Pulse University Festival

Βάσεις Δεδομένων

Εξαμηνιαία Εργασία ΗΜΜΥ ΕΜΠ 2025

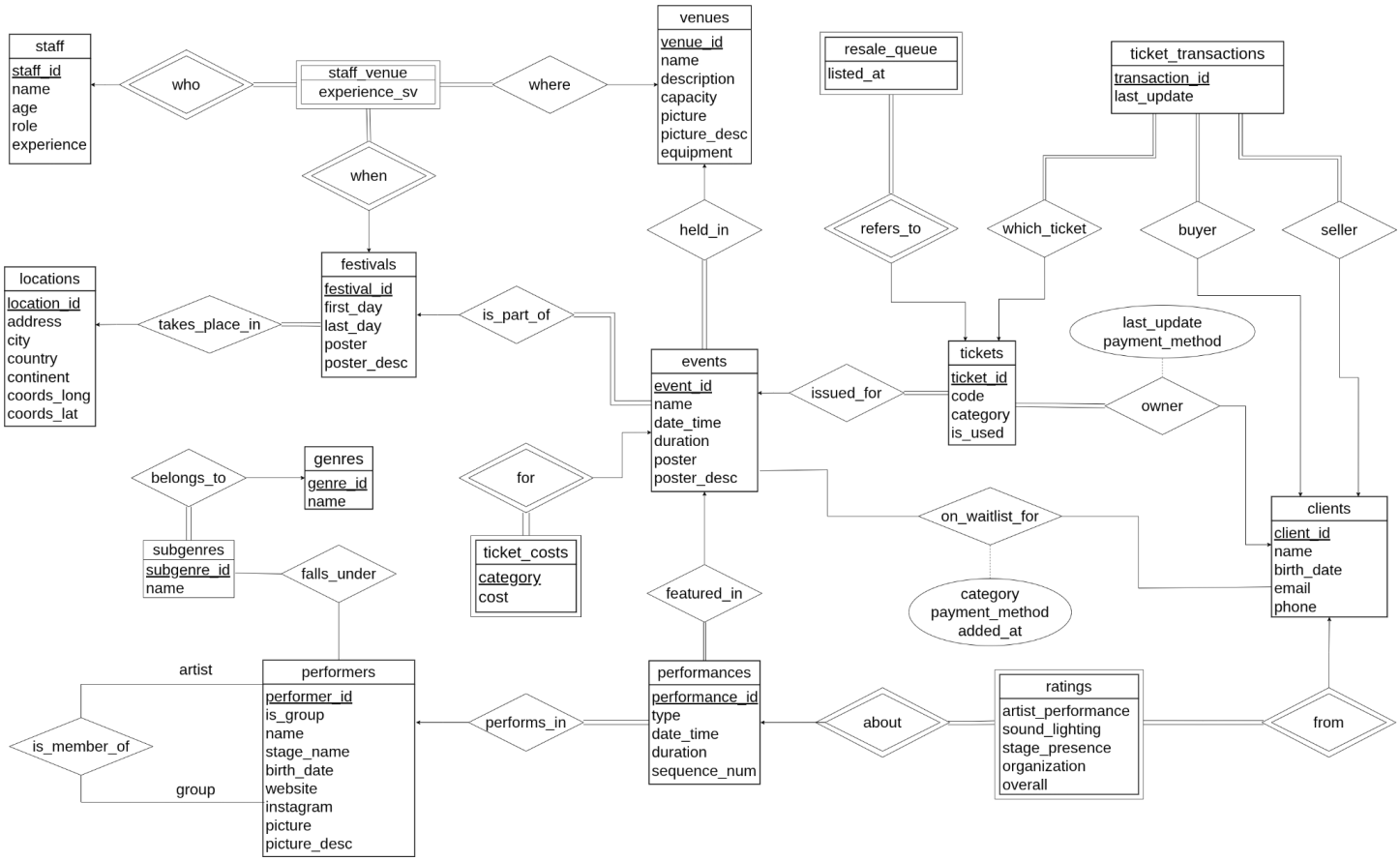
https://gitlab.com/xheonin/database-ntua2025

Φοιτητές

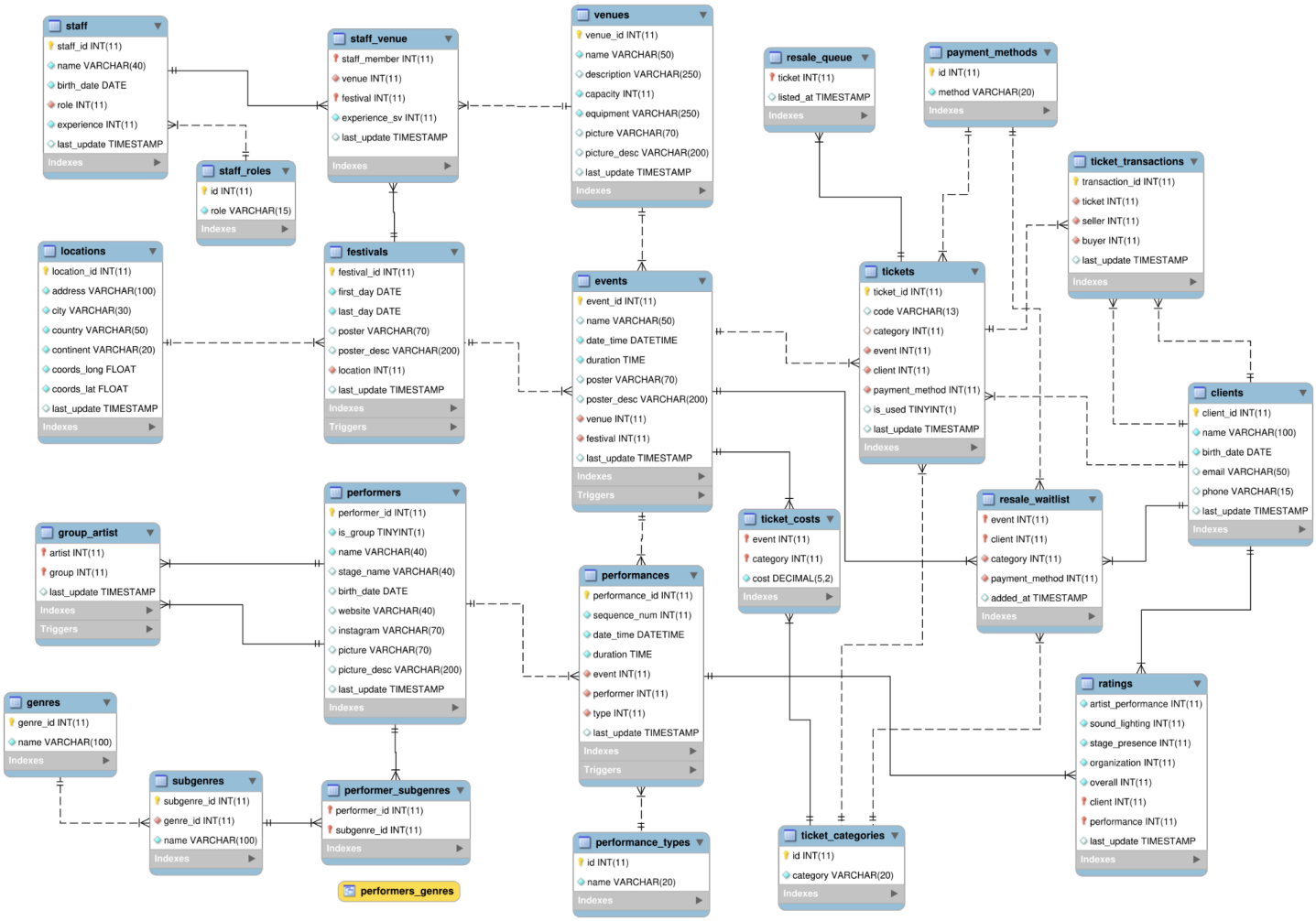
Μαντζώρος Γεράσιμος (03122011)
Μωραΐτης Δημήτρης (03122175)
Τσιόδρας Νίκος (03122143)

Διαγράμματα

Entity - Relationship diagram



Relational schema



Tables

Για την υλοποίηση της βάσης δεδομένων του φεστιβάλ μουσικής Pulse University υλοποιήσαμε τα εξής tables με τους περιορισμούς που θα περιγράψουμε:

locations

Εκεί αποθηκεύονται οι τοποθεσίες στις οποίες μπορούν να γίνουν τα φεστιβάλ. Περιλαμβάνει στοιχεία για την διεύθυνση, την πόλη, την χώρα, την ήπειρο και τις συντεταγμένες.

festivals

Εκεί αποθηκεύονται δεδομένα για τα φεστιβάλ κάθε χρονιάς. Έχει πληροφορίες για την πρώτη και την τελευταία μέρα του φεστιβάλ, την αφίσα με την περιγραφή της και την τοποθεσία ως foreign key από το table locations.

venues

Εκεί αποθηκεύονται τα κτίρια και οι σκηνές που θα γίνονται οι διάφορες παραστάσεις. Περιέχει τα ονόματα, τις περιγραφές, τις χωρητικότητες, τον τεχνικό εξοπλισμό και εικόνες με τις περιγραφές τους για τις σκηνές.

events

Εκεί αποθηκεύονται όλες οι παραστάσεις για κάθε φεστιβάλ. Περιλαμβάνει το όνομα κάθε παράστασης, την ημερομηνία, την διάρκεια, την αφίσα με την περιγραφή της, την σκηνή που θα διεξαχθεί (ως foreign key από το venues) και το φεστιβάλ που θα γίνει (ως foreign key από το festivals).

