CS258 Database Systems Coursework 2025-2026

GigSystem Specification

A music promotion company is planning to book acts for an upcoming festival. The system must consist of the following tables with the following attributes:

act

actID	SERIAL
actname	VARCHAR(100)
genre	VARCHAR(10)
standardfee	INTEGER

gig_ticket

gigID	INTEGER
pricetype	VARCHAR(2)
price	INTEGER

act_gig

actID	INTEGER
gigID	INTEGER
actgigfee	INTEGER
ontime	TIMESTAMP
duration	INTEGER

ticket

ticketID	SERIAL
gigID	INTEGER
pricetype	VARCHAR(2)
cost	INTEGER
customername	VARCHAR(100)
customeremail	VARCHAR(100)

gig

gigID	SERIAL
venueID	INTEGER
gigtitle	VARCHAR(100)
gigdatetime	TIMESTAMP
gigstatus	VARCHAR(1)

venue

venueID	SERIAL
venuename	VARCHAR(100)
hirecost	INTEGER
capacity	INTEGER

The data supplied will never exceed the lengths indicated above. For instance, CustomerName and CustomerEmail should never be more than 100 characters long, and genre will never be longer than 10 characters. You must not change any of the data types from those indicated. You must not add extra columns (or rename existing columns). You may create your own extra tables if desired (but this is not necessary, and not advised).

Acts (such as a band, solo musician, comedian, etc.) have a name and a standardfee (in £) that they would usually charge for being part of a gig. Act names are unique. Acts have a genre (such as “Classical”, “Rock”, “Pop” — these are just examples, you should not restrict genres). Act's can never have a negative standardfee (but may perform for free).

Gigs take place in a venue, have a gigtitle, take place on a particular gigdatetime, and have a gigstatus of either “C” (Cancelled) or "G" (GoingAhead) (this is always case-sensitive).

Acts are associated with a particular gig via the act_gig table. Each act charges an actgigfee (in £), which may be different to their standardfee for a gig. Act fees are not negative (but may be free). Acts have a particular ontime when they start (which must be at or after the gig's gigdatetime), and their performance lasts for a particular duration (a number of minutes).

Venues have a venuename, a hirecost (in £) and a capacity (maximum number of customers allowed at once). Venue names are unique. Venue hire costs are not negative (but may be free).

Customers buy tickets. There may be different types of ticket, each with a different price (in £). Every gig will always have a standard ticket type 'A' (for 'Adult'), but some gigs may also have concessionary tickets available (e.g. 'C' for Children or 'F' for free tickets — these are just examples, you should allow for extra pricetype values to be defined). Ticket prices are not negative (but may be 0 for free tickets). Ticket type 'A' is never free.

A customer's purchased ticket is stored in the ticket table with their CustomerName and CustomerEmail address. You can assume that a customer is identifiable by their email address (no two customers share an email address), and that the customer always uses the same name and email address when booking tickets. The cost of the ticket is the same as the price listed in gig_ticket for that pricetype, unless the cost of the ticket has been adjusted to 0 (see Task 4). Customers may buy multiple tickets for the same gig (they might be bringing friends).

Business Rules

The following business rules must also be enforced to keep the database in a valid state:

1. There must be no overlap between acts at a gig (although one act can start as soon as the previous act has finished, e.g. it's fine if act 1 finishes at 20:30 and act 2 starts at 20:30)
2. Acts cannot perform in multiple gigs at the same time
3. Acts can perform multiple times at the same gig (e.g., a first half, then an interval, then a second half)
    - Acts only receive one fee per gig, e.g. if their actgigfee is 1000, they only receive £1000 regardless of how many times they peform at a single gig
4. Each performance is no shorter than 15 minutes and no longer than 90 minutes
    - The same act cannot perform twice without a break (either an interval or a different act playing)
5. Acts can perform multiple gigs on the same day but need a 60 minute gap to travel between venues (all venues are close together, and don't require different travelling times)
    - Acts can travel to a different gig before the end of a gig (as long as the act has finished performing)
6. Venues can be used by multiple gigs on the same day (provided it's not at the same time) but need a 180 minute gap between gigs (so that staff can tidy the venue)
7. All intervals (breaks where no act is playing) must be no shorter than 10 minutes and no longer than 30 minutes
8. The first act must start at the same ontime as the gigdatetime of the gig
9. For each gig, there should not be more tickets sold than the capacity of the venue
10. The final act of a gig must finish at least 60 minutes after the start of the gig
11. Gigs involving genres 'Rock' or 'Pop' (case sensitive) at any point in the gig must finish no later than 11pm (inclusive) to prevent disruption to neighbouring residents, all other gigs must finish by 1am
12. Gigs cannot start after 11:59pm. The earliest a gig can start is 9am
13. All times can be assumed to be rounded to the nearest minute
14. Venue/Act clashes can ignore cancelled gigs

