Information Systems Management

Jean-François COUCHOT
couchot[at]femto-st[dot]fr

December 7, 2023

Contents

1 Introduction to Databases 2
- 1.1 What is a database . . . 2
  - 1.1.1 Data consistency . . . 2
  - 1.1.2 Models . . . 3
  - 1.1.3 Querying a database with SQL . . . 4

2 Data Query Language 5
- 2.1 The Chinook SQLite sample database . . . 5
- 2.2 Simple selection with `SELECT` . . . 6
- 2.3 Ordering tuples with `ORDER BY` . . . 6
- 2.4 Remove duplicate rows in the result set with `DISTINCT` . . . 7
- 2.5 Filtering with `WHERE` . . . 8
- 2.6 Getting linked data with `JOIN` . . . 9
  - 2.6.1 Coarse idea to merge tables . . . 10
  - 2.6.2 Linking with `INNER JOIN...ON` . . . 11
  - 2.6.3 `JOIN` on more than 2 tables . . . 11
  - 2.6.4 Auto `JOIN` . . . 12
- 2.7 Aggregating values of an attribute . . . 13
  - 2.7.1 Simplest case: aggregating values over all the results . . . 13
  - 2.7.2 Aggregating values by `GROUP`ing rows according to some attribute values . . . 14
  - 2.7.3 Conditions that only can be expressed with an aggregation function: `HAVING` . . . 15
- 2.8 Nested queries . . . 15
- 2.9 Set theoretic operators between queries: `UNION`, `INTERSECT`, `EXCEPT` . . . 16
  - 2.9.1 `UNION` or `INTERSECT`ion between two relations . . . 16
  - 2.9.2 `EXCEPT` between two relations . . . 16

3 Data Manipulation Language 18
- 3.1 `INSERT` rows into 1 table . . . 18
  - 3.1.1 Insert data explicitly defined by its `VALUES` . . . 18
  - 3.1.2 Inserting data from a `SELECT` . . . 19
- 3.2 `UPDATE` data of existing rows in 1 table . . . 19
  - 3.2.1 `UPDATE` one row . . . 20
  - 3.2.2 `UPDATE` many rows . . . 20
  - 3.2.3 `UPDATE` with parametric values . . . 20
- 3.3 `DELETE` rows from 1 table . . . 21

4 Data Definition Language 22
- 4.1 Data types . . . 22
- 4.2 `CREATE TABLE` . . . 22
- 4.3 `PRIMARY KEY` . . . 23
- 4.4 `FOREIGN KEY` . . . 24
  - 4.4.1 Motivating `FOREIGN KEY` with an example . . . 24
  - 4.4.2 `ON DELETE` and `ON UPDATE` actions . . . 25

Chapter 1

Introduction to Databases

This chapter is widely inspired from [Aud09, sql].

1.1 What is a database

It is hard to provide a sentence that exactly defines what a database is. At least, we could write:

DEFINITION 1.1 (DATABASE). A *database* is an organized set of information with a common objective.

A database is thus a structured and organised set, that allows to store a large set of data with the objective of easing the processing of adding, updating, searching for data... Of course, the core of this course is focused on digital database.

DEFINITION 1.2 (DIGITAL DATABASE). A *digital database* is a structured set of data stored on media accessible through a computer, which can be queried and updated by a community of users.

1.1.1 Data consistency

Creating a database aims at gathering data that are linked to each other in order to retrieve information using criteria, which is based on the information content. However if the content is not consistent (e.g. Couchot is sometimes written as Coucho), or if it is duplicated in many locations in the database, this may not be tractable. One can formalise this as data *inconsistency* and *redundancy* respectively.

DEFINITION 1.3 (DATA REDUNDANCY). Data redundancy occurs when the same piece of data is stored in two or more separate places

This data redundancy may lead to following problems:

- Wasted storage space.
- More difficult database update, database query.
- It may lead to Data Inconsistency.

Let us take an example of a cricket player table given in Table 1.1. For each player, the same information is provided twice: which teams does s/he belong to? The answer is given as an ID and as a name.

DEFINITION 1.4 (DATA INCONSISTENCY). Data *inconsistency* refers to a situation of keeping the same data in different formats in two different locations.

"New Zealand" and "New Zeland" are instances of data inconsistency. Normalization (seen later) helps to avoid redundancy and inconsistency.

Player Name	Player Age	Team Name	Team ID
Virat Kohli	32	India	1
Rohit Sharma	34	India	1
Ross Taylor	37	New Zealand	2
Shikhar Dhawan	35	India	1
Kane Williamson	30	New Zeland	2

Table 1.1: First example for cricket player table

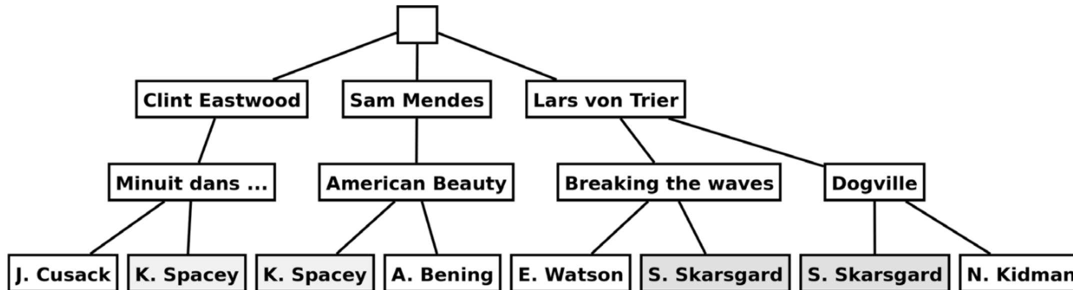

Figure 1.1: Hierarchy Model of a movie database

1.1.2 Models

1.1.2.1 Hierarchical Database

A hierarchical database is a form of database management system that links records in a tree structure so that each record has only one owner. Figure 1.1 gives an example of such model.

These models have been progressively discarded because they violate the constraints of consistency and non-redundancy in favor of the relational models seen below. However, these models can be interesting because of the way they can be processed quickly. We speak of NoSQL, for instance.

This type of database won't be treated in this course.

1.1.2.2 Relational Database (Codd)

The relational model (RM) for database management is an approach to managing data using a structure and language consistent with first-order predicate logic, described in 1969 by F. Codd. All data is represented in terms of tuples, grouped into relations. A database organized in terms of the relational model is a relational database. Important terms[1] are:

- Data *types* as used in a typical relational database. Might be the set of integers, the set of character strings, or the set of dates. . . respectively named as "int", "char", "date", ...
- *Attributes*, which are the term used in the theory for what is commonly referred to as a column name. An attribute name might be "name" or "age". Each attribute has a given type.
- *Tuples*, which are basically the same thing as a rows without any order between them;
- *Relations*. A relation is a table structure definition (a set of column definitions) along with the data appearing in that structure. The structure definition is the heading and the data appearing in it is the body, a set of rows.

Here are two fundamental rules a relational database has to possess.

- Duplicate rows: the same tuple cannot appear more than once in a relation.
- Duplicate column names: every attribute hast to be unique.