performers

Εκεί αποθηκεύονται οι καλλιτέχνες και τα συγκροτήματα που θα παίρνουν μέρος στις παραστάσεις κάθε φεστιβάλ. Αρχικά υπάρχουν πληροφορίες για το όνομα, την ημερομηνία γέννησης, την ιστοσελίδα, τα μέσα κοινωνικής δικτύωσης και μια εικόνα για τον καθένα. Επίσης υπάρχει attribute για το εάν μία καταχώρηση αφορά μεμονωμένο καλλιτέχνη ή συγκρότημα. Υπάρχει ένα table "**group_artist**" το οποίο διαχειρίζεται μια σχέση many to many μεταξύ καταχωρήσεων του table performers, ώστε να γνωρίζουμε ποιοι καλλιτέχνες είναι μέρος ενός ή περισσότερων συγκροτημάτων.

genres & subgenres

Εκεί αποθηκεύονται τα είδη και υποείδη μουσικής. Λειτουργούν ως "enum tables" καθώς τα στοιχεία τους χρησιμοποιούνται κατά κόρων ως foreign key σε attributes από άλλα tables. Το subgenres έχει foreign key το genre_id καθώς κάθε υποείδος πρέπει να σχετίζεται με ένα κύριο είδος.

performer_subgenres

Το table αυτό διαχειρίζεται την σχέση many to many μεταξύ καλλιτεχνών και υποειδών. Για να βρεθεί το κύριο είδος ενός καλλιτέχνη ψάχνουμε μέσω του subgenre.

performance_types

Enum table για τα είδη των παραστάσεων (πχ. warm up, headline, special guest).

performances

Εκεί αποθηκεύονται όλες οι εμφανίσεις για κάθε παράσταση των φεστιβάλ. Η κάθε καταχώρηση έχει πληροφορίες για την σειρά της εμφάνισης στην παράσταση που θα διεξαχθεί, την ώρα έναρξης, την διάρκεια, την παράσταση (ως foreign key από το events), τον καλλιτέχνη (ως foreign key από το performers) και τον τύπο (ως foreign key από το performance_types). Για την διάρκεια υπάρχει περιορισμός να είναι από 0 έως 3 ώρες. Ο περιορισμός για την διάρκεια των διαλειμμάτων ελέγχεται από τη βάση δυναμικά, κατά την εισαγωγή των δεδομένων, χρησιμοποιώντας το attribute sequence number.

clients

Εκεί αποθηκεύονται όλοι οι εγγεγραμμένοι πελάτες του συστήματος. Περιέχει πληροφορίες για το όνομα, την ημερομηνία γέννησης, την ηλεκτρονική διεύθυνση και το τηλέφωνο του καθενός.

staff_roles

Enum table για τους ρόλους του προσωπικού (πχ. technician, security, helper).

staff

Εκεί αποθηκεύονται όλοι οι υπάλληλοι των φεστιβάλ. Έχει στοιχεία για το όνομα, την ημερομηνία γέννησης, τον ρόλο του κάθε εργαζόμενου (ως foreign key από το staff_roles) και την εμπειρία του (με περιορισμό να είναι ανάμεσα στις τιμές 0 εως 5).

staff_venue

Είναι το table μιας σχέσης many to many μεταξύ των staff, των venues και των festival, επιδεικνύοντας ποιοί υπάλληλοι δουλεύουν/αν σε συγκεκριμένες σκηνές και σε συγκεκριμένο φεστιβάλ. Πέρα από τα attributes για staff_member, venue και festival, υπάρχει και το experience_sv για να γνωρίζουμε την εμπειρία που είχε κάποιος εργαζόμενος ανα χρονιά. Αυτή η επιλογή έγινε καθώς η εμπειρία ενός εργαζόμενου ενδέχεται να αυξηθεί κάθε χρόνο.

ticket_categories

Enum table για τις κατηγορίες των εισιτηρίων (πχ. Standard, vip, backstage).

payment_methods

Enum table για τις μεθόδους πληρωμής ενός εισιτηρίου (πχ. credit, debit, wire).

tickets

Εκεί αποθηκεύονται όλα τα αγορασμένα εισιτήρια για τις παραστάσεις των φεστιβάλ. Περιλαμβάνει τον κωδικό EAN-13 του κάθε εισιτηρίου, την κατηγορία του (ως foreign key από το ticket_categories), την παράσταση για την οποία προορίζεται (ως foreign key από το events), τον πελάτη που το αγόρασε (ως foreign key από το clients), τον τρόπο πληρωμής (ως foreign key από το payment_methods) και ένα attribute που μας δείχνει αν το εισιτήριο έχει χρησιμοποιηθεί.

ticket_costs

Εκεί αποθηκεύονται οι τιμές των εισιτηρίων ανα κατηγορία εισιτηρίου και παράσταση. Έχει την παράσταση ως foreign key από το events και την κατηγορία ως foreign key από το ticket_categories.

resale_waitlist

Αυτό το table αποτελεί την ουρά πελατών για την μεταπώληση εισιτηρίων, δηλαδή μια λίστα με τους πελάτες οι οποίοι περιμένουν να αγοράσουν εισιτήριο (από μεταπώληση) για συγκεκριμένη παράσταση και κατηγορία εισιτηρίου. Έχει την παράσταση ως foreign key από το events, τον πελάτη ως foreign key από το clients, την κατηγορία ως foreign key από το ticket_categories και τον τρόπο πληρωμής ως foreign key από το payment_methods.

resale_queue

Εκεί αποθηκεύονται όλα τα εισιτήρια τα οποία έχουν τεθεί διαθέσιμα για μεταπώληση. Έχει το εισιτήριο ως foreign key από το tickets. Η διαχείριση των καταχωρήσεων σε αυτό και το προηγούμενο table γίνεται με procedures τα οποία είναι υπεύθυνα για τον χειρισμό των μεταπωλήσεων.

ticket_transactions

Εδώ καταχωρούνται όλες οι μεταπωλήσεις εισιτηρίων. Περιέχουν το εισιτήριο ως foreign key από το tickets, τον πωλητή και τον αγοραστή ως foreign key από το clients.

ratings

Σε αυτό το table καταχωρούνται όλες οι αξιολογήσεις που έχουν κάνει οι πελάτες για κάποια εμφάνιση. Κάθε αξιολόγηση περιέχει βαθμολογία για την ερμηνεία των καλλιτεχνών, τον ήχο και φωτισμό, την σκηνική παρουσία, την οργάνωση και την συνολική εντύπωση, όλες από τις οποίες είναι σε κλίμακα από 1 έως 5. Επίσης έχει τον πελάτη που έκανε την αξιολόγηση ως foreign key από το clients και την εμφάνιση για την οποία πρόκειται ως foreign key από το performances.

Triggers

Όλα από τα παρακάτω triggers μας αναγκάζουν να προσθέτουμε έγκυρα δεδομένα βάσει των ζητούμενων περιορισμών. Αν πρόκειται να προσθέσουμε ένα δεδομένο το οποίο δεν πληρεί τις προϋποθέσεις τότε τυπώνουμε το αντίστοιχο σφάλμα και διακόπτουμε την καταχώρηση.

different_location_each_year

Ελέγχει κάθε φορά που εισάγουμε ένα καινούριο φεστιβάλ στην βάση δεδομένων μας να έχει διαφορετική τοποθεσία από το αμέσως προηγούμενό του.

check_performance_time

Ελέγχει κάθε εμφάνιση που εισάγεται να βρίσκεται μέσα στα όρια που ορίζει της παράστασης στην οποία ανήκει. Επίσης υπολογίζει και ελέγχει την διάρκεια του διαλείμματος σε σχέση με την προηγούμενη εμφάνιση να είναι από 5 έως 30 λεπτά.

check_event_date_within_festival

Ελέγχει αν η παράσταση που εισάγεται είναι μεταξύ των χρονικών ορίων του εκάστοτε φεστιβάλ.

deny_overlapping_events

Ελέγχει αν η παράσταση που εισάγεται επικαλύπτεται χρονικά με κάποια άλλη. Χρησιμοποιεί την συνάρτηση check_time_overlap για να υπολογίσει τις επικαλύψεις.

check_artist_availability

Το trigger αυτό ελέγχει σε κάθε εισαγωγή μιας εμφάνισης αν ο καλλιτέχνης ή το συγκρότημα που συμμετέχει σε αυτό είναι διαθέσιμος. Δηλαδή:

- Στην περίπτωση μεμονωμένου καλλιτέχνη, ελέγχει αν την ώρα της εμφάνισης είναι προγραμματισμένος να εμφανιστεί αλλού μόνος του είτε σε συγκρότημα που συμμετέχει
- Αντίστοιχα στην περίπτωση συγκροτήματος, κάνει τον έλεγχο για όλο το συγκρότημα και τους παραπάνω ελέγχους για όλα τα μέλη του

Εδώ χρησιμοποιούμε ξανά την συνάρτηση check_time_overlap για τους ελέγχους.

check_group_artist_integrity

Ελέγχει ότι το ζευγάρι καλλιτέχνη-συγκροτήματος που προστίθεται στην σχέση is_member_of (βλέπε ER) είναι έγκυρο. Δηλαδή βλέπει αν το attribute is_group για τον καλλιτέχνη είναι false ενώ για το συγκρότημα είναι true.

check_consecutive_participations

Στην προσθήκη νέας εμφάνισης, ελέγχει αν ο καλλιτέχνης ή το συγκρότημα που συμμετέχει έχει λάβει μέρος σε εμφανίσεις τα 3 τελευταία χρόνια.

Procedures

purchase_ticket

Αποτελεί την κύρια συνάρτηση που χρησιμοποιεί η βάση δεδομένων για να γίνει αγορά ενός εισιτηρίου. Παίρνει ως παραμέτρους τον επισκέπτη, την παράσταση, την επιθυμητή κατηγορία εισιτηρίου και τον τρόπο πληρωμής και ελέγχει:

- Αν ο ενδιαφερόμενος πελάτης έχει ήδη εισιτήριο για την παράσταση (άρα δεν μπορεί να αγοράσει καινούργιο) και
- Αν η κατηγορία του εισιτηρίου για την συγκεκριμένη παράσταση είναι sold out.