Part 1: Table Schema Definitions [20%]

Based on the above specification (with the same table names and column names), write a schema.sql file to create the necessary tables.

Use SQL to specify any key and referential integrity constraints you might need to capture the system requirements as accurately as possible. Also specify any additional constraints that you believe should apply to the system (based on the business rules). Some business rules may require extra processing, and will only be marked as part of the tasks that modify the database.

Additionally, your schema.sql file should include everything that is required to create the objects required for your database and is necessary to allow your JDBC code to run. This could include additional`VIEW`s, `SEQUENCE`s, `TRIGGER`s, `FUNCTION`s, `PROCEDURE`s — that will be marked as part of the tasks (defined below). Your schema.sql file should ***not*** include a `CREATE DATABASE` statement. When the coursework is marked, a database will be created for you.

Part 2: Design Choices [5%]

The tables as described in the introduction should not be changed for the purposes of this coursework. If you were to make any modifications, what changes might you make to improve the table structure? Write your answer in a section headed Design Choices in `README.md`.

Part 3: Application [75%]

Write (and test) a Java program that implements the Tasks specified below. In the source code we have provided, you will find two files:

- `GigSystem.java` contains a template of your main Java program. You must implement each of the tasks to carry out the described behaviour. You may also add a menu in the main method (similar to that used in the labs), but the marking will only assess the content of the Tasks' methods. If you implement a menu, you will be able to test your system interactively (allowing the user to type input, and sending it to your JDBC queries), using the readEntry method to gather user input, however, you must not read user input within the Task methods (otherwise automated tests will hang when calling these methods).
- `GigTester.java` will give you an idea of how your `GigSystem.java` methods will be called. There is a testTask method for each of the Tasks, but not all are fully implemented. The sample data provided is not an exhaustive set of tests, but are illustrative of the basic requirements of the query. In each case, there may be other situations that can occur that are not demonstrated by the sample data. You are expected to experiment with other data to ensure that your implementation can cope with all eventualities. More information about testing is given below.

Task 1: Gig Line-Up

The Java program needs to be able to find the line-up for any given gigID. There should be the actname, the time they will start and the time they will finish. Times should be given in 24 hour clock, without seconds. The `task1` method must return this information in the two-dimensional array of strings, as in the example shown below.

```
[[“ViewBee 40”, ”18:00”, ”18:50”],

[“The Where”, ”19:00”, ”20:25”],

[“The Selecter”, ”20:25”, ”21:25”]]
```

where the columns are 'Act Name', 'On Time', 'Off Time' respectively.

Note: You must return the array of Strings in the format described above. The strings represent a table as shown below. If you want to visualise such a table, you can use the printTable method.

Act Name	On Time	Off Time
ViewBee 40	18:00	18:50
The Where	19:00	20:25
The Selecter	20:25	21:25

Task 2: Organising a Gig

Set up a new gig at a given venue (referred to as a string containing the venue name).

There will be an array of ActPerformanceDetails objects which gives details of the actID, the fee, the the datetime the act will start, and the duration of the act. There will be a standard adult ticket price provided (adultTicketPrice).

The ActPerformanceDetails will not necessarily be provided in chronological order.

If any details of the gig (or acts) violate any of the business rules do not accept this gig, and ensure the database state is as it was before the method was called.

Task 3: Booking a Ticket

A customer wants to buy a ticket. You will be provided with a gigID, a customer name, email and a ticketType (which may match a pricetype from gig_ticket). If any details are inconsistent (e.g. if the gig does not exist, or there is no matching pricetype, or there is some other error), do not allow the ticket purchase and ensure the database state is as it was before the method was called.