To obtain this, a process of database normalization has to be performed, which introduces furthermore the notion of *key*. For example, Table 1.5 is a relational database with

[1]Wikipedia, https://en.wikipedia.org/wiki/Relational_model

Id	Title	Director
1	Midnight in the Garden of Good and Evil	7
2	American Beauty	8
3	Breaking the waves	9
4	Dogville	9

Table 1.2: movie relation

Id	First Name	Last Name
1	John	Cusack
2	Kevin	Spacey
3	Annette	Bening
4	Emily	Watson
5	Stellan	Skarsgard
6	Ncole	Kidman
7	Clint	Eastwood
8	Sam	Mendes
9	Lars	Von Trier

Table 1.3: person relation

IdP	idM
1	1
2	1
2	2
3	2
4	3
5	3
5	4
6	4

Table 1.4: distribution relation

Table 1.5: A rational database of movies

- 3 relations (or 3 tables): Movie, Person, Actors
- All the attributes Id, IdP, IdM, Director possess the int type
- All the attributes Title, First Name, and Last name are stings.
- The keys are Id for movie relation, Id for person relation, and (IdP, IdM) for distribution relation.

1.1.3 Querying a database with SQL

The are many types of Simple Query Language (SQL) statements that are:

- Data Query Language (DQL) Statements: SELECT
- Data Manipulation Language (DML) Statements: INSERT, DELETE, UPDATE,...
- Data Definition Language (DDL) Statements: CREATE, ALTER, DROP, RENAME

This is the plan for the next three chapters.

Chapter 2

Data Query Language

In all this chapter, we consider the database to be created. We will just extract some data from the database.

First section presents the Chinook SQLite sample database we'll use in all this chapter. All the queries are based on the `SELECT` statement. This one will be presented in section 2.2. Ordering results and outputig only distinct results will be shown in sections 2.3 and 2.4 respectively. Section 2.5 introduces the `WHERE` statement that filters queries results. When queries deal with many tables, it is necessay to `JOIN` them, as presented in section 2.6.

Section 2.7 shwos how we can apply some aggregating function on the result values.

2.1 The Chinook SQLite sample database

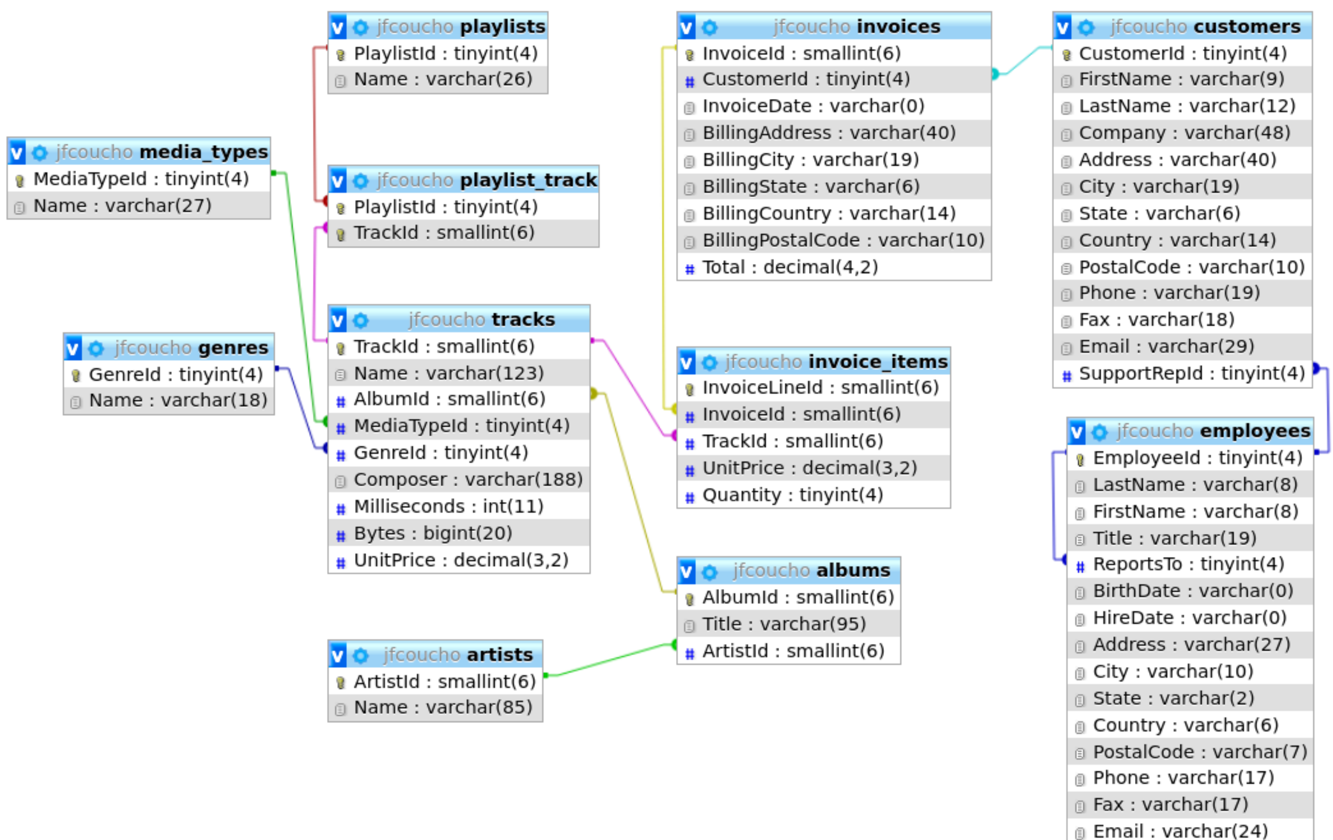

Figure 2.1: Entity-Relation of SQLite Sample Database from https://www.sqlitetutorial.net/sqlite-sample-database/

In all this chapter we'll consider the Chinook sample database of SQLite Tutorial [1], displayed at figure 2.1. This database contains 11 tables. Among them:

[1] https://www.sqlitetutorial.net/

- *artists*: data about artists. It is a simple table that contains only the artist identifier (`Artistid`) and name (`name`).
- *tracks*: data of songs. Each track belongs to one album
- *albums*: data about a list of tracks. Each album belongs to one artist and one artist may have multiple albums.
- *playlists* and *playlist_track*: each playlist contains a list of tracks. Each track may belong to multiple playlists. The relationship between the playlists table and tracks table is many-to-many.

2.2 Simple selection with SELECT

We fist explain how to use **SELECT** statement to query data from a single table.

In its (most) basic presentation, the **SELECT** statement is presented as follows:

```
1 SELECT column_list
2 FROM table;
```

SQLite evaluates this statement as follows:

1. First, the `FORM` clause is analyzed to is analysed to find out from which table the data should be extracted. Notice that there can be several tables in the `FORM` clause (see Section 2.6).
2. Second, the **SELECT** clause is analyzed to find which column name or a list of comma-separated column names to display.
3. The semicolon (;) terminates the statement.

Example 2.1 (First queries[2]).

The following query

```
1 SELECT Title
2 from albums;
```

would lead to

Title
For Those About To Rock We Salute You
Balls to the Wall
Restless and Wild
Let There Be Rock
Big Ones

Exercise 2.1. 1. *What would be the result of the following query?*

```
1 SELECT
2   trackid,
3   name,
4   composer,
5   unitprice
6 FROM
7   tracks;
```

2. *Display all the available music genres.*
3. *Display the first name, the last name and the id of all costumers.*

Next section shows how to order results output by SQLite.