Αν περάσει τους παραπάνω ελέγχους τότε και μόνο τότε εισάγει στο table tickets την αντίστοιχη καταχώρηση. Επίσης παράγει τον κωδικό EAN, μέσω της συνάρτησης gen_ean13(), και τον προσθέτει στο εισιτήριο που μόλις έβαλε μέσα στη βάση.

list_for_resale

Χρησιμοποιείται ώστε ένας πελάτης να μπορεί να τοποθετεί το εισιτήριο του στην ουρά μεταπώλησης. Παίρνει ως παράμετρο το ticket_id και ελέγχει αν πληρούνται οι παρακάτω προϋποθέσεις ώστε να μπορεί να γίνει μεταπώληση του εν λόγω εισιτήριου:

- Πρέπει να είναι έγκυρο εισιτήριο
- Το εισιτήριο πρέπει να μην είναι χρησιμοποιημένο (ελέγχεται μέσω του attribute is_used)
- Η ουρά μεταπώλησης για την παράσταση να είναι ανοιχτή, δηλαδή να είναι sold out

Στην συνέχεια η διαδικασία ελέγχει αν υπάρχει τουλάχιστον ένας ενδιαφερόμενος για αγορά του ζεύγους παράστασης - κατηγορίας εισιτηρίου και αν βρεθεί εκτελείται αυτόματα η συναλλαγή, δηλαδή γίνεται καταχώρηση στο table ticket_transactions με τον πωλητή, τον αγοραστή και το εισιτήριο, διαγράφεται ο ενδιαφερόμενος από την λίστα αναμονής και ανανεώνονται τα πεδία client και payment_method του εισιτηρίου. Σε περίπτωση που δεν υπάρχει ενδιαφερόμενος, προστίθεται το εισιτήριο στην σειρά μεταπώλησης (resale_queue).

purchase_from_resale

Χρησιμοποιείται ώστε ένας πελάτης να μπορεί να αγοράσει ένα εισιτήριο από την ουρά μεταπώλησης. Παίρνει ως παραμέτρους τον ενδιαφερόμενο αγοραστή, την παράσταση, την κατηγορία εισιτηρίου και τον τρόπο πληρωμής και εκτελεί τους παρακάτω ελέγχους:

- Ο ενδιαφερόμενος να μην έχει ήδη εισιτήριο για αυτό το event
- Το event να είναι sold out

Στην συνέχεια η διαδικασία ελέγχει αν υπάρχει ήδη κάποιο εισιτήριο στην ουρά μεταπώλησης με τα απαιτούμενα χαρακτηριστικά και αν βρεθεί εκτελείται η συναλλαγή αντίστοιχα με πριν. Εάν δεν βρεθεί κάποιο εισιτήριο, τότε ο πελάτης μπαίνει στην λίστα αναμονής resale_waitlist.
Αξίζει να σημειωθεί πως ο τρόπος λειτουργίας των διαδικασιών list_for_resale και purchase_from_resale εξασφαλίζει ότι ποτέ δεν θα υπάρξει ενδιαφερόμενος πελάτης στην ουρά αναμονής και συμβατό εισιτήριο στην ουρά μεταπώλησης, αφού η εισαγωγή στις ουρές γίνεται μόνο όταν αποτύχει ο έλεγχος για συμβατότητα. Επίσης για να μην υπάρχουν race condition μεταξύ εισαγωγών στους πίνακες resale_queue και resale_waitlist χρησιμοποιήσαμε το keyphrase “FOR UPDATE” το οποίο βάζει locks σε γραμμές που επιλέγονται από τα αντίστοιχα SELECT στους πίνακες αυτούς.

Indexes

Οι επιλογές μας για index βασίστηκαν στα foreign key του κάθε table έτσι ώστε να επιταχύνουμε τις join operations, καθώς και σε attributes που συχνά χρησιμοποιούνται για ταξινόμιση, ομαδοποίηση και φιλτράρισμα.

Views

Για την διευκόλυνση εύρεσης ορισμένων στοιχείων στο σχήμα μας, δημιουργήσαμε το παρακάτω view:

performers_genres

Εκεί φαίνονται όλα τα είδη μουσικής για κάθε καλλιτέχνη ή συγκρότημα. Αυτό είναι χρήσιμο καθώς εξαρχής υπάρχει μόνο μια σχέση many to many (falls_under από το ER) μεταξύ υποειδών και καλλιτεχνών, ενώ το κύριο είδος βρίσκεται μέσω του attribute genre_id στο υποείδος. Αυτή η επιλογή έγινε για λόγους κανονικοποίησης.

Functions

check_time_overlap

Ελέγχει αν δύο χρονικά διαστήματα επικαλύπτονται, λαμβάνοντας υπόψη ημερομηνία και ώρα.

gen_ean13

Παράγει έναν κωδικό EAN-13 για κάποιο δοσμένο αριθμό (id του εισιτηρίου).

Παραδοχές

Έγιναν οι εξής παραδοχές από την εκφώνηση της εργασίας:

- Το φεστιβάλ δεν μπορεί να διεξαχθεί στην ίδια τοποθεσία για δύο διαδοχικές χρονιές.
- Η διάρκεια του διαλείμματος μιας εμφάνισης δεν αποθηκεύεται στην καταχώρηση του performances, αλλά υπολογίζεται δυναμικά μέσω trigger (check_performance_time) με την χρήση της διάρκειας και την σειρά της εμφάνισης.
- Όταν γίνεται ο έλεγχος για 3 διαδοχικές συμμετοχές ενός καλλιτέχνη, δεν προσμετρούνται συμμετοχές του μέσα σε ένα συγκρότημα.

Σχεδιαστικές Επιλογές

- Θεωρούμε ότι οι υπάλληλοι για κάθε φεστιβάλ ανατίθενται σε κάποιο συγκεκριμένο venue, στο οποίο εργάζονται για όλη τη διάρκεια του φεστιβάλ και υποστηρίζουν όλες τις παραστάσεις που πραγματοποιούνται σε αυτό.
- Στις καταχωρήσεις του staff αποθηκεύεται η τρέχουσα εμπειρία ενός υπαλλήλου, ενώ στις καταχωρήσεις του staff_venue καταγράφεται η εμπειρία του την συγκεκριμένη χρονιά.
- Στα ενδεικτικά δεδομένα ο αριθμός των εργαζομένων ανα σκηνή (venue) είναι ακριβώς:
    - το 5% της χωρητικότητας για προσωπικό ασφαλείας
    - το 2% της χωρητικότητας για βοηθητικό προσωπικό και
    - το 3% της χωρητικότητας για τεχνικό προσωπικό.

Ερωτήματα

Ερώτημα 1

Βρίσκουμε τα συνολικά έσοδα του festival ανά χρονιά και παρέχουμε μία ανάλυση ανά τρόπο πληρωμής. Για να το πετύχουμε αυτό κάνουμε τα κατάλληλα JOIN ώστε να φτάσουμε από το κάθε ticket στο αντίστοιχο festival και χρησιμοποιούμε την συνάρτηση άθροισης SUM σε συνδυασμό με CASE ώστε να διαχωρίσουμε τους τρόπους πληρωμής.

```
SELECT
 YEAR(f.`first_day`) AS `year`,
 SUM(cst.`cost`) AS `income`,
 SUM(CASE WHEN inf.`payment_method` = 1 THEN cst.`cost` ELSE 0 END) AS `income_debit`,
 SUM(CASE WHEN inf.`payment_method` = 2 THEN cst.`cost` ELSE 0 END) AS `income_credit`,
 SUM(CASE WHEN inf.`payment_method` = 3 THEN cst.`cost` ELSE 0 END) AS `income_wire`
FROM
 `tickets` inf
 JOIN `events` evn
   ON evn.`event_id` = inf.`event`
 JOIN `ticket_costs` cst
   ON cst.`event` = inf.`event` AND cst.`category` = inf.`category`
 JOIN `festivals` f
   ON f.`festival_id` = evn.`festival`
GROUP BY f.`festival_id`;
```

Ερώτημα 2

Θέτουμε αυθαίρετα το μουσικό είδος σε “Pop” και το έτος στο 2017. Αρχικά συνδέουμε με LEFT JOIN όλους τους καλλιτέχνες με όλες τις εμφανίσεις τους, για τις οποίες ο καλλιτέχνης ασχολείται με “Pop” μουσική και το έτος της εμφάνισης είναι 2017. Συνεπώς μετά το LEFT JOIN έχουμε όλους τους σχετικούς καλλιτέχνες στα “αριστερα” και μετά:

- Ένδειξη ‘yes’ αν ο καλλιτέχνης εμφανίστηκε την αντίστοιχη χρονία.
- “Ένδειξη ‘no’ αν δεν εμφανίστηκε την αντίστοιχη χρονία.

