’ test for multivalued dependencies.
The NJB’ test says that if the universal relation R is decomposed into R1 and R2 with respect to a set of FDs and multivalued dependencies, they satisfy the property only when
the FD ((R1 ∩ R2) ↠ (R1 − R2)) or
the FD ((R1 ∩ R2) ↠ (R2 − R1)) holds good.

In this case, SONG ∩ AUTHOR = song_id.

AUTHOR – SONG = fn, ln.

{song_id} ↠ {fn, ln} holds good.

Therefore, this set satisfies NJB.

Therefore, all of the sets satisfy NJP.

❖ SQL Section:
- After creating the database, we create its tables by running the script below.

create_tables.sql

We have successfully created the tables, with all the check constraints and referential integrity constraints.

```
spin_it_db=# \d
          List of relations
 Schema |    Name    | Type  |  Owner
--------+------------+-------+----------
 public | album      | table | postgres
 public | author     | table | postgres
 public | house      | table | postgres
 public | instrument | table | postgres
 public | musician   | table | postgres
 public | performs   | table | postgres
 public | plays      | table | postgres
 public | song       | table | postgres
(8 rows)
```

- Let us try to write a trigger, with respect to the requirement that, at max, three musicians can share a house. Running the script below, we get

trigger.sql

```
CREATE FUNCTION
CREATE TRIGGER
spin_it_db-#
```

The trigger has been created.

- Now, we insert values into the tables by running the script below.

insertion.sql

```
INSERT 0 5
INSERT 0 10
INSERT 0 11
INSERT 0 13
INSERT 0 37
INSERT 0 69
INSERT 0 31
INSERT 0 73
spin_it_db=#
```

The values have been successfully added.

- Examples of simple queries:
    - Retrieve the names of all the albums produced by the musician who goes by the stage name ‘Zazu’.

```
spin_it_db=# SELECT stage_name, a_title
spin_it_db-# FROM MUSICIAN, ALBUM
spin_it_db-# WHERE musician.musician_id=album.musician_id AND stage_name='Zazu';
 stage_name |      a_title
------------+-------------------
 Zazu       | Sleepless Slumber
 Zazu       | Everyday
(2 rows)
```

    - What are the different types of albums?

```
spin_it_db=# SELECT DISTINCT(type)
spin_it_db-# FROM ALBUM;
     type
--------------
 EP
 Live Album
 Studio Album
(3 rows)
```

    - What are the names of the albums copyrighted before 2000?

```
spin_it_db=# SELECT a_title, copyright_date FROM ALBUM WHERE copyright_date < '2000-01-01';
   a_title   | copyright_date
-------------+----------------
 Poet's Soul | 1998-12-04
 Orbit       | 1990-10-22
(2 rows)
```

    - Retrieve all the songs (in descending order) performed by the musician Pearl Waters, along with their albums.

```
spin_it_db=# SELECT s_title, album_id
spin_it_db-# FROM PERFORMS, MUSICIAN, SONG
spin_it_db-# WHERE MUSICIAN.musician_id=PERFORMS.musician_id
spin_it_db-# AND SONG.song_id=PERFORMS.song_id
spin_it_db-# AND stage_name='Pearl Waters'
spin_it_db-# ORDER BY s_title DESC;
     s_title      | album_id
------------------+----------
 Tomorrow         | AL006
 Terrestrial Turf | AL008
 Serenity         | AL025
 Lucid            | AL003
 Flow             | AL025
 Dreaming         | AL003
 Charisma         | AL007
(7 rows)
```

- Examples of complex queries:
    - List the stage names of all the musicians who perform any of the songs Trixie performs.

```
spin_it_db=# SELECT DISTINCT stage_name
spin_it_db-# FROM PERFORMS, MUSICIAN
spin_it_db-# WHERE MUSICIAN.musician_id=PERFORMS.musician_id
spin_it_db-# AND song_id IN(
spin_it_db(# SELECT song_id
spin_it_db(# FROM PERFORMS
spin_it_db(# WHERE musician_id='SIMUS096')
spin_it_db-# EXCEPT
spin_it_db-# (SELECT stage_name
spin_it_db(# FROM MUSICIAN
spin_it_db(# WHERE musician_id='SIMUS096');
  stage_name
--------------
 Misty
 Stubot
 Benj Benson
 Greta
 Reggie Olsen
(5 rows)
```