2.3 Ordering tuples with ORDER BY

Output rows may or may not be in the order that they were inserted. These are data in an unspecified order. To sort the result set, it is sufficient to add the **ORDER BY** clause to the **SELECT** after **FROM**, as follows:

```
SELECT
   column_1, column_2, ...
FROM
   table
ORDER BY
    column_1 ASC,
    column_2 DESC,
    column_3;
```

where the sorting statement have to be placed after the **ORDER BY** clause. Obviously, **ASC** and **DESC** keywords mean ascending and descending order respectively. If none of the **ASC** or **DESC** parameters are specified (as here with `column_3`), the **ASC** order is executed by default Notice that the **ORDER BY** clause sorts rows using columns from left to right (here, first `column_1`, next `column_2`,...).

Example 2.2 (Ordered queries). *Here are some queries easy to understand:*

```
SELECT FirstName FROM customers ORDER BY FirstName;
SELECT albumid, name, milliseconds FROM tracks ORDER BY albumid ASC;
SELECT albumid, name, milliseconds FROM tracks ORDER BY albumid ASC, milliseconds DESC;
```

Exercise 2.2.

1. *From table `invoices`, display the `InvoiceId`, the `InvoiceDate`, and the `CustomerId`.*
2. *Sort results by an increasing order applied to the `InvoiceDate` and the `CustomerId`.*
3. *Display the names, first name and date of birth of the employees, from the oldest to the youngest*

2.4 Remove duplicate rows in the result set with DISTINCT

SELECT DISTINCT ... instead of **SELECT** removes duplicates in the output results.

- If only one attribute name is given after **SELECT DISTINCT**, duplicated values of this attribute are not displayed.
- If many attributes `att1,att2,...,attn`, are given duplicated values of `(v1,v2,...,vn)` are not shown.

Example 2.3 (With and without DISTINCT).

*Without the **DISTINCT** keyword.*

```
SELECT city FROM customers
ORDER BY city;
```

City
Amsterdam
Bangalore
Berlin
Berlin
Bordeaux
⋮
Lisbon
London
London
⋮

*where `Berlin`, `London`, and `Mountain` **View** are duplicated.*

*With the **DISTINCT** keyword.*

```
SELECT DISTINCT city FROM customers
ORDER BY city;
```

City
Amsterdam
Bangalore
Berlin
Bordeaux
⋮
Lisbon
London
⋮

without any duplication.

Exercise 2.3.

1. *Display audio track names and notice that some are duplicates.*
2. *Filter these results to show only single tracks.*

2.5 Filtering with WHERE

The **WHERE** clause only displays the results that satisfy a given condition. It is used as follows:

```
SELECT
  column_list
FROM
  table
WHERE
  search_condition;
```

Example 2.4 (Displaying tracks, but not all of them). *1. First, we display some track data from tracks whose album ID is 1]*

```
SELECT
  Name,
  Milliseconds,
  Albumid
FROM
  tracks
WHERE
  Albumid = 1;
```

Name	Milliseconds	AlbumId
For Those About To Rock (We Salute You)	343719	1
Put The Finger On You	205662	1
Let's Get It Up	233926	1
Inject The Venom	210834	1
Snowballed	203102	1
Evil Walks	263497	1
C.O.D.	199836	1
Breaking The Rules	263288	1
Night Of The Long Knives	205688	1
Spellbound	270863	1

2. *we furthermore restrict to songs that have a duration longer than 4 minutes.*

```
SELECT
  Name,
  Milliseconds,
  Albumid
FROM
  tracks
WHERE
  Albumid = 1
  AND milliseconds > 240000;
```

Name	Milliseconds	AlbumId
For Those About To Rock (We Salute You)	343719	1
Evil Walks	263497	1
Breaking The Rules	263288	1
Spellbound	270863	1

Between the 2 conditions, the **AND** *keyword has been added to express both conditions have to be* **True**.

More generally, conditions can be comparisons as given in Table 2.1.
More details about syntax of **BETWEEN**[3], **IN**[4], **LIKE**[5], and **IS NULL** [6] are given in the associated footnotes.

Example 2.5 (BETWEEN, IN, LIKE, and IS NULL operators). *Let us consider the following queries:*

- `SELECT * FROM invoices WHERE InvoiceDate BETWEEN '2010-01-01' AND '2010-01-31';` *it displays invoices whose date takes place between the First January of 2010 and the 31st of the same month.*
- `SELECT * FROM invoices WHERE BillingCountry IN ('Germany','France');` *it displays invoices whose billing country is either Germany or France.*
- `SELECT * FROM invoices WHERE BillingCountry = "Germany" AND BillingPostalCode LIKE "1____";` *it displays invoices whose billing country is in Germany and whose postal code has 5 digits and starts with 1.*

[3] https://www.sqlitetutorial.net/sqlite-between/
[4] https://www.sqlitetutorial.net/sqlite-in/
[5] https://www.sqlitetutorial.net/sqlite-like/
[6] https://www.sqlitetutorial.net/sqlite-is-null/

Operator	Meaning
`=`	Equal to
`<>` or `!=`	Not equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
BETWEEN X **AND** Y	returns **True** if a value is greater than or equal to X **AND** lower than or equal to Y.
IN	returns **True** if a value is in a list of values.
LIKE	returns **True** if a value matches a pattern % (any sequence of 0/many characters), _ (any single character).
IS NULL	returns **True** the value is **NULL**

Table 2.1: Comparison operators

- *`SELECT FirstName, LastName FROM customers WHERE Company IS NULL;` it displays first name and last name of customers who do not work in a company.*

Logical operators allow to combine logical expressions. A logical operator returns 1, 0, or a NULL value. Table 2.2 illustrates the basic logical operators.

Operator	Meaning
AND	returns **True** if both expressions are **True** and **False** otherwise
OR	returns **True** if either expression is True
NOT	reverses the value of other operators such as **NOT IN**, **NOT BETWEEN**, **IS NOT NULL** etc.

Table 2.2: Basic logical operators

Exercise 2.4. *Provide a query only showing:*

1. *a list of unique billing countries from the Invoice table.*
2. *first name and last name of customers from Brazil.*
3. *first name and last name of customers who are not in the USA.*
4. *first name and last name of customers who are working in a company.*
5. *the Employees who are Sales Agents.*
6. *the InvoiceId, the InvoiceDate and the Total of invoices were date is in 2009 or in 2011.*

2.6 Getting linked data with JOIN

In relational databases and due to normalization process, data is often distributed in many related tables. Links between tables are formalized thanks to **PRIMARY KEY**s and **FOREIGN KEY**s. Fore instance, as seen in Figure 2.1:

- The `media_types` relation has the `MediaTypeId` **KEY**. The `tracks` relation has the `MediaTypeid` **FOREIGN KEY** which is linked to the former. It precises which mediatype has this track.
- The `artists` relation has the `ArtistId` **KEY**. The `albums` relation has the `ArtistId` **FOREIGN KEY** which is linked to the former. Each album has been created by one artist.

2.6.1 Coarse idea to merge tables

Suppose for instance two tables `albums` and `artist` as given in figure 2.1. Atributes of `albums` are `AlbumId`, `Title`, and `ArtistId` whereas attributes of `artists` are `ArtistId` and `Name`. The attributes `ArtistId` is a reference to `ArtistId` of table `artists`.

Data is split into two tables to avoid inconsistency Table 2.3 and 2.4 give excerpts of these two tables. In such a model, the artist Alanis Morissette for instance will be stored once and thus always written in the same manner.

AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
2	Balls to the Wall	2
3	Restless and Wild	2
4	Let There Be Rock	1
5	Big Ones	3

Table 2.3: Excerpt of table `albums`

ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith
4	Alanis Morissette

Table 2.4: Excerpt of table `artists`

AlbumId	Title	Name
1	For Those About To Rock We Salute You	AC/DC
1	For Those About To Rock We Salute You	Accept
	⋮	
1	For Those About To Rock We Salute You	Philip Glass Ensemble
2	Balls to the Wall	AC/DC
2	Balls to the Wall	Accept
	⋮	
2	Balls to the Wall	Philip Glass Ensemble

Table 2.5: Result of query "**SELECT** `AlbumId, Title, Name` **FROM** `albums, artists`"

AlbumId	Title	Name
1	For Those About To Rock We Salute You	AC/DC
2	Balls to the Wall	Accept
3	Restless and Wild	Accept
	⋮	
347	Koyaanisqatsi (Soundtrack from the Motion Picture)	Philip Glass Ensemble

Table 2.6: Result of query "**SELECT** `AlbumId, Title, Name` **FROM** `albums` **INNER JOIN** `artists` **ON** `albums.ArtistId = artists.ArtistId`"

Suppose now we want to display for each album its ID, its title and its contributor artist. A coarse idea would be to execute such query:

```
1 SELECT
2   AlbumId, Title, Name
3 FROM
4   albums, artists
```

whose result is given in Table 2.5. In this query, all the artists are associated with each album. Indeed we did not force the artist who created the album (`albums.ArtistId`) to be the same as the one whose name is displayed (`artists.ArtistId`). Mathematically speaking, the cartesian product between the 2 sets `albums` and `artists` is computed. Next section shows how to constrain the join between tables.

2.6.2 Linking with INNER JOIN...ON

```
SELECT
  Title,
  Name
FROM
  albums INNER JOIN artists ON  albums.ArtistId = artists.ArtistId
```

or equivalently

```
SELECT
  Title,
  Name
FROM
  albums, artists
WHERE
  albums.ArtistId = artists.ArtistId
```

whose result is given in Table 2.6. For each row in the `albums` table, the **INNER JOIN** clause compares the value of the `ArtistId` column with the value of the `ArtistId` column in the `artists` table. If both are equal, it combines data from `albums.AlbumId`, `albums.Title`, and `artists.Name` columns and includes this row in the result set.

Exercise 2.5. *Provide a query only showing:*

1. *The track name and its genre when unit price is greater than 1.*
2. *All the album names from artist "Led Zeppelin".*

It is possible to query data from multiple tables by using **INNER JOIN** clause, which combines columns from correlated tables.

2.6.3 JOIN on more than 2 tables

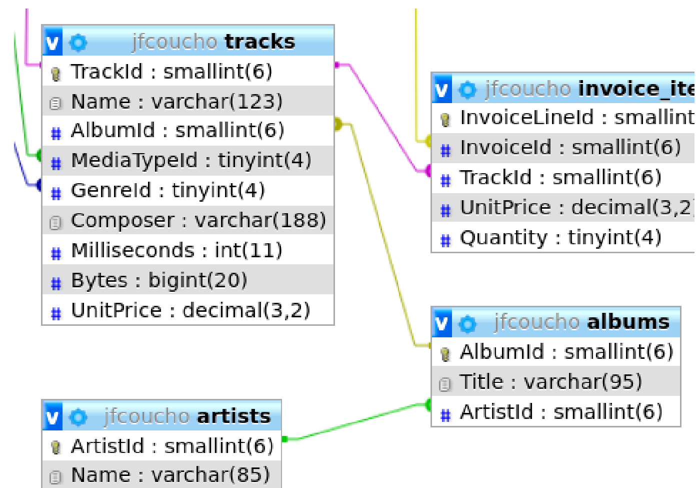

Figure 2.2: JOIN on 3 tables

Figure 2.2 shows these tables `tracks`, `albums`, `artists` and their links. One track belongs to one album and one album have many tracks. The `tracks` table is associated with the `albums` table via `AlbumId` column. One album has been created exactly by one artist and one artist may have zero, one or many albums. The `albums` table is linked to the `artists` table via `ArtistId` column.

To display each track (its id and its name), in which album it appears and its associated artist, it is sufficient to use two **INNER JOIN** clauses in the **SELECT** statement as follows:

```sql
SELECT
    TrackId,
    tracks.Name AS 'Track Name',
    albums.Title AS 'Album Title',
    artists.Name AS 'Artist Name'
FROM
    tracks
    INNER JOIN albums ON tracks.AlbumId = albums.AlbumId
    INNER JOIN artists ON albums.ArtistId = artists.ArtistId;
```

Exercise 2.6. *Provide a query only showing:*

1. *the Invoices of customers who are from Brazil. The resultant table should show the customer's full name, Invoice ID, Date of the invoice and billing country.*
2. *the track name with each invoice line item.*
3. *all the Tracks. The resulting table should include the Album name, Media type and Genre.*
4. *all the genre names from the artist Led Zepplin.*
5. *all the customers (names and ID) who paid for one track created by Iron Maiden at least .*
6. *all the artist names who created a track that appears in one playlist, at least.*

2.6.4 Auto `JOIN`

Auto join make it easy to detect which values are duplicated. It is sufficient

1. to join the table (left) on itself (right) according to the attribute whose duplicated values are being searched for and
2. to impose that the identifier of the left table is greater than that of the right table

Example 2.6 (Selecting tracks with duplicated title name).

1. *We first join the `tracks` table with itself on the attribute `Name` as follows:*

```sql
SELECT L_tracks.TrackId,
         L_tracks.Name,
         R_tracks.TrackId,
         R_tracks.Name
FROM tracks AS L_tracks
INNER JOIN tracks AS R_tracks
    ON L_tracks.Name = R_tracks.Name
ORDER BY  L_tracks.Name
```

whose results are

TrackId	Name	TrackId	Name
3027	"40"	3027	"40"
2918	"?"	2918	"?"
⋮			
1221	2 Minutes To Midnight	1221	2 Minutes To Midnight
1221	2 Minutes To Midnight	1289	2 Minutes To Midnight
1221	2 Minutes To Midnight	1319	2 Minutes To Midnight
⋮			
1289	2 Minutes To Midnight	1221	2 Minutes To Midnight
1289	2 Minutes To Midnight	1319	2 Minutes To Midnight

Each track is linked with itself. But some track are linked with others. For instance, track (1221, 2 Minutes To Midnight) is linked to track (1289,2 Minutes To Midnight).

2. *Let us furthermore impose that `L_tracks.TrackId > R_tracks.TrackId` as follows:*

```
SELECT L_tracks.TrackId,
         L_tracks.Name,
         R_tracks.TrackId,
         R_tracks.Name
FROM tracks AS L_tracks
INNER JOIN tracks AS R_tracks
    ON L_tracks.Name = R_tracks.Name
WHERE L_tracks.TrackId > R_tracks.TrackId
ORDER BY  L_tracks.Name
```

This contraint will remove all the rows where Ids are equal (the first row, the second) which are not realy duplicated values.