Συνεπώς στο SELECT μπορούμε να δείξουμε κατάλληλα τους “Pop” καλλιτέχνες οι οποίοι είχαν εμφανιστεί το 2017.

```
SET @myGenre = "Pop";
SET @myYear = 2017;

SELECT DISTINCT
 PGen.`performer_name` AS `performer_name`,
 PGen.`performer_id` AS `performer_id`,
 CASE
   WHEN PInf.`performer` IS NOT NULL THEN 'Yes'
   ELSE 'No'
 END AS `participated_this_year`
FROM `performers_genres` PGen
LEFT JOIN `performances` PInf
ON PGen.`performer_id` = PInf.`performer`
AND YEAR(PInf.`date_time`) = @myYear
WHERE PGen.`genre_name` = @myGenre;
```

Ερώτημα 3

Κάνουμε JOIN τις εμφανίσεις, τις παραστάσεις και τους καλλιτέχνες, φιλτράρουμε για τύπο εμφάνισης 1 (warm up) και κάνουμε GROUP BY με το performer_id και το festival (το οποίο μας δίνει το event). Συνεπώς χρησιμοποιώντας το keyphrase “HAVING COUNT(*) > 2” παίρνουμε το επιθυμητό αποτέλεσμα.

```
SELECT DISTINCT prs.`name`
FROM `performances` pes
JOIN `events` eve
ON pes.`event` = eve.`event_id`
JOIN `performers` prs
ON pes.`performer` = prs.`performer_id`
WHERE pes.`type` = 1
GROUP BY prs.`performer_id`, eve.`festival`
HAVING COUNT(*) > 2;
```

Ερώτημα 4

Για να βρούμε τον μέσο όρο αξιολογήσεων (ερμηνεία καλλιτεχνών, συνολική εντύπωση), ορίζουμε το όνομα του καλλιτέχνη για τον οποίο ενδιαφερόμαστε (π.χ. Billie Eilish), έπειτα βρίσκουμε με JOIN όλες τις παραστάσεις στις οποίες έχει συμμετάσχει και τέλος με JOIN στις κριτικές και χρήση του AVG() βρίσκουμε τον μέσο όρο των κριτικών για αυτές τις παραστάσεις.

```
SELECT
 AVG(r.`artist_performance`) AS `performance_rating`,
 AVG(r.`overall`) AS `overall_rating`
FROM `ratings` r
JOIN `performances` perfc
ON r.`performance` = perfc.`performance_id`
JOIN `performers` perfm
ON perfc.`performer` = perfm.`performer_id`
WHERE perfm.`name` = "Billie Eilish"
```

Indexes που διαθέτουμε:

- idx_performer_name ON performers(name)
- idx_fk_performer ON performances(performer)
- PRIMARY ON ratings(performance, client)

Query plan:

- Βρίσκουμε το id του performer με index scan στο name (ή sequential scan αν το table είναι 1 page) → my_performer_id.
- Κατασκευάζουμε Bitmap heap χρησιμοποιώντας τον index idx_fk_performer και βρίσκουμε τα pages του table performances που περιέχουν tuple με performer = my_performer_id.
- Βρίσκουμε όλα τα αντίστοιχα performance_ids απο τα relevant pages.
- Εκτελούμε Nested Loop → Για κάθε performance_id, με index scan στα ratings (χρησιμοποιώντας τον index PRIMARY), βρίσκουμε τις αντίστοιχες βαθμολογίες ανανεώνουμε κατάλληλα τις μεταβλητές του aggregation.

Ο λόγος που χρησιμοποιούμε Bitmap heap είναι ότι ο index idx_fk_performer δεν είναι clustered, με αποτέλεσμα το απλό index scan να οδηγούσε, ενδεχομένως, σε άσκοπα I/O operations.

Εκτέλεση του παραπάνω πλάνου:

```
Aggregate  (cost=19.97..19.98 rows=1 width=64) (actual time=0.229..0.231 rows=1 loops=1)
  ->  Nested Loop  (cost=4.66..19.67 rows=60 width=8) (actual time=0.076..0.206 rows=90 loops=1)
        ->  Nested Loop  (cost=4.38..14.17 rows=4 width=4) (actual time=0.055..0.081 rows=11 loops=1)
              ->  Seq Scan on performers perfm  (cost=0.00..4.50 rows=1 width=4)
                    (actual time=0.014..0.028 rows=1 loops=1)
              Filter: ((name)::text = 'Billie Eilish'::text)
              Rows Removed by Filter: 119
              ->  Bitmap Heap Scan on performances perfc  (cost=4.38..9.54 rows=13 width=8)
                    (actual time=0.037..0.047 rows=11 loops=1)
              Recheck Cond: (performer = perfm.performer_id)
              Heap Blocks: exact=5
              ->  Bitmap Index Scan on performances_perforemer  (cost=0.00..4.37 rows=13 width=0)
                    (actual time=0.030..0.030 rows=11 loops=1)
                    Index Cond: (performer = perfm.performer_id)
        ->  Index Scan using ratings_pkey on ratings r  (cost=0.28..1.24 rows=14 width=12)
                    (actual time=0.007..0.009 rows=8 loops=11)
              Index Cond: (performance = perfc.performance_id)
Planning Time: 1.928 ms
Execution Time: 0.410 ms
```

Εναλλακτικά, αν χρησιμοποιήσουμε Hash Join (+ 0 στο join condition):

```
Aggregate  (cost=182.19..182.20 rows=1 width=64) (actual time=0.910..0.912 rows=1 loops=1)
  ->  Hash Join  (cost=14.22..181.88 rows=60 width=8) (actual time=0.121..0.899 rows=90 loops=1)
        Hash Cond: ((r.performance + 0) = perfc.performance_id)
        ->  Seq Scan on ratings r  (cost=0.00..131.41 rows=7141 width=12) (actual time=0.012..0.341 rows=7141 loops=1)
        ->  Hash  (cost=14.17..14.17 rows=4 width=4) (actual time=0.049..0.050 rows=11 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 9kB
                ->  Nested Loop  (cost=4.38..14.17 rows=4 width=4) (actual time=0.025..0.045 rows=11 loops=1)
                ->  Seq Scan on performers perfm  (cost=0.00..4.50 rows=1 width=4)
                                (actual time=0.008..0.018 rows=1 loops=1)
                        Filter: ((name)::text = 'Billie Eilish'::text)
                        Rows Removed by Filter: 119
                ->  Bitmap Heap Scan on performances perfc  (cost=4.38..9.54 rows=13 width=8)
                                (actual time=0.015..0.023 rows=11 loops=1)
                        Recheck Cond: (performer = perfm.performer_id)
                        Heap Blocks: exact=5
                        ->  Bitmap Index Scan on performances_perforemer  (cost=0.00..4.37 rows=13 width=0)
                                        (actual time=0.009..0.009 rows=11 loops=1)
                                Index Cond: (performer = perfm.performer_id)
Planning Time: 0.255 ms
Execution Time: 0.956 ms
```

Παρατηρείται επιπλέον καθυστέρηση λόγω του sequential scan των ratings (7000 rows απο τα οποία μόνο 90 είναι relevant).

Τέλος, δοκιμάζουμε να εκτελέσουμε merge join (+0 στο join condition + disable hash join):

```
Aggregate  (cost=676.37..676.38 rows=1 width=64) (actual time=2.887..2.890 rows=1 loops=1)
  ->  Merge Join  (cost=602.71..675.59 rows=155 width=8) (actual time=2.270..2.873 rows=90 loops=1)
        Merge Cond: (((perfc.performance_id + 0)) = ((r.performance + 0)))
        ->  Sort  (cost=14.21..14.22 rows=4 width=4) (actual time=0.041..0.044 rows=11 loops=1)
                Sort Key: ((perfc.performance_id + 0))
                Sort Method: quicksort  Memory: 25kB
                ->  Nested Loop  (cost=4.38..14.17 rows=4 width=4) (actual time=0.021..0.036 rows=11 loops=1)
                ->  Seq Scan on performers perfm  (cost=0.00..4.50 rows=1 width=4)
                                (actual time=0.009..0.020 rows=1 loops=1)
                        Filter: ((name)::text = 'Billie Eilish'::text)
                        Rows Removed by Filter: 119
                ->  Bitmap Heap Scan on performances perfc  (cost=4.38..9.54 rows=13 width=8)
                                (actual time=0.010..0.012 rows=11 loops=1)
                        Recheck Cond: (performer = perfm.performer_id)
                        Heap Blocks: exact=1
                        ->  Bitmap Index Scan on performances_perforemer  (cost=0.00..4.37 rows=13 width=0)
                                        (actual time=0.005..0.005 rows=11 loops=1)
                                Index Cond: (performer = perfm.performer_id)
        ->  Sort  (cost=588.50..606.35 rows=7141 width=12) (actual time=2.154..2.416 rows=7075 loops=1)
                Sort Key: ((r.performance + 0))
                Sort Method: quicksort  Memory: 471kB
                ->  Seq Scan on ratings r  (cost=0.00..131.41 rows=7141 width=12)
                        (actual time=0.004..0.996 rows=7141 loops=1)
Planning Time: 0.333 ms
Execution Time: 2.997 ms
```

Παρατηρούμε ότι ακόμα και για clustered indexes το merge join χρειάζεται περισσότερο χρόνο, αφού εκτελεί αναπόφευκτα sequential scan στα ratings δύο φορές.

Ερώτημα 5

Αρχικά φτιάχνουμε έναν προσωρινό πίνακα ο οποίος περιέχει όλα τα ξεχωριστά ζευγάρια παράστασης - καλλιτέχνη (αν στην παράσταση συμμετέχει group τότε εμφανίζεται μια γραμμή για κάθε μέλος του group). Έπειτα από τα ζευγάρια αυτά φτιάχνουμε ακόμα έναν πίνακα ο οποίος τώρα υπολογίζει τον αριθμό των διακριτών εμφανίσεων του κάθε καλλιτέχνη ανά φεστιβάλ και περιορίζει την ηλικία των καλλιτεχνών ώστε να είναι μικρότερη από 30. Τέλος χρησιμοποιώντας την MAX() είναι εύκολο να βρούμε τους νέους καλλιτέχνες με τον μεγαλύτερο αριθμό εμφανίσεων στο φεστιβάλ.

```
WITH `event_artist` AS (
 SELECT DISTINCT
   p.`event`,
   COALESCE(ga.`artist`, p.`performer`) AS `artist`
 FROM `performances` p
 LEFT JOIN `group_artist` ga ON p.`performer` = ga.`group`
),
`young_artist_participation` AS (
 SELECT
   ea.`artist`,
   COUNT(DISTINCT e.`festival`) AS `participations`,
   pf.`name`,
   TIMESTAMPDIFF(YEAR, pf.`birth_date`, CURRENT_DATE) AS `age`
 FROM `event_artist` ea
 JOIN `events` e ON ea.`event` = e.`event_id`
 JOIN `performers` pf ON pf.`performer_id` = ea.`artist`
 WHERE TIMESTAMPDIFF(YEAR, pf.`birth_date`, CURRENT_DATE) < 30
 GROUP BY ea.`artist`, pf.`name`, pf.`birth_date`
),
`max_participation` AS (
 SELECT MAX(`participations`) AS `max_participations`
 FROM `young_artist_participation`
)
SELECT
 yap.`artist` AS `performer_id`,
 yap.`name`,
 yap.`age`,
 yap.`participations`
FROM `young_artist_participation` yap
JOIN `max_participation` mp ON yap.`participations` = mp.`max_participations`;
```