Task 4: Cancelling an Act

An act needs to cancel a gig (gigID and actName supplied).

Remove all performances of the specified act from the specified gig. Adjust the ontime of all subsequent performances in the gig to remove the gap left by the cancellation (e.g., if the cancelled performance lasts 40 minutes, move all subsequent acts 40 minutes earlier). Do not attempt to connect/squash intervals (e.g. if the performance was between a 20 minute interval and a 20 minute interval, there would now be a 40 minute interval, which would violate the business rules). If the cancellation of the act would violate the constraints in the specification (e.g. intervals that are too long), or if the act is the headline act for this gig (the final or only act), cancel the entire gig.

To cancel just an act (updated line-up)

If the gig does not need to be cancelled, remove the act from the gig, adjust the timing as described above, and return the updated line-up (in the same format as task 1).

To cancel an entire gig

Change gigstatus to 'C' (cancelled). Leave all gig line-up (`act_gig`) details as they are (do not remove the cancelling act), but change the cost of all tickets sold for that gig to be 0 (but do not change the original price of the ticket).

Return a 2D array of strings containing names and email addresses of customers who have affected tickets, ordered by customer name (ascending alphabetical order), containing no duplicates.

Task 5: Tickets Needed to Sell

For each gig, find how many more tickets (of the cheapest ticket price for that gig) ***still*** need to be sold for the promoters to, at least, be able to pay all the agreed fees (as listed in the act_gig table, not from the act's standard fee) for the acts and to pay the venue fee. The output should only include the gigID and the tickets required to sell, ordered by gigID (smallest first). Include gigs that haven't sold any tickets yet. If a gig has already sold enough tickets to pay the agreed fees, represent this as 0 tickets.

You must return an array of strings that represent the table.

Sample output (headings must not be in your output, these illustrate expected column order):

gigID	Tickets To Sell
1	1600
2	2000
3	1525

Task 6: How Many Tickets Sold

Create a 2-dimensional array of strings to show the total number of tickets (of any pricetype) that each act has sold. Only consider gigs where the act is listed as a headline act (the final or only act), and only include gigs that are not listed as cancelled.

For each act, find the number of tickets sold per year and show the total number of tickets ever sold by each act when the act was a headline act.

Order your result with the acts who have sold the least total number of tickets first, then ordered by Year (with 'Total' at the end of each group of years).

Sample output (headings must not be in your output, these illustrate expected column order):

Act name	Year	Total Tickets Sold
QLS	2018	2
QLS	2019	1
QLS	Total	3
ViewBee 40	2017	3
ViewBee 40	2018	1
ViewBee 40	Total	4
Scalar Swift	2017	3
Scalar Swift	2018	1
Scalar Swift	2019	1
Scalar Swift	Total	5
Join Division	2016	2
Join Division	2018	2
Join Division	2020	3
Join Division	Total	7
The Selecter	2017	4
The Selecter	2018	4
The Selecter	Total	8
The Where	2016	1
The Where	2017	3
The Where	2018	5
The Where	2020	4
The Where	Total	13

Task 7: Regular Customers

The festival organisers want to know who regularly attends gigs that feature particular acts. Create a two-dimensional array of strings that shows each act who has ever performed a gig as a headline act (the final or only act) along with the names of customers who have attended at least one of these gigs per calendar year (if the act performed such a gig as a headline act in that year). The output should list the acts in alphabetical order and the customers in order of the number of tickets that the customer has ever bought for a gig where that act was the headline act (with customers who have bought the most tickets listed first). Only consider gigs that have not been cancelled.

If the act has no such customers, and the act has played gigs as a headline act, make sure the act is still listed, but with '[None]' in the Customer Name column.

Sample output (headings must not be in your output, these illustrate expected column order):

Act Name	Customer Name
Join Division	G Jones
QLS	[None]
Scalar Swift	G Jones
Scalar Swift	J Smith

Task 8: Economically Feasible Gigs

The festival organisers want to organise a gig with a single act. They're trying to choose an act for a specific venue, but don't want to charge more than the average ticket price, rounded to the nearest £ (you can assume this is a positive number, not free, and there will always be a positive number of tickets sold). However, the organisers want to make sure they break even, so they need to be able to sell enough tickets to be able to pay for the act's standardfee and the cost of hiring the venue.