3. *If we just want to keep the duplicated track names, it is thus sufficient to execute:*

```
SELECT DISTINCT L_tracks.Name
FROM tracks AS L_tracks
INNER JOIN tracks AS R_tracks
    ON L_tracks.Name = R_tracks.Name
WHERE L_tracks.TrackId > R_tracks.TrackId
ORDER BY  L_tracks.Name
```

Exercise 2.7 (Direct applications of auto-join). *Answer to the following questions thanks to a* **SQL** *query.*

1. *Wich city is cited at least twice in the customers table?*

2. *Which genre appears at least twice in the tracks table?*

2.7 Aggregating values of an attribute

Aggregate functions operate on a set of rows and return a single result. They are:

- **MIN**(), **MAX**(): returns the minimum and the maximum value in a group resp.
- **SUM**(), **AVG**(): returns the sum of values and the average value of a group resp.
- **COUNT**(): returns the number of rows that match a specified condition.

All these functions are applicable on an attribute (a column), not on a row!
We first show how they can be used on the whole result set.

2.7.1 Simplest case: aggregating values over all the results

For all these aggregate functions the syntax is

```
SELECT
  AGGREGATE_FUNCTION([DISTINCT] attribute_1),
  attribute_2, ...
FROM
  table1, table2
  ...
```

Example 2.7. 1. *Display the name, and its duration of the audio track with the minimal duration.*

```
SELECT Name, MIN(Milliseconds) FROM tracks
```

2. *Display the average duration (in minutes) of the entire set of audio tracks.*

```
SELECT AVG(Milliseconds)/60/1000 AS 'average duration' FROM tracks
```

3. *display the whole duration (in ms) of the album "Sozinho Remix Ao Vivo"*

```
SELECT SUM(Milliseconds)
FROM tracks NATURAL JOIN albums
WHERE title = 'Sozinho Remix Ao Vivo';
```

The following focusses on counting rows.

- `COUNT(expression)`: all non-`NULL` values including duplicates; this the default value;
- `COUNT(DISTINCT expression)`: only unique and non-null values are counted;
- `COUNT(*)`: returns the number of values, including NULL and duplicates.

Example 2.8 (Queries with `COUNT`).

```
SELECT COUNT(*) FROM invoices;
--number of rows from the invoices table

SELECT COUNT(*) FROM tracks WHERE albumId = 10;
--number of tracks whose albumId is 10:

SELECT COUNT(DISTINCT BillingState) FROM invoices;
--number of distinct states in invoices table
```

Exercise 2.8 (Using aggregate functions). *Provide queries that*

1. *displays the smallest track (in term of bytes).*
2. *displays the number of items for Invoice ID 37.*
3. *express the total duration (in ms) of all tracks produced by Led Zeppelin.*
4. *displays how many customers purchass at least one track.*
5. *displays how many customers did not fill in the state attribute.*

2.7.2 Aggregating values by `GROUP`ing rows according to some attribute values

Sometimes it is useful to make calculations by grouping rows that have the same value for some attributes. For instance, for each album you can find out the duration of the longest track. This would be:

```
SELECT AlbumId, MAX(Milliseconds) AS 'duration of lt' FROM tracks
GROUP BY AlbumId;
```

The examples below show how to aggregate results grouped by the values of some attributes.

Example 2.9.

```
SELECT albumid, COUNT(albumid)  AS NB_track FROM tracks GROUP BY albumid;
  -- displays all the albums and the number of tracks in each album:

SELECT albums.albumId, albums.Title , SUM(Milliseconds)/1000/60 AS duration
FROM tracks,albums WHERE tracks.albumId = albums.albumId GROUP BY tracks.albumId ;
  -- displays the duration of all the albums in minutes

SELECT invoiceId, SUM(tracks.unitPrice * quantity) AS invoice_total_price
FROM invoice_items, tracks
WHERE invoice_items.trackId = tracks.trackId GROUP BY invoiceID;
  -- displays the total price of each invoice

```

Exercise 2.9. *Provide a query showing:*

1. *the number of line items for each invoice*
2. *the number of invoices per billing country.*
3. *the number of times each track appears in invoices, where both tracksName and trackId are displayed*
4. *Total sales per billing country. Which country spents the most?*
5. *the number of tracks per genre. The genre name should be displayed instead of its ID.*
6. *the number of existing tracks for each media type.*
7. *the number of tracks that have been sold by a particular artist.*

2.7.3 Conditions that only can be expressed with an aggregation function: `HAVING`

One may want to express conditions using aggregation functions. For example "(albums with) the number of tracks greater than 10" could be written as "`... COUNT(trackId) > 10`". These conditions are not allowed with the `WHERE` clause because they concern a group. We use the `HAVING` clause which must follow a `GROUP BY`.

The following example proposes combination of the `SUM` aggregation function with `GROUP BY` and `HAVING` clauses. This latter is furthermore based on an alias introduced by `AS`.

Example 2.10.

```
1
2   SELECT albums.albumId,
3          albums.Title ,
4          SUM(Milliseconds)/1000/60 AS 'duration'
5  FROM tracks NATURAL JOIN albums
6  GROUP BY  tracks.albumId
7  HAVING duration > 60
8  ORDER BY  duration DESC
```

This query displays all the albums where the whole duration is greater than 1 hour.

Exercise 2.10. *Provide a query showing:*

1. *the albums (albumID, album title) whose number of tracks is greater than 13.*
2. *the tracks (name) whose name appears more than 1 time .*
3. *the artists (id and name) whose have at least 3 genres.*

2.8 Nested queries

A nested `SELECT` is a query within a query: there is a `SELECT` statement within the main `SELECT`.

Example 2.11 (Tracks whose duration is more than twice the average duration of the tracks). *It can be done in two steps:*

1. *What follows computes the average duration of all the tracks:*

   ```
   1  SELECT AVG(Milliseconds) FROM tracks;
   ```

   which returns 393599.2121039109

2. *It remains to diplay tracks whose duration is more than twice this number*

   ```
   1  SELECT trackID, name FROM tracks WHERE Milliseconds > 2*393599.2121039109
   ```

However this task can be solved in one step with a nested query:

```sql
SELECT trackID,
       name
FROM tracks
WHERE Milliseconds > 2*
    (SELECT AVG(Milliseconds)
    FROM tracks)
```

The subquery returns a single value. With this average duration returned by the nested query, the outer query can select tracks who satisfy the duration condition.

When the nested query is independant, it is interpreted first and results are computed. Next the outer is interpreted using these results.

Exercise 2.11 (Nested queries).

1. *List of artists who created more than 10 tracks, and thus, the distinct genres for these artists*
2. *List of tracks which appear at least twice in the* `invoice_items` *table. Thus display the average duration of these tracks*
3. *List of artists whose number of track purchased is greater than 40. What are the most present musical genres among the tracks created by these artists?*
4. *List of users (ID, first name, last name) who have made at least one invoice whose total amount exceeds the average amount of an invoice by 20 %.*

2.9 Set theoretic operators between queries: `UNION`, `INTERSECT`, `EXCEPT`

2.9.1 `UNION` or `INTERSECT`ion between two relations

It is sometimes required to combine similar data from multiple relations or multiple database into a complete result set. For instance, let us consider "playlists that contain only 1 musical genre or 1 artist " It can be easyly translated into

```sql
SELECT playlist_track.playlistID
FROM playlist_track INNER JOIN tracks ON playlist_track.trackID = tracks.trackID
INNER JOIN genres ON tracks.genreID = genres.genreID
GROUP BY  playlistID
HAVING COUNT(DISTINCT genres.genreID) =1
UNION
SELECT playlist_track.playlistID
FROM playlist_track INNER JOIN tracks ON playlist_track.trackID = tracks.trackID
INNER JOIN albums ON tracks.albumID = albums.albumID
GROUP BY  playlistID
HAVING COUNT(DISTINCT albums.albumID) =1
```

The `INTERSECT` statement produces the intersection between the two relations.