Ερώτημα 6

Για να βρούμε τις παραστάσεις που έχει παρακολουθήσει ένας επισκέπτης, βρίσκουμε όλα τα χρησιμοποιημένα εισιτήρια του επισκέπτη και από αυτά τις αντίστοιχες παραστάσεις. Επειτά για κάθε παράσταση βρίσκουμε τον μέσο όρο των αξιολογήσεων του επισκέπτη (αρχικά μέσος όρος ανα εμφάνιση και έπειτα μέσος όρος ανάμεσα σε όλες τις εμφανίσεις της κάθε παράστασης με aggregation).

```
SET @visitor = 5033;
SELECT
 ev.`event_id` AS `event_id`,
 ev.`name` AS `event_name`,
 CASE
  WHEN AVG((r.`artist_performance` + r.`sound_lighting` + r.`stage_presence`
            + r.`organization` + r.`overall`) / 5) IS NULL
  THEN 'no rating'
  ELSE CAST(AVG((r.`artist_performance` + r.`sound_lighting` + r.`stage_presence`
                 + r.`organization` + r.`overall`) / 5) AS CHAR)
 END AS average_rating
FROM `tickets` tk
JOIN `events` ev
  ON tk.`event` = ev.`event_id`
JOIN `performances` pf
  ON pf.`event` = tk.`event`
LEFT JOIN ratings r
  ON tk.`client` = r.`client` AND pf.`performance_id` = r.`performance`
WHERE tk.`client` = @visitor AND tk.`is_used` = TRUE
GROUP BY pf.`event`;
```

Χρήσιμα Indexes που διαθέτουμε:

- Idx_fk_client ON tickets(client, is_used, event)
- PRIMARY ON events(event_id)
- Idx_fk_event ON performances(event)
- PRIMARY ON ratings(performance, client)

Query plan:

- Φιλτράρουμε τα tickets με τον index idx_fk_client και βρίσκουμε αυτά για τα οποία client = @visitor και is_used = 1.
- Με Nested Loop και χρησιμοποιώντας τον PRIMARY index των events βρίσκουμε το αντίστοιχο event name για κάθε εισιτήριο που πέρασε το φιλτράρισμα του πρώτου βήματος.
- Με Nested Loop και χρησιμοποιώντας τον index idx_pk_events βρίσκουμε τα αντίστοιχα performances για κάθε event που υπολογίσαμε στα προηγούμενα 2 βήματα.
- Με Nested Loop και χρησιμοποιώντας τον index PRIMARY των ratings βρίσκουμε το αντίστοιχα rating για κάθε performance (αν δεν υπάρχει το πεδίο γίνεται NULL).
- Τέλος το αποτέλεσμα ομαδοποιείται ως προς event_id και γίνεται το aggregation.

Εκτέλεση του παραπάνω πλάνου:

```
GroupAggregate  (cost=0.99..60.91 rows=19 width=67) (actual time=0.043..0.085 rows=8 loops=1)
  Group Key: ev.event_id
  ->  Nested Loop Left Join  (cost=0.99..60.19 rows=19 width=55) (actual time=0.027..0.070 rows=20 loops=1)
        ->  Nested Loop  (cost=0.71..42.27 rows=19 width=43) (actual time=0.023..0.047 rows=20 loops=1)
              Join Filter: (tk.event = pf.event)
              ->  Nested Loop  (cost=0.43..37.96 rows=9 width=43) (actual time=0.017..0.027 rows=8 loops=1)
                    ->  Index Only Scan using idx_tickets_cl_used_ev on tickets tk  (cost=0.29..4.47 rows=9 width=8)
                                      (actual time=0.011..0.012 rows=8 loops=1)
                          Index Cond: ((client = 5033) AND (is_used = true))
                          Heap Fetches: 0
                    ->  Index Scan using events_pkey on events ev  (cost=0.14..3.72 rows=1 width=35)
                              (actual time=0.001..0.001 rows=1 loops=8)
                          Index Cond: (event_id = tk.event)
              ->  Index Scan using performances_event on performances pf  (cost=0.28..0.45 rows=2 width=8)
                              (actual time=0.001..0.002 rows=2 loops=8)
                    Index Cond: (event = ev.event_id)
        ->  Index Scan using idx_ratings_cl_perf on ratings r  (cost=0.28..0.93 rows=1 width=28)
                              (actual time=0.001..0.001 rows=0 loops=20)
              Index Cond: ((client = 5033) AND (performance = pf.performance_id))
```

Planning Time: 0.386 ms

Execution Time: 0.127 ms

Εναλλακτικό πλάνο με hash join (events, tickets_filtered) και hash aggregate:

```
HashAggregate  (cost=37.32..37.70 rows=19 width=67) (actual time=0.105..0.111 rows=8 loops=1)
  Group Key: ev.event_id
  Batches: 1  Memory Usage: 24kB
  ->  Nested Loop Left Join  (cost=5.14..36.98 rows=19 width=55) (actual time=0.026..0.097 rows=20 loops=1)
        ->  Nested Loop  (cost=4.86..19.06 rows=19 width=43) (actual time=0.022..0.075 rows=20 loops=1)
              Join Filter: (tk.event = pf.event)
              ->  Hash Join  (cost=4.58..14.75 rows=9 width=43) (actual time=0.019..0.060 rows=8 loops=1)
                    Hash Cond: (ev.event_id = tk.event)
                    ->  Seq Scan on events ev  (cost=0.00..9.50 rows=250 width=35)
                              (actual time=0.003..0.021 rows=250 loops=1)
                    ->  Hash  (cost=4.47..4.47 rows=9 width=8) (actual time=0.010..0.010 rows=8 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 9kB
                          ->  Index Only Scan using idx_tickets_cl_used_ev on tickets tk  (cost=0.29..4.47 rows=9 width=8)
                                      (actual time=0.006..0.007 rows=8 loops=1)
                                Index Cond: ((client = 5033) AND (is_used = true))
                                Heap Fetches: 0
              ->  Index Scan using performances_event on performances pf  (cost=0.28..0.45 rows=2 width=8)
                          (actual time=0.001..0.001 rows=2 loops=8)
                    Index Cond: (event = ev.event_id)
        ->  Index Scan using idx_ratings_cl_perf on ratings r  (cost=0.28..0.93 rows=1 width=28)
                          (actual time=0.001..0.001 rows=0 loops=20)
              Index Cond: ((client = 5033) AND (performance = pf.performance_id))
```

Planning Time: 0.495 ms

Execution Time: 0.146 ms

Ο χρόνος εκτέλεσης παραμένει σχεδόν ίδιος.

Τέλος εκτελούμε πλάνο με sorting και merge join μόνο:

```
GroupAggregate  (cost=198.88..199.64 rows=19 width=67) (actual time=0.495..0.502 rows=8 loops=1)
  Group Key: ev.event_id
  ->  Sort  (cost=198.88..198.92 rows=19 width=55) (actual time=0.488..0.490 rows=20 loops=1)
        Sort Key: ev.event_id
        Sort Method: quicksort  Memory: 26kB
        ->  Merge Left Join  (cost=198.05..198.47 rows=19 width=55) (actual time=0.479..0.484 rows=20 loops=1)
              Merge Cond: (pf.performance_id = ((r.performance + 0)))
              ->  Sort  (cost=30.00..30.05 rows=19 width=43) (actual time=0.117..0.118 rows=20 loops=1)
                    Sort Key: pf.performance_id
                    Sort Method: quicksort  Memory: 26kB
                    ->  Nested Loop  (cost=20.02..29.60 rows=19 width=43) (actual time=0.083..0.112 rows=20 loops=1)
                          Join Filter: (tk.event = pf.event)
                          ->  Merge Join  (cost=19.75..25.29 rows=9 width=43) (actual time=0.077..0.095 rows=8 loops=1)
                                Merge Cond: (tk.event = ev.event_id)
                                ->  Index Only Scan using idx_tickets_cl_used_ev on tickets tk
                                          (cost=0.29..4.47 rows=9 width=8) (actual time=0.015..0.016 rows=8 loops=1)
                                      Index Cond: ((client = 5033) AND (is_used = true))
                                      Heap Fetches: 0
                                ->  Sort  (cost=19.46..20.08 rows=250 width=35)
                                                (actual time=0.059..0.066 rows=229 loops=1)
                                      Sort Key: ev.event_id
                                      Sort Method: quicksort  Memory: 40kB
                                      ->  Seq Scan on events ev  (cost=0.00..9.50 rows=250 width=35)
                                                (actual time=0.007..0.028 rows=250 loops=1)
                          ->  Index Scan using performances_event on performances pf  (cost=0.28..0.45 rows=2 width=8)
                                      (actual time=0.001..0.002 rows=2 loops=8)
                                Index Cond: (event = ev.event_id)
              ->  Sort  (cost=168.05..168.14 rows=36 width=28) (actual time=0.361..0.361 rows=7 loops=1)
                    Sort Key: ((r.performance + 0))
                    Sort Method: quicksort  Memory: 25kB
                    ->  Seq Scan on ratings r  (cost=0.00..167.12 rows=36 width=28)
                                (actual time=0.029..0.359 rows=7 loops=1)
                          Filter: ((client + 0) = 5033)
                          Rows Removed by Filter: 7134
Planning Time: 0.517 ms
Execution Time: 0.540 ms
```

Ερώτημα 7

Βρίσκουμε τον μέσο όρο εμπειρίας του προσωπικού για κάθε φεστιβάλ χρησιμοποιώντας το attribute experience_sv, στο οποίο καταχωρείται ο βαθμός εμπειρίας που είχε ο αντίστοιχος υπάλληλος την περίοδο που εργάστηκε στο συγκεκριμένο φεστιβάλ. Ταξινομούμε τους μέσους όρους κατά αύξουσα σειρά και με το LIMIT 1 κρατάμε το φεστιβάλ με τον χαμηλότερο μέσο όρο εμπειρίας.

```
SELECT
 `festival`,
 AVG(`experience_sv`) AS `avg_festival_exp`
FROM `staff_venue`
GROUP BY `festival`
ORDER BY `avg_festival_exp`
LIMIT 1;
```