Based on the standardfee for each act, and the average price amongst all tickets that have been sold (amongst all gigs that have not been cancelled), create a two-dimensional array of strings that includes each venue and each act that it would be economically feasible to book in that venue (assuming that act is the only act at the gig). If there are no such acts, do not include any rows for the venue.

Also include the minimum number of tickets that would need to be sold (assuming each ticket is sold at the average price). Order your output in alphabetical order of venue name, then the number of tickets required (with highest number of tickets first). Hint: tickets can only be sold in integer numbers; it's impossible to sell half a ticket.

Sample output (headings must not be in your output, these illustrate expected column order):

Venue Name	Act Name	Tickets Required
Arts Centre Theatre	Join Division	150
Big Hall	The Where	675
Big Hall	Join Division	375
Cinema	Join Division	175
City Hall	Join Division	225
Symphony Hall	ViewBee 40	1275
Symphony Hall	Scalar Swift	1250
Symphony Hall	QLS	1225
Symphony Hall	The Selecter	1200
Symphony Hall	The Where	825
Symphony Hall	Join Division	525

Town Hall	The Where	575
Town Hall	Join Division	275
Village Green	Join Division	100
Village Hall	Join Division	75

Writing your code

You must not modify the existing method signatures - i.e., you must not change method names, input parameters or return types in `GigSystem.java`

For methods that require returning output, it is recommended to use the existing `convertResultToStrings` method.

For debugging purposes, you might find it useful to use the `printTable` method.

For `GigSystem.java`, you must only import what is immediately available on the DCS system (do not use external libraries, and do not modify the Maven `pom.xml` file). Other than parts of `java.sql`, we strongly recommend that you do not import other libraries (the best solutions will only require minimal Java). For `GigTester.java`, you can import any Java library, as you don't need to submit it.

Full instructions on getting started can be found in `INSTRUCTIONS.md`

Testing and Marking

At all times (before a task method starts and after it has completed, even if the task encounters an error), the integrity of the database (following the specification) must be maintained.

Before a Java task is called, the database will be in a known state that does not violate the stated rules. Whenever a task has completed, we will check that none of the database integrity rules have been violated. Some test data (separate files from those provided) will be loaded in replication mode (see the `run.sh` script), which means that `TRIGGER`s will not fire during the insertion of the data (unless they have specifically been defined with `ENABLE REPLICA` or `ENABLE ALWAYS`, see https://www.postgresql.org/docs/current/sql-altertable.html — we strongly suggest that you do not use `ENABLE REPLICA` or `ENABLE ALWAYS`, otherwise the test data may not be initialized as expected). After the test data has been inserted, the database is switched back to the normal '*origin*' mode, which will mean that triggers will behave as expected when the task methods are called. Other than this initial loading phase, each test will only modify the database via the Java task methods - data supplied will always be of the correct datatype, but may break business rules. You must ensure the database is not in an inconsistent state (i.e., violating business rules) after a Java task has been called.

Only the inputs/outputs of methods will be tested, there's no need for "friendly error handling" — in fact, it's better to allow exceptions to be logged (with stacktrace) to help markers to identify issues that can be explained in feedback.

If a result is empty, you can return either an empty String array (probably easiest), or null.

You must not use separate schema paths (i.e., do not use `CREATE SCHEMA`), your tables should be of the form `CREATE TABLE act` ***do not use*** `CREATE TABLE public.act` or `CREATE TABLE myschema.act`.

You should not assume the order that tasks are called (e.g. don't rely on task1 being called before task2, etc.). Methods must not prevent execution of other methods — for example, do not terminate execution of the entire program within the task methods (only return from the methods as described in the specification).

We have provided you with a test suite in `GigTester.java`. There is a testTask method for each of the tasks, but not all are fully implemented, and you may like to extend the existing ones to think about different cases.

There is no need to submit your `GigTester.java` file, however we strongly recommend that you do write tests. The coursework will be marked using a similar framework to `GigTester.java`. If a task does not work in the test suite, it is likely to score 0 when marked.