2.9.2 `EXCEPT` between two relations

SQLite EXCEPT operator compares the result sets of two queries and returns distinct rows from the left query that are not output by the right query.

For instance, let us write query displaying "artists who never produce rock music".

```sql
SELECT albums.artistID FROM albums
EXCEPT
SELECT albums.artistID
FROM albums INNER JOIN tracks ON albums.albumID = tracks.albumID
INNER JOIN genres ON tracks.genreID = genres.genreID
WHERE genres.name ="Rock"
```

take care, it is not equal to

```
SELECT albums.artistID
FROM albums INNER JOIN tracks ON albums.albumID = tracks.albumID
INNER JOIN genres ON tracks.genreID = genres.genreID
WHERE genres.name !="Rock"
```

For instance Artist 8 is returned by the latter, not by the former.

Exercise 2.12 (Playing with `UNION`, `INTERSECT`, `EXCEPT`). *Provide queries with:*

1. *artists (ID and name) who created albums but who have not sold any tracks.*
2. *artists (ID and name) who have sold 1 track, at most.*
3. *playlists that do not contain any Reggae tracks.*

Chapter 3

Data Manipulation Language

In all this chapter, we consider that the database has already been created. We will only modify the data of one table at a time. Modifications are **INSERT** (Section 3.1), **UPDATE** (Section 3.2), and **DELETE** (Section 3.3).

3.1 INSERT rows into 1 table

The **INSERT** statement allows to insert rows into a table either row by row where each row is defined by its `VALUE` or from data provided by a **SELECT** statement.

3.1.1 Insert data explicitly defined by its VALUES

Inserting multiple rows into 1 table is achieved by the following form of the **INSERT** statement

```
1  INSERT INTO Table (Att1, Att2,...) VALUES (x1, x2,...), (y1, y2,...),...
```

where

- `(Att1, Att2,...)` is a sub-list of all attributes of **Table** we want to precise in the new records `(x1, x2,...)`,
- `(x1, x2,...), (y1, y2,...)` is the list of tuples of values such that:
  - the value order is the same than the attribute order (`xn` ↔ `Attn`);
  - each value `xn`: has to belong to the domain of `Attn`;
  - in case of multiple insertion: use comma-separated tuples.

Example 3.1 (Inserting 3 artists and displaying modifications.). *What follows inserts 3 rows into the* `artists` *table.*

```
1  INSERT INTO artists (name)
2  VALUES
3    ("Buddy Rich"),
4    ("Candido"),
5    ("Charlie Byrd");
```

Result can thus be verified thanks to:

```
1  SELECT
2          ArtistId,
3          Name
4  FROM
5          artists
6  ORDER BY
7          ArtistId DESC
8  LIMIT 5;
```

Exercise 3.1 (A recent Red Hot Chili Peppers album). *"Return of the Dream Canteen" is a recent album of the group Red Hot Chili Peppers*

1. *How to verify whether this group is already in this database or not?*

2. *How to verify whether this album is already in this database or not?*

3. *Insert this album and the following 3 songs, each of them has "Rock" genre and "MPEG audio file" as mediaType, and 0.99$ as unitprice:*

   (a) *Tippa My Tongue, 04:20.*

   (b) *Peace and LoveE, 04:03.*

   (c) *Reach Out, 04:11.*

In the previous exercise, some values have not been provided (Composer and Bytes for instance). In this case, the default **NULL** value is inserted into the database for those attributes.

Exercise 3.2 (Problem with Instertion). *We want to add the artist named "Bud Powell". To achieve this, the following query is proposed:*

```
1   INSERT INTO artists (artistID,name) VALUES (1,'Bud Powell');
```

1. *Implement this query and verify it generates an error.*

2. *Explain this error and correct the query.*

3.1.2 Inserting data from a SELECT

The second form of the INSERT statement contains a SELECT statement instead of a VALUES clause. A new entry is inserted into the table for each row of data returned by executing the SELECT statement. If a attribute-list is specified, the number of attributes in the result of the SELECT must be the same as the number of items in the attribute-list. Otherwise, if no attribute-list is specified, the number of attributes in the result of the SELECT must be the same as the number of attributes in the table. The syntax is as follows:

```
1  INSERT INTO Table (Att1, Att2, ...)  SELECT Att1, Att2, ...
```

Example 3.2 (Inserting a new playlist from another one). *The following query inserts a new playlist (named 'Your New PlayList') containing 50% of the tracks from the "Brazilian Music" existing playlist in the database*

```
1   -- Step 1: Create a new playlist
2   INSERT INTO playlists (Name) VALUES ('Your New Playlist');
3
4   -- Step 2: Insert 50% of tracks FROM 'Brazilian Music' into the new playlist
5   INSERT INTO playlist_track (PlaylistId, TrackId)
6   SELECT
7       (SELECT PlaylistId FROM playlists
8       WHERE Name = 'Your New Playlist') AS TheNewPlaylistId, playlist_track.TrackId
9   FROM playlist_track INNER JOIN playlists
10      ON playlists.PlaylistId = playlist_track.PlaylistId
11  WHERE Name = 'Brazilian Music'
12  ORDER BY  RANDOM() LIMIT
13      (SELECT CAST(COUNT(TrackId)*0.5 AS INT) FROM playlist_track INNER JOIN playlists
14          ON playlists.PlaylistId = playlist_track.PlaylistId
15          WHERE Name = 'Brazilian Music');
```

Exercise 3.3 (Inserting with SELECT). *Create a new invoice for Leonie Köhler customer with all the items of her last invoice.*

3.2 UPDATE data of existing rows in 1 table

Updating rows values of 1 table is achieved by the following form of the **UPDATE** statement

```
UPDATE Table  SET Att1 = Val1, Att2 = Val2, ... WHERE Condition
```

where

- all rows of `Table` which verify the `Condition` of the `WHERE` statement will be updated
- the value of each attribute `Att1`, `Att2` will be set to `Val1`, `Val2`.

3.2.1 UPDATE one row

When the **WHERE** condition is a equality between a primary key and a defined value, only one row is returned. It is this row that is modified.

Example 3.3 (Changing Address of Margaret Park). *The following query change location data (address, city, state, postal code of the employee Margaret Park.*

```
UPDATE employees SET city = 'Toronto', address= '14 Langton Ave', state = 'ON', postalcode = 'M4N 3C4'
WHERE employeeid =
    (SELECT employeeid
    FROM employees
    WHERE LastName='Park'
            AND FirstName='Margaret')
```

3.2.2 UPDATE many rows

When several rows verify the **WHERE** condition, all these rows will be **UPDATE**d by this change.