- Name all the songs which have exactly two authors.

```
spin_it_db=# SELECT s_title
spin_it_db-# FROM SONG
spin_it_db-# WHERE song_id IN
spin_it_db-# (SELECT AUTHOR.song_id
spin_it_db(# FROM AUTHOR
spin_it_db(# GROUP BY AUTHOR.song_id
spin_it_db(# HAVING COUNT(*)=2
spin_it_db(# ORDER BY song_id ASC);
         s_title
---------------------------
 So Awake
 Lucid
 Tomorrow
 Emerald Sea
 Terrestrial Turf
 Charisma
 Charmed
 Beating The Box
 Shoutout
 The Charlatan's Charleston
 Out Of This World
 Flying High
 Almost There
 It's Coming Back
 Flow
(15 rows)
```

- Count the number of each instrument.

```
spin_it_db=# SELECT i_name, COUNT(*)
spin_it_db-# FROM INSTRUMENT
spin_it_db-# GROUP BY i_name;
  i_name   | count
-----------+-------
 Violin    |     2
 Guitar    |     3
 Saxophone |     2
 Keyboard  |     1
 Drum      |     1
 Flute     |     2
(6 rows)
```

- Find out the number of occupants in each house.

```
spin_it_db=# SELECT addr, COUNT(*)
spin_it_db-# FROM MUSICIAN
spin_it_db-# GROUP BY addr;
        addr          | count
----------------------+-------
 #232, Maple Drive    |     1
 #663, Sunset Avenue  |     3
 #518, Brook Lane     |     2
 #047, Fairview Road  |     1
 #312, Willow Street  |     3
(5 rows)
```

- Testing the trigger:

  Using the previous query, let's test out our trigger. The trigger makes sure that no more than three of our musicians live in the same home. If we try to insert a fourth musician, with an address which already has three occupants, it should automatically set the address value to NULL.

```
spin_it_db=# INSERT INTO MUSICIAN VALUES
spin_it_db-# ('SIMUS446', 'Jewel', 'Stacy', 'D', 'Greene', 28, '#663, Sunset Avenue');
INSERT 0 1
spin_it_db=# SELECT * FROM MUSICIAN;
 musician_id |  stage_name  |  fname   | minit |  lname   | age |        addr
-------------+--------------+----------+-------+----------+-----+---------------------
 SIMUS446    | Jewel        | Stacy    | D     | Greene   |  28 |
 SIMUS010    | Zazu         | Ryan     | S     | Aaron    |  24 | #663, Sunset Avenue
 SIMUS215    | Reggie Olsen | Kyle     | B     | Davis    |  32 | #312, Willow Street
 SIMUS003    | Luna Jade    | Natalie  | U     | Baker    |  22 | #663, Sunset Avenue
 SIMUS096    | Trixie       | Mae      | C     | Tudgeman |  34 | #232, Maple Drive
 SIMUS472    | Sammy J      | Samuel   |       | Johnson  |  43 | #518, Brook Lane
 SIMUS331    | Greta        | Greta    | G     | Smith    |  54 | #312, Willow Street
 SIMUS108    | Benj Benson  | Benjamin |       | Turner   |  26 | #047, Fairview Road
 SIMUS588    | Misty        | Natalie  |       | Vermont  |  38 | #518, Brook Lane
 SIMUS239    | Stubot       | Stuart   | P     | Abbott   |  33 | #663, Sunset Avenue
 SIMUS177    | Pearl Waters | Ava      | Q     | Waters   |  27 | #312, Willow Street
(11 rows)
```

We can see that '#663, Sunset Avenue' already had three occupants. When we try to add another, it sets the address of that row to NULL. Therefore, our trigger is working perfectly.

Hence, the database has been built.