Ερώτημα 8

Χρησιμοποιούμε μία έγκυρη ημέρα (εντός των ορίων ενός φεστιβάλ) για να βρούμε τα μέλη από το προσωπικό υποστήριξης τα οποία δεν έχουν προγραμματισμένη εργασία. Κάνουμε τον πινακα staff LEFT JOIN με τον πίνακα των μελών του προσωπικού που δούλεψαν εκείνη την ημέρα σε κάποιο event, συνεπώς στον νέο πίνακα έχουμε NULL value στα records που μας ενδιαφέρουν.

```
SET @specific_date = '2022-08-04';

SELECT s.`staff_id`, s.`name`
FROM `staff` s
LEFT JOIN (
 SELECT DISTINCT sv.`staff_member`
 FROM `staff_venue` sv
 JOIN `events` e ON sv.`venue` = e.`venue`
 WHERE DATE(e.`date_time`) = @specific_date
) worked ON worked.`staff_member` = s.`staff_id`
WHERE worked.`staff_member` IS NULL
AND s.`role` = 3;
```

Ερώτημα 9

Φτιάχνουμε έναν προσωρινό πίνακα που κρατάει τον επισκέπτη, την χρονιά και τον αριθμό παραστάσεων που παρακολούθησε ο επισκέπτης εκείνο τον χρόνο. Έπειτα από τον πίνακα αυτό κρατάμε μόνο τους επισκέπτες που είχαν τον ίδιο αριθμό παρακολουθήσεων με κάποιον άλλο επισκέπτη τον ίδιο χρόνο και τυπώνουμε τις απαραίτητες πληροφορίες.

```
WITH `ClientAttendancesPerYearFiltered` AS (
 SELECT
   t.`client` AS `client_id`,
   YEAR(e.`date_time`) AS `event_year`,
   COUNT(t.`ticket_id`) AS `events_attended`
 FROM `tickets` t
   JOIN `events` e ON t.`event` = e.`event_id`
 WHERE t.`is_used` = true
 GROUP BY t.`client`, YEAR(e.`date_time`)
 HAVING COUNT(t.`ticket_id`) > 3
),
`WithDuplicates` AS (
 SELECT
   caf.*,
   COUNT(*) OVER (
     PARTITION BY `event_year`, `events_attended`
   ) AS `same_count_clients`
 FROM `ClientAttendancesPerYearFiltered` caf
)
SELECT
 caf.`event_year`,
 caf.`events_attended`,
 caf.`client_id`,
 c.`name` AS `client1_name`
FROM `WithDuplicates` caf
JOIN `clients` c ON c.`client_id` = caf.`client_id`
WHERE `same_count_clients` > 1
ORDER BY
 caf.`event_year`,
 caf.`events_attended`,
 `client_id`;
```

Ερώτημα 10

Με aggregation βρίσκουμε το πλήθος των παραστάσεων, στις οποίες εμφανίζονται καλλιτέχνες που ανήκουν σε ένα ζεύγος ειδών. Έπειτα, με ταξινόμηση βρίσκουμε τα τρία ζεύγη με τις περισσότερες εμφανίσεις σε φεστιβάλ.

```
SELECT
 LEAST(ps1.`genre_name`, ps2.`genre_name`) AS `genre1`,
 GREATEST(ps1.`genre_name`, ps2.`genre_name`) AS `genre2`,
 COUNT(DISTINCT per.`performer_id`) AS `pair_count`
FROM `performers` per
JOIN `performers_genres` ps1
 ON per.`performer_id` = ps1.`performer_id`
JOIN `performers_genres` ps2
 ON per.`performer_id` = ps2.`performer_id`
WHERE ps1.`genre_name` < ps2.`genre_name`
GROUP BY genre1, genre2
ORDER BY `pair_count` DESC
LIMIT 3;
```

Ερώτημα 11

Βρίσκουμε το πλήθος των φεστιβάλ, στα οποία έχει συμμετάσχει κάθε καλλιτέχνης, μετρώντας το πλήθος των διαφορετικών festival ids ανάμεσα σε όλες τις παραστάσεις στις οποίες έχει εμφανιστεί.
Έπειτα, βρίσκουμε την μέγιστη τιμή των εμφανίσεων και κρατάμε τους καλλιτέχνες που έχουν εμφανιστεί σε φεστιβάλ το πολύ πέντε φορές λιγότερο από τη μέγιστη τιμή.

```
WITH `participation_counts` AS (
 SELECT
   `performer`,
   COUNT(DISTINCT eve.`festival`) AS `participation_count`
 FROM `performances` p
 JOIN `events` eve
 ON eve.`event_id` = p.`event`
 GROUP BY `performer`
),
`max_participation` AS (
 SELECT MAX(`participation_count`) AS `max_count`
 FROM `participation_counts`
)
SELECT
 per.`performer_id`,
 per.`name`,
 pc.`participation_count`
FROM `participation_counts` pc
JOIN `performers` per ON per.`performer_id` = pc.`performer`
JOIN `max_participation` mp ON 1 = 1
WHERE pc.`participation_count` <= mp.`max_count` - 5;
```

Ερώτημα 12

Για κάθε ημέρα του φεστιβάλ, υπολογίζουμε το πλήθος των υπαλλήλων που χρειάζονται ως εξής:

- Βρίσκουμε ποιά venues είναι ενεργά εκείνη τη μέρα, δηλαδή έχουν προγραμματισμένη παράσταση για εκείνη τη μέρα
- Βρίσκουμε τους υπαλλήλους που χρειάζονται τα αντίστοιχα venues
- Αθροίζουμε τα αποτελέσματα.

```
SELECT
 e.`festival` AS `festival_id`,
 DATE(e.`date_time`) AS `festival_day`,
 sr.`role` AS `staff_role`,
 CASE
   WHEN sr.`id` = 1 THEN SUM(CEIL(v.`capacity` * 0.03))
   WHEN sr.`id` = 2 THEN SUM(CEIL(v.`capacity` * 0.05))
   WHEN sr.`id` = 3 THEN SUM(CEIL(v.`capacity` * 0.02))
   ELSE 0
 END AS `staff_count`
FROM `events` e
JOIN `venues` v ON e.`venue` = v.`venue_id`
JOIN `staff_roles` sr
GROUP BY
 e.`festival`,
 DATE(e.`date_time`),
 sr.`role`
ORDER BY
 `festival_id`,
 `festival_day`,
 `staff_role`;
```

Ερώτημα 13

Με διαδοχικά joins βρίσκουμε την ήπειρο στην οποία έχει διεξαχθεί κάθε εμφάνιση. Έπειτα, για κάθε καλλιτέχνη βρίσκουμε το πλήθος των διαφορετικών ηπείρων στις οποίες έχει εμφανιστεί και κρατάμε αυτούς που έχουν εμφανιστεί σε πάνω από τρεις.

```
SELECT perf.`performer_id`, perf.`name`
FROM `performances` per
JOIN `performers` perf ON per.`performer` = perf.`performer_id`
JOIN `events` eve ON per.`event` = eve.`event_id`
JOIN `festivals` f ON eve.`festival` = f.`festival_id`
JOIN `locations` loc ON f.`location` = loc.`location_id`
GROUP BY perf.`performer_id`, perf.`name`
HAVING COUNT(DISTINCT loc.`continent`) > 2;
```

Ερώτημα 14

Βρίσκουμε τον αριθμό των εμφανίσεων ανά είδος μουσικής και χρονιάς με τον περιορισμό ο αριθμός αυτός να είναι τουλάχιστον 3. Έπειτα από τον πίνακα αυτό επιλέγουμε τις γραμμές για τις οποίες ισχύει ότι:

- Ο αριθμός εμφανίσεων είναι ίδιος.
- Το είδος μουσικής είναι ίδιο.
- Οι χρονιές για τις οποίες ο αριθμός εμφανίσεων είναι ίδιος,να είναι διαδοχικές.

```
WITH `performance_counts` AS (
 SELECT
   pg.`genre_name` AS `genre_name`,
   YEAR(p.`date_time`) AS year,
   COUNT(DISTINCT p.`performance_id`) AS `appearances`
 FROM `performances` p
 JOIN `performers_genres` pg
   ON p.`performer` = pg.`performer_id`
 GROUP BY pg.genre_name, YEAR(p.date_time)
 HAVING COUNT(*) >= 3
)
SELECT
 pc1.`genre_name`,
 pc1.`year` AS `year1`,
 pc2.`year` AS `year2`,
 pc1.`appearances`
FROM
 `performance_counts` pc1
JOIN
 `performance_counts` pc2
 ON pc1.`genre_name` = pc2.`genre_name`
 AND pc2.`year` = pc1.`year` + 1
 AND pc1.`appearances` = pc2.`appearances`
ORDER BY `year1`, pc1.`genre_name`;
```