Example 3.4 (Modifying the genre of all Red Hot Chili Peppers tracks). *Sometimes, tracks of Red Hot Chili Peppers are misclassified as "Alternative & Punk" and sometimes as "Rock". The following query corrects this erroneous classification.*

```
UPDATE tracks SET genreID =
    (SELECT GenreId
    FROM genres
    WHERE Name LIKE "Rock")
WHERE AlbumId IN
    (SELECT AlbumId FROM albums INNER JOIN artists ON
    albums.ArtistId = artists.ArtistId
    WHERE artists.Name LIKE 'Red%');
```

which can be verified with

```
SELECT DISTINCT genres.Name,
         artists.Name
         FROM genres INNER JOIN tracks ON genres.GenreId = tracks.genreId
         INNER JOIN albums ON tracks.AlbumId = albums.AlbumId
         INNER JOIN artists ON albums.ArtistId = artists .ArtistId
         WHERE artists.Name LIKE 'Red%';
```

3.2.3 UPDATE with parametric values

It is possible to **UPDATE** attributes with values that depend on other attributes values

For instance, it is possible to increase the track unitprice of 10% for tracks whose unitprice is lower than 1, and of 5% only otherwise.

```
UPDATE tracks SET UnitPrice =
    CASE
    WHEN UnitPrice < 1 THEN
    1.1*UnitPrice
    ELSE 1.05*UnitPrice END
```

Exercise 3.4 (Update, Update, Update). *Provide the following queries, which*

1. *set the duration of track "Balls to the Wall" to 342500ms.*
2. *Set the price to 1$ of all Alanis Morissette tracks.*
3. *Set the last name in uppercase (`UPPER` function) in the customers table*

3.3 `DELETE` rows from 1 table

`DELETE` statement removes from a table *all rows verifying a condition* specified in a `WHERE` statement. It is defined as follows:

```
DELETE FROM Table WHERE Condition
```

Example 3.5 (Deleting all artists whose name starts with A). *This is achieved with the following code.*

```
DELETE FROM artists WHERE Name LIKE'A%'
```

Example 3.6 (Deleting all artists whose tracks does no appear in any invoice). *First of all the nested query that select all artists whose tracks does no appear in any invoice, and next, deletion.*

```
DELETE
FROM artists
WHERE ArtistId IN
    (SELECT ArtistId
    FROM artists EXCEPT SELECT  albums.ArtistId
    FROM albums INNER JOIN tracks ON albums.AlbumId = tracks.AlbumId
    INNER JOIN invoice_items ON tracks.TrackId = invoice_items.TrackId)
```

Notice that if no `WHERE` statement is provided, all rows from the table are removed. In the following query, all rows from `artists` table are removed.

```
DELETE FROM invoices
```

Exercise 3.5 (Removing rows).

1. *Delete all rows from table `playlist_track` and from table `playlists`.*
2. *Delete all genres that are never associated with a track.*
3. *Delete all tracks that do not appear in any invoice.*

Notice that in the SQLiteTutorial example, it is possible to delete all the artists even if references to these artists are not removed: the `albums` table always contains references to all `ArtistIds`. This behavior is problematic since it leads to inconsistency. Next chapter shows how to avoid this in the database definition.

Chapter 4

Data Definition Language

4.1 Data types

SQLite provides 4 data classes that are:

- `INTEGER` are positive or negative whole numbers. It can be refined into `BIGINT`(8 bytes), `INT` (4 bytes), `MEDIUMINT` (3 bytes), `SMALLINT` (2 bytes), and `TINYINT` (1 byte).
- `REAL` are numbers with decimal values. It can be refined into `DOUBLE` (8 bytes), `FLOAT(p)` (4 bytes if `p` <24),
- `TEXT` are unlimited length strings, stored using the database encoding (UTF-8 for instance). It can be refined into `CHARACTER(p)` (at most `p` characters padded with " ", possibly) `VARCHAR(p)` (at most `p`, no padding), `NATIVE CHARACTER(p)` (at most `p` unicode characters padded with " ", possibly)
- `BLOB` stands for a binary large object (image, sound) that can store any kind of data.

If the main objective is not efficiency at all, one can restrict oneself to these 4 classes.

Exercise 4.1 (A type to each data).
What type can be associated with each of the following data:

1. *A french cell phone number starting with 06/07, with digits only?*
2. *An email address?*
3. *A price (everyday objects)?*
4. *A date of birth?*
5. *A last name?*
6. *A sentence?*
7. *A duration (of a few minutes) in milliseconds?*

4.2 `CREATE TABLE`

The **`CREATE TABLE`** statement do the task of creating a new table in a SQL database. Its syntax is as follows:

```
1 CREATE TABLE table_name (
2 column_1 data_type PRIMARY KEY,
3 column_2 data_type NOT NULL,
4 column_3 data_type DEFAULT 0,
5 table_constraints
6 ) [WITHOUT ROWID];
```

where:

- **`table_name`** specifies the name of the table currently being created;

- `column_1`, `column_2`, `column_3` specifies the attribute (column) names. Each attribute has a name, data type, possibly a constraint among:
  - **PRIMARY KEY**: this attribute is the primary table key;
  - **UNIQUE**: each row must contain a unique combination of values in all the columns identified by the **UNIQUE** constraint;
  - **NOT NULL**: the attribute may not contain a **NULL** value
  - **CHECK** attribute constraint: each time a new row is inserted into the table or an existing row is updated, the expression associated with each **CHECK** constraint is evaluated;
- `table_constraints` specifies other constraints, like **PRIMARY KEY** defined with many column, **FOREIGN KEY**...
- **WITHOUT** ROWID option: specific to SQLite. By default (i.e. when **WITHOUT** ROWID is not present), SQLite creates an implicit attribute referred to as the rowid. The rowid column stores a 64-bit signed integer key that uniquely identifies the row inside the table. By specifying **WITHOUT** ROWID, this action is avoided.

Exercise 4.2 (First Creates).

1. *Create the table* `artists`
2. *Create the table* `albums`

4.3 PRIMARY KEY

Each table in SQLite must have one **PRIMARY KEY** which is either supported by a single attribute or a set of attributes.

If **PRIMARY KEY** is added to an attribute definition (as in the above example), then the primary key for the table consists of that single attribute. Otherwise, if a **PRIMARY KEY** clause is specified as a table-constraint, then the primary key of the table consists of the list of attributes specified as part of the **PRIMARY KEY** clause.

Example 4.1 (Table `playlist_track` with a pair as PRIMARY KEY). .

The following script defines table `playlist_track` *represented in figure 4.1*

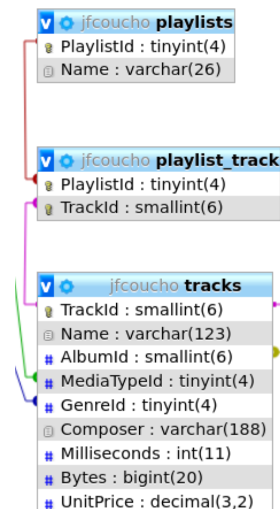

Figure 4.1: Playlist_track between Playlists and Tracks

```
1  CREATE TABLE playlist_track (
2    PlaylistId INTEGER,
3    TrackId INTEGER,
4    PRIMARY KEY (PlaylistId,TrackId),
5    ...) WITHOUT ROWID
```

- *A track, defined by its `TrackId` may belong to many playlists, each being defined by it `PlaylistID` and*
- *A playlists defined by it `PlaylistID` may contain many tracks, each being defined by its `TrackId`.*

Notice first that the **PRIMARY KEY** is required for all the **WITHOUT** ROWID tables. For other tables, it is optional. In this case, a rowid column that uniquely identifies each row inside the table is indeed created. However, this rowid attribute cannot be referenced.

Notice secondly that if an integer attribute is a primary key and you want its value to increase automatically with each insertion, this is very straightforward in SQLite. It is sufficient that the attribute is declared as INTEGER and **WITHOUT** ROWID must not be added.