There are two data files provided:

- `tests/testbig.sql` — a sample of 50 gigs with randomly assigned acts
- `tests/testsmall.sql` — a sample of 5 gigs, chosen specifically to help you experiment with tasks 7 and 8

The sample data provided is not an exhaustive set of tests, they should help you understand the basic requirements of the method. In each case, there may be other situations that can occur that are not demonstrated by the sample data.

The `tests/testbig.sql` and `tests/testsmall.sql` files do not violate the stated rules but generating random test data is likely to violate some of the rules. For more information about using the test suite, please see the `INSTRUCTIONS.md` file

When using the test suite, the reset-data.sql file automatically sets all sequences to be 10001. This is to ensure that test data can be inserted with IDs between 1 and 10000. This should automatically work provided you specify the appropriate columns as `SERIAL` as shown on page 1. There's no need to roll sequences back if you cancel an insertion — there is no importance on the exact value of serial numbers. Please make sure that your system does not rely on any values below 10000, as these may contain test data during the marking process.

Submission

You are expected to submit exactly three files (any additional files submitted will not be marked/tested):

- `schema.sql` — containing all the SQL definitions needed to recreate your database
- `README.md` — containing information about your solution and answers to the 'Design Choices' part. This can contain plain text, or you can use markdown formatting.
- `GigSystem.java` — your completed java file

Document your solutions

Any decisions you make should be justified in your `README.md` file (this can be plain text, or you can add Markdown formatting if you like). For each task, you should write around 50-300 words explaining what your solution does (describing the behaviour of your SQL statements/queries). Any Java that does not simply pass/retrieve data to/from SQL should be commented.

What constitutes a good answer?

Use prepared statements where appropriate — this is good for both security and efficiency.

It is expected that your program should be able to appropriately handle any error cases that might occur (such as invalid inputs or failures that arise from violating database constraints). You should catch exceptions, but there is no need to print 'friendly' error messages — it is better to print the stack trace so that any exceptions can be interpreted. Our automated system will check whether the data has been inserted correctly and whether the correct values are returned.

Use SQL instead of Java where possible. To maintain the integrity of the database (including the business rules), higher marks will be awarded to solutions that make full use of the principles of the RDBMS (Postgres) rather than using procedural techniques. For each query that returns a result (those methods that have return type `String[][]`), it is possible to write an answer that gets all data from the database without processing the query result (other than converting your ResultSet to a two-dimensional String array using the `convertResultToStrings` method). Any solutions that use Java to help with filtering, ordering or changing the structure of the returned data will receive a much lower mark than completely SQL solutions.

FAQ

An FAQ will be maintained at the module material page.

Final Remarks

Please make sure your code works on the DCS servers (you must use PostgreSQL 16, which is the version installed on the DCS servers). Keep a copy of everything you submit! You may discuss with fellow students the material covered in lectures and seminars, but you are not allowed to collaborate on the assignment.

The University of Warwick takes plagiarism seriously, and penalties will be incurred if any form of plagiarism is detected. Copying, or basing your work on, solutions written by people who have not taken this module is also counted as plagiarism. This includes material that has been downloaded from the internet.

Use of Large Language Models and Generative AI Tools

In line with the Department of Computer Science policy on the use of Large Language Models (LLM) and Generative AI (GenAI) tools (such as ChatGPT, Microsoft CoPilot, Google Bard/Gemini, etc.), tools such as these can be used for the generation of new avenues of thought. However, these cannot be used as a replacement to learning and work. As such, alongside the Department of Computer Science guidance on these tools, you should provide the following information if you decide to use these tools:

- The name and version number of the LLM and/or GenAI tool used
- The task the prompt was used for
- The exact prompt given
- The exact results produced by the LLM and/or GenAI tool
- The date and time the prompt was given

Correct use of LLMs and/or GenAI will not be used to reduce your marks, but to see where your information has been attained. As such, it is important that all prompts are included in the README.md (if an LLM and/or GenAI has been used).

This information should be provided as an Appendix to your work. To aid in this, a template and example has been provided at the end of this problem sheet (Appendix A). Failure to present any prompt that was used could be considered as plagiarism, as you would be presenting work that was not your own.

If you decide not to use LLM and/or GenAI tools, then you should clearly state this in the README.md.

For more information on this, please see the relevant section on the Student Handbook page on coursework.

https://warwick.ac.uk/fac/sci/dcs/teaching/handbook/coursework/