Ερώτημα 15

Υπολογίζουμε τον μέσο όρο τον αξιολογήσεων ανά επισκέπτη και ανά performer, ταξινομούμε τα αποτελέσματα κατά φθίνουσα σειρά με και μέσω του LIMIT 5 κρατάμε τα πρώτα 5 στοιχεία. Η ισοδυναμία σε βαθμολογίες λύνεται με βάση τον αριθμό αξιολογήσεων του κάθε επισκέπτη για τον αντίστοιχο καλλιτέχνη και έπειτα με βάση το όνομα του επισκέπτη.

```
SELECT
 c.`name` AS `client_name`,
 pf.`name` AS `performer_name`,
 AVG(r.`artist_performance`) AS `rating_score`,
 COUNT(*) AS `num_of_ratings`
FROM `clients` c
JOIN `ratings` r ON c.`client_id` = r.`client`
JOIN `performances` p ON r.`performance` = p.`performance_id`
JOIN `performers` pf ON p.`performer` = pf.`performer_id`
GROUP BY c.`name`, pf.`name`
ORDER BY
 `rating_score` DESC,
 `num_of_ratings` DESC,
 c.`name`
LIMIT 5;
```

```sql
/*------------------*/
/*----- TABLES -----*/
/*------------------*/

CREATE TABLE `locations` (
 `location_id` INT AUTO_INCREMENT PRIMARY KEY,
 `address` VARCHAR(100) NOT NULL,
 `city` VARCHAR(30) NOT NULL,
 `country` VARCHAR(50) NOT NULL,
 `continent` VARCHAR(20) NOT NULL,
 `coords_long` FLOAT NOT NULL,
 `coords_lat` FLOAT NOT NULL,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `festivals` (
 `festival_id` INT AUTO_INCREMENT PRIMARY KEY,
 `first_day` DATE NOT NULL,
 `last_day` DATE NOT NULL,
 `poster` VARCHAR(70),
 `poster_desc` VARCHAR(200),
 `location` INT NOT NULL,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 KEY `idx_fk_location` (`location`),
 CONSTRAINT `fk_festival_location`
   FOREIGN KEY (`location`)
   REFERENCES `locations` (`location_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `venues` (
 `venue_id` INT AUTO_INCREMENT PRIMARY KEY,
 `name` VARCHAR(50) NOT NULL,
 `description` VARCHAR(250),
 `capacity` INT NOT NULL,
 `equipment` VARCHAR(250) NOT NULL,
 `picture` VARCHAR(70),
 `picture_desc` VARCHAR(200),
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `events` (
 `event_id` INT AUTO_INCREMENT PRIMARY KEY,
 `name` VARCHAR(50),
 `date_time` DATETIME NOT NULL,
 `duration` TIME NOT NULL,
 `poster` VARCHAR(70),
 `poster_desc` VARCHAR(200),
 `venue` INT NOT NULL,
 `festival` INT NOT NULL,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 KEY `idx_fk_venue` (`venue`),
 KEY `idx_fk_festival` (`festival`),
 KEY `idx_events_id_date` (`event_id`, `date_time`),
 CONSTRAINT `fk_event_venue`
   FOREIGN KEY (`venue`)
   REFERENCES `venues` (`venue_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_event_festival`
   FOREIGN KEY (`festival`)
   REFERENCES `festivals` (`festival_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `performers` (
 `performer_id` INT AUTO_INCREMENT PRIMARY KEY,
 `is_group` BOOLEAN NOT NULL,
 `name` VARCHAR(40) NOT NULL,
 `stage_name` VARCHAR(40),
 `birth_date` DATE,
 `website` varchar(40),
 `instagram` varchar(70),
 `picture` VARCHAR(70),
 `picture_desc` VARCHAR(200),
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 KEY `idx_performer_name` (`name`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `genres` (
 `genre_id` INT AUTO_INCREMENT PRIMARY KEY,
 `name` VARCHAR(100) NOT NULL,
 UNIQUE KEY `idx_genre_name` (`name`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `subgenres` (
 `subgenre_id` INT AUTO_INCREMENT PRIMARY KEY,
 `genre_id` INT NOT NULL,
 `name` VARCHAR(100) NOT NULL,
 UNIQUE KEY `idx_fk_genre` (`genre_id`, `name`),
 CONSTRAINT `fk_subgen_gen`
   FOREIGN KEY (`genre_id`)
   REFERENCES `genres`(`genre_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `performer_subgenres` (
 `performer_id` INT,
 `subgenre_id` INT,
 PRIMARY KEY (`performer_id`, `subgenre_id`),
 KEY `idx_fk_subgenre` (`subgenre_id`),
 CONSTRAINT `fk_perfsub_per`
   FOREIGN KEY (`performer_id`)
   REFERENCES `performers`(`performer_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_perfsub_subgen`
   FOREIGN KEY (`subgenre_id`)
   REFERENCES `subgenres`(`subgenre_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `group_artist` (
 `artist` INT NOT NULL,
 `group` INT NOT NULL,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 PRIMARY KEY (`group`, `artist`),
 KEY `idx_fk_artist` (`artist`),
 CONSTRAINT `fk_ga_artist`
   FOREIGN KEY (`artist`)
   REFERENCES `performers` (`performer_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_ga_group`
   FOREIGN KEY (`group`)
   REFERENCES `performers` (`performer_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `performance_types` (
 `id` INT AUTO_INCREMENT PRIMARY KEY,
 `name` VARCHAR(20) NOT NULL
) ENGINE = InnoDB DEFAULT CHARSET = utf8;
```

```sql
CREATE TABLE `performances` (
 `performance_id` INT AUTO_INCREMENT PRIMARY KEY,
 `sequence_num` INT NOT NULL,
 `date_time` DATETIME NOT NULL,
 `duration` TIME NOT NULL,
 `event` INT NOT NULL,
 `performer` INT NOT NULL,
 `type` INT NOT NULL,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 KEY `idx_fk_event` (`event`),
 KEY `idx_fk_performer` (`performer`),
 CONSTRAINT `fk_performance_event`
   FOREIGN KEY (`event`)
   REFERENCES `events` (`event_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_performance_performer`
   FOREIGN KEY (`performer`)
   REFERENCES `performers` (`performer_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_performances_types`
   FOREIGN KEY (`type`)
   REFERENCES `performance_types` (`id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `check_performance_duration` CHECK (
   `duration` > '00:00:00'
   AND `duration` <= '03:00:00'
 )
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `clients` (
 `client_id` INT AUTO_INCREMENT PRIMARY KEY,
 `name` VARCHAR(100) NOT NULL,
 `birth_date` DATE NOT NULL,
 `email` VARCHAR(50),
 `phone` VARCHAR(15),
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `staff_roles` (
 `id` INT AUTO_INCREMENT PRIMARY KEY,
 `role` VARCHAR(15) NOT NULL
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `staff` (
 `staff_id` INT AUTO_INCREMENT PRIMARY KEY,
 `name` VARCHAR(40) NOT NULL,
 `birth_date` DATE NOT NULL,
 `role` INT NOT NULL,
 `experience` INT NOT NULL,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 KEY `idx_staff_role` (`role`),
 CONSTRAINT `fk_staff_role`
   FOREIGN KEY (`role`)
   REFERENCES `staff_roles` (`id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `check_experience_rate` CHECK (
   `experience` >= 0
   AND `experience` <= 5
 )
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `staff_venue` (
 `staff_member` INT NOT NULL,
 `venue` INT NOT NULL,
 `festival` INT NOT NULL,
 `experience_sv` INT NOT NULL,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 PRIMARY KEY (`festival`, `staff_member`),
 KEY `idx_fk_staff` (`staff_member`),
 KEY `idx_fk_venue` (`venue`),
 CONSTRAINT `fk_sv_staff`
   FOREIGN KEY (`staff_member`)
   REFERENCES `staff` (`staff_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_sv_venue`
   FOREIGN KEY (`venue`)
   REFERENCES `venues` (`venue_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_sv_festival`
   FOREIGN KEY (`festival`)
   REFERENCES `festivals` (`festival_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `check_experience_sv_rate` CHECK (
   `experience_sv` >= 0
   AND `experience_sv` <= 5
 )
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `ticket_categories` (
 `id` INT AUTO_INCREMENT PRIMARY KEY,
 `category` VARCHAR(20) NOT NULL
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `payment_methods` (
 `id` INT AUTO_INCREMENT PRIMARY KEY,
 `method` VARCHAR(20) NOT NULL
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `tickets` (
 `ticket_id` INT AUTO_INCREMENT PRIMARY KEY,
 `code` VARCHAR(13) DEFAULT NULL,
 `category` INT,
 `event` INT NOT NULL,
 `client` INT NOT NULL,
 `payment_method` INT NOT NULL,
 `is_used` BOOLEAN DEFAULT FALSE,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 KEY `idx_event_category_payment`
  (`event`, `category`, `payment_method`),
 KEY `idx_fk_client` (`client`, `is_used`, `event`),
 UNIQUE (`client`, `event`),
 CONSTRAINT `fk_ticket_event`
   FOREIGN KEY (`event`)
   REFERENCES `events` (`event_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_ticket_category`
   FOREIGN KEY (`category`)
   REFERENCES `ticket_categories` (`id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_ticket_client`
   FOREIGN KEY (`client`)
   REFERENCES `clients` (`client_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_ticket_pmethod`
   FOREIGN KEY (`payment_method`)
   REFERENCES `payment_methods` (`id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;
```

```sql
CREATE TABLE `ticket_costs` (
 `event` INT NOT NULL,
 `category` INT NOT NULL,
 `cost` NUMERIC(5, 2) NOT NULL,
 PRIMARY KEY (`event`, `category`),
 CONSTRAINT `fk_costs_event`
   FOREIGN KEY (`event`)
   REFERENCES `events` (`event_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_costs_category`
   FOREIGN KEY (`category`)
   REFERENCES `ticket_categories` (`id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `resale_waitlist` (
 `event` INT NOT NULL,