Example 4.2. *Table `media_types` with `mediaTypeId` as auto increasing integer value*

The following query creates the `media_types` table.

```
CREATE TABLE media_types (
  MediaTypeId INTEGER PRIMARY KEY,
  Name VARCHAR(27))
```

*Since the `MediaTypeId` attribute is defined as a `INTEGER` **`PRIMARY KEY`** and since **`WITHOUT`** `ROWID` is not append, the attribute `MediaTypeId` will be automatically incremented with each insertion like:*

```
INSERT INTO media_types (Name) VALUES ('MPEG 3')
```

Exercise 4.3 (Create tables with autoincrement behavior).

1. *Create tables `artist` and `album` such that `artist.ArtisId` and `album.AlbumID` are `INTEGER` which are automatically incremented on each insertion.*
2. *Insert data into `artist` table and next into `album` one without setting the ID of the aforementioned tables.*

4.4 FOREIGN KEY

4.4.1 Motivating FOREIGN KEY with an example

Exercise 4.3 created the following 2 tables `artist` and `album` as follows:

```
CREATE TABLE artist(
  ArtistId INTEGER PRIMARY KEY,
  Name VARCHAR(85)
);
CREATE TABLE album(
  AlbumId INTEGER PRIMARY KEY,
  Title VARCHAR(95),
  ArtistId INTEGER -- Must map to an artist.ArtistId!
);
```

The comment in the declaration (namely `album.ArtistId` must map to an `artist.ArtistId`!) says that for each row in the album table there exists a corresponding row in the artist table. Unfortunately, it is only a comment. Rows might be inserted into the `album` table that do not correspond to any row in the `artist` table. Or rows might be deleted from the artist table, leaving orphaned rows in the track `album` that do not correspond to any of the remaining rows in `artist`.

During the table creation, one solution is to add an SQL foreign key definition into the table-constraint (after all attribute declaration) as follows:

```
CREATE TABLE album(
  AlbumId INTEGER PRIMARY KEY,
  Title VARCHAR(95),
  ArtistId INTEGER,
  FOREIGN KEY(ArtistId) REFERENCES artist(ArtistId))
);
```

Now, attempting to insert a row into the album table that does not correspond to any row in the artist table will fail, as will attempting to delete a row from the artist table when there exist dependent rows in the album table.

```
CREATE TABLE artist (ArtistId INTEGER PRIMARY KEY, Name VARCHAR(85));
INSERT INTO artist (Name) VALUES ("Frank Sinatra"), ("Dean Martin") ;
SELECT * FROM artist;
--1 Frank Sinatra
--2 Dean Martin


PRAGMA foreign_keys = ON;

CREATE TABLE album(
  AlbumId INTEGER PRIMARY KEY,
  Title VARCHAR(95),
  ArtistId INTEGER,
  FOREIGN KEY(ArtistId) REFERENCES artist(ArtistId));

--INSERT INTO album (Title, ArtistId) VALUES ("Dino: Italian Love Songs",200);
-- Error: FOREIGN KEY constraint failed
-- There is no artist with ArtistId equal to 200!

INSERT INTO album (Title, ArtistId) VALUES ("Dino: Italian Love Songs",2);
INSERT INTO album (Title, ArtistId) VALUES ("A Winter Romance",2);

SELECT * FROM album;
--1        Dino: Italian Love Songs  2
--2        A Winter Romance          2

DELETE FROM artist WHERE Name = 'Frank Sinatra';
SELECT * FROM artist;
--2        Dean Martin

--DELETE FROM artist WHERE Name = 'Dean Martin';
--Error: FOREIGN KEY constraint failed
```

4.4.2 `ON DELETE` and `ON UPDATE` actions

Foreign key `ON DELETE` and `ON UPDATE` clauses define actions that take place when deleting rows from the referenced table or modifying the referenced key values of existing rows. Basically, either action is prohibited (`RESTRICT` keyword) or it is propagated (`CASCADE` keyword) into dependant children as follows:

- `ON UPDATE RESTRICT`, `ON DELETE RESTRICT`: modifying or deleting a parent key which is referenced in one or more child keys is prohibited. It is the same behavior as without `ON UPDATE` or `ON DELETE`;
- `ON UPDATE CASCADE`, `ON DELETE CASCADE` propagates the update or delete operation on the parent key to each dependent child key. For an "`ON DELETE CASCADE`" action, this means that each row in the child table that was associated with the deleted parent row is also deleted. For an "`ON UPDATE CASCADE`" action, it means that the values stored in each dependent child key are modified to match the new parent key values.

Example 4.3. *`ON DELETE CASCADE`*

The following example shows the normal behavior with `ON DELETE CASCADE` constraint.

```
CREATE TABLE artist (ArtistId INTEGER PRIMARY KEY, Name VARCHAR(85));
INSERT INTO artist (Name) VALUES ("Frank Sinatra"), ("Dean Martin") ;

PRAGMA foreign_keys = ON;

CREATE TABLE album(
  AlbumId INTEGER PRIMARY KEY,
  Title VARCHAR(95),
  ArtistId INTEGER,
  FOREIGN KEY(ArtistId) REFERENCES artist(ArtistId) ON DELETE CASCADE ON UPDATE CASCADE);

INSERT INTO album (Title, ArtistId) VALUES ("Dino: Italian Love Songs",2);
INSERT INTO album (Title, ArtistId) VALUES ("A Winter Romance",2);
INSERT INTO album (Title, ArtistId) VALUES ("Christmas Songs by Sinatra",1);


DELETE FROM artist WHERE Name = 'Dean Martin';
```

```
SELECT * FROM album;

--3        Christmas Songs by Sinatra        1
-- all albums from 'Dean Martin' have been removed too.
```

Exercise 4.4 (Creation of tables with foreign keys and associated actions).

1. *Define the* `genres` *table such that the* `GenreId` *is a auto incremented integer primary key.*
2. *Define the* `tracks` *table such that the* `TrackId` *is a auto incremented integer primary key and whose attributes* `MediaTypeId`*,* `GenreId`*, and* `AlbumId` *are foreign keys with* **`CASCADE`** *action for* **`ON UPDATE`** *and for*

Bibliography

[Aud09] Laurent Audibert. *Bases de données : de la modélisation au SQL : conception des bases de données, modèle relationnel et algèbre relationnelle, langage SQL, programmation SQL : support de cours, exercices corrigés*. Ellipses, 2009.

[BA19] Sofia Benbelkacem and Baghdad Atmani. Random forests for diabetes diagnosis. In *2019 International Conference on Computer and Information Sciences (ICCIS)*, pages 1–4. IEEE, 2019.

[ED08] Khaled El Emam and Fida Kamal Dankar. Research paper: Protecting privacy using k-anonymity. *J. Am. Medical Informatics Assoc.*, 15(5):627–637, 2008.

[INS18] INSEE. Guide du secret statistique. Technical report, INSEE, 2018.

[MKGV07] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1):3, 2007.

[NC20] Benjamin Nguyen and Claude Castelluccia. Techniques d'anonymisation tabulaire : concepts et mise en oeuvre. *CoRR*, abs/2001.02650, 2020.

[otEU16] Official Journal of the European Union. Consolidated versions of the treaty on european union and the treaty on the functioning of the european union, 2016. 2016/C 202/01.

[sql] Sqlite tutorial. https://www.sqlitetutorial.net.

[Swe02] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.