 `client` INT NOT NULL,
 `category` INT NOT NULL,
 `payment_method` INT NOT NULL,
 `added_at` TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP,
 PRIMARY KEY (`client`, `event`),
 KEY `idx_event_category_w1` (`event`, `category`),
 KEY `idx_rw_time`(`added_at`),
 CONSTRAINT `fk_waitlist_event`
   FOREIGN KEY (`event`)
   REFERENCES `events` (`event_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_waitlist_client`
   FOREIGN KEY (`client`)
   REFERENCES `clients` (`client_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_waitlist_category`
   FOREIGN KEY (`category`)
   REFERENCES `ticket_categories` (`id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_waitlist_method`
   FOREIGN KEY (`payment_method`)
   REFERENCES `payment_methods` (`id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `resale_queue` (
 `ticket` INT PRIMARY KEY,
 `listed_at` TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP,
 KEY `idx_rq_time` (`listed_at`),
 CONSTRAINT `fk_resale_ticket`
   FOREIGN KEY (`ticket`)
   REFERENCES `tickets` (`ticket_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `ratings` (
 `artist_performance` INT NOT NULL,
 `sound_lighting` INT NOT NULL,
 `stage_presence` INT NOT NULL,
 `organization` INT NOT NULL,
 `overall` INT NOT NULL,
 `client` INT NOT NULL,
 `performance` INT NOT NULL,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 PRIMARY KEY (`performance`, `client`),
 KEY `idx_fk_performance_r` (`client`, `performance`),
 CONSTRAINT `fk_rating_client`
   FOREIGN KEY (`client`)
   REFERENCES `clients` (`client_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_rating_performance`
   FOREIGN KEY (`performance`)
   REFERENCES `performances` (`performance_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `check_rating_artist_perf` CHECK (
   `artist_performance` >= 1
   AND `artist_performance` <= 5
 ),
 CONSTRAINT `check_rating_sound_lighting` CHECK (
   `sound_lighting` >= 1
   AND `sound_lighting` <= 5
 ),
 CONSTRAINT `check_rating_stage_presence` CHECK (
   `stage_presence` >= 1
   AND `stage_presence` <= 5
 ),
 CONSTRAINT `check_rating_organization` CHECK (
   `organization` >= 1
   AND `organization` <= 5
 ),
 CONSTRAINT `check_rating_overall` CHECK (
   `overall` >= 1
   AND `overall` <= 5
 )
) ENGINE = InnoDB DEFAULT CHARSET = utf8;

CREATE TABLE `ticket_transactions` (
 `transaction_id` INT AUTO_INCREMENT PRIMARY KEY,
 `ticket` INT NOT NULL,
 `seller` INT NOT NULL,
 `buyer` INT NOT NULL,
 `last_update` TIMESTAMP
   DEFAULT CURRENT_TIMESTAMP
   ON UPDATE CURRENT_TIMESTAMP,
 CONSTRAINT `fk_transaction_ticket`
   FOREIGN KEY (`ticket`)
   REFERENCES `tickets` (`ticket_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_transaction_seller`
   FOREIGN KEY (`seller`)
   REFERENCES `clients` (`client_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE,
 CONSTRAINT `fk_transaction_buyer`
   FOREIGN KEY (`buyer`)
   REFERENCES `clients` (`client_id`)
   ON DELETE RESTRICT
   ON UPDATE CASCADE
) ENGINE = InnoDB DEFAULT CHARSET = utf8;


/*--------------------*/
/*------ VIEWS ------*/
/*--------------------*/

/**
 * Holds information about the genre of each performer.
 */
CREATE VIEW `performers_genres` AS
SELECT
 perf.`performer_id`,
 perf.`name` AS performer_name,
 gen.`name` AS genre_name
FROM `performers` perf
JOIN `performer_subgenres` perf_sub
 ON perf.`performer_id` = perf_sub.`performer_id`
JOIN `subgenres` sub
 ON sub.`subgenre_id` = perf_sub.`subgenre_id`
JOIN `genres` gen
 ON gen.`genre_id` = sub.`genre_id`;
```

```
/*--------------------*/
/*---- FUNCTIONS ----*/
/*--------------------*/

DELIMITER ||

/**
 * Function to check if two time frames (a and b) overlap.
 */
CREATE FUNCTION `check_time_overlap`(
 a_start DATETIME,
 a_durat TIME,
 b_start DATETIME,
 b_durat TIME
)
RETURNS TINYINT(1)
BEGIN
 RETURN NOT (
   a_start >= ADDTIME(b_start, b_durat)
   OR ADDTIME(a_start, a_durat) <= b_start
 );
END ||

/**
 * Generates an EAN-13 code from a given integer.
 * Code from:
https://gist.github.com/Den2016/2b1f6e1badc97bbc58e8e424766fe437
 */
CREATE FUNCTION `gen_ean13`(`code` INT)
RETURNS VARCHAR(13)
LANGUAGE SQL NOT DETERMINISTIC
NO SQL SQL SECURITY INVOKER
BEGIN
  ...
END ||


/*--------------------*/
/*---- PROCEDURES ----*/
/*--------------------*/

/**
* Creates a sale of a ticket based on the given info.
*/
CREATE PROCEDURE `purchase_ticket` (
 IN `client_id` INT,
 IN `event_id` INT,
 IN `category` INT,
 IN `payment_method` INT
)
BEGIN
 DECLARE allowed INT DEFAULT 0;
 DECLARE sold INT DEFAULT 0;
 DECLARE num_tickets INT;
 DECLARE event_capacity INT;
 DECLARE generated_code VARCHAR(13);
 DECLARE has_ticket INT DEFAULT 0;

 -- Step 1: Check if client already has a ticket for this
event
 SELECT COUNT(*) INTO has_ticket
 FROM tickets
 WHERE client = client_id AND event = event_id;

 IF has_ticket > 0 THEN
   SIGNAL SQLSTATE '45001'
     SET MESSAGE_TEXT =
       'Client already owns a ticket for this event.';
 END IF;

 -- Step 2: Get event capacity
 SELECT ven.capacity INTO event_capacity
 FROM events evn
 JOIN venues ven ON evn.venue = ven.venue_id
 WHERE evn.event_id = event_id;

 -- Step 3: Determine category share
 SET allowed = CASE
   WHEN category = 1
     THEN event_capacity -
       CEIL(event_capacity*0.1) - CEIL(event_capacity*0.05)
   WHEN category = 2 THEN CEIL(event_capacity*0.1)
   WHEN category = 3 THEN CEIL(event_capacity*0.05)
   ELSE 0.00
 END;

 -- Step 4: Count how many tickets already sold for this
 -- event + category
 SELECT COUNT(*) INTO sold
 FROM tickets tk
 WHERE tk.event = event_id AND tk.category = category;

 -- Step 5: Check if sold out
 IF sold >= allowed THEN
   SIGNAL SQLSTATE '45002'
     SET MESSAGE_TEXT = 'Tickets sold out for this category.';
 END IF;

 -- Step 6: Insert ticket
 START TRANSACTION;

 INSERT INTO tickets (
   category, event, client, payment_method
 )
 VALUES (
   category,
   event_id,
   client_id,
   payment_method
 );

 -- Get the last inserted ticket's ID
 SET @last_ticket_id = LAST_INSERT_ID();

 UPDATE tickets
 SET code = gen_ean13(@last_ticket_id)
 WHERE ticket_id = @last_ticket_id;

 COMMIT;
END ||


/**
 * Adds a ticket to the resale queue.
 */
CREATE PROCEDURE `list_for_resale` (IN `p_ticket_id` INT)
BEGIN
 DECLARE v_event_id INT;
 DECLARE v_category INT;
 DECLARE v_is_used BOOLEAN;
 DECLARE v_sold INT DEFAULT 0;
 DECLARE v_event_capacity INT;
 DECLARE v_buyer INT DEFAULT NULL;
 DECLARE v_seller INT;

 -- Get ticket details
 SELECT t.event, t.category, t.is_used, t.client
 INTO v_event_id, v_category, v_is_used, v_seller
 FROM tickets t
 WHERE t.ticket_id = p_ticket_id;

 -- Check if the ticket was found
 IF v_event_id IS NULL THEN
   SIGNAL SQLSTATE '45000'
     SET MESSAGE_TEXT = 'Ticket does not exist.';
 END IF;

 -- Check if ticket has already been used
 IF v_is_used = TRUE THEN
   SIGNAL SQLSTATE '45001'
     SET MESSAGE_TEXT = 'Used ticket cannot be resold.';
 END IF;
```

```
 -- Get event capacity
 SELECT ven.capacity
 INTO v_event_capacity
 FROM events evn
 JOIN venues ven ON evn.venue = ven.venue_id
 WHERE evn.event_id = v_event_id;

 -- Count sold tickets in that category
 SELECT COUNT(*)
 INTO v_sold
 FROM tickets tk
 WHERE tk.event = v_event_id;

 -- Check if tickets are sold out
 IF v_sold < v_event_capacity THEN
   SIGNAL SQLSTATE '45002'
     SET MESSAGE_TEXT =
       'Tickets are not sold out for this event.';
 END IF;

 START TRANSACTION;

 -- Get first buyer from waitlist
 SELECT rw.client
 INTO v_buyer
 FROM resale_waitlist rw
 WHERE rw.category = v_category AND rw.event = v_event_id
 ORDER BY rw.added_at ASC
 LIMIT 1
 FOR UPDATE;

 -- If a buyer is found, perform transaction
 IF v_buyer IS NOT NULL THEN

   INSERT INTO `ticket_transactions`
     (`ticket`, `buyer`, `seller`)
   VALUES (p_ticket_id, v_buyer, v_seller);

   UPDATE `tickets` t
   SET t.`client` = v_buyer
   WHERE t.`ticket_id` = p_ticket_id;

   DELETE FROM `resale_waitlist`
   WHERE client = v_buyer
     AND category = v_category
     AND event = v_event_id;

   COMMIT;
 ELSE
   ROLLBACK;

   -- If no buyer, add ticket to resale queue
   INSERT INTO `resale_queue` (`ticket`)
   VALUES (p_ticket_id);
 END IF;
END ||


/**
 * Adds a customer to the waiting list for a ticket resale
 * for a specific event and ticket category.
 */
CREATE PROCEDURE `purchase_from_resale`(
 IN `p_client_id` INT,
 IN `p_event_id` INT,
 IN `p_category_id` INT,
 IN `p_payment_method_id` INT
)
BEGIN
 DECLARE v_ticket_id INT DEFAULT NULL;
 DECLARE v_seller_id INT;
 DECLARE has_ticket INT DEFAULT 0;
 DECLARE v_event_capacity INT;
 DECLARE v_sold INT DEFAULT 0;

 -- Check if client already has a ticket for this event
 SELECT COUNT(*) INTO has_ticket
 FROM tickets
 WHERE client = p_client_id AND event = p_event_id;

 IF has_ticket > 0 THEN
   SIGNAL SQLSTATE '45001'
     SET MESSAGE_TEXT =
       'Client already owns a ticket for this event.';
 END IF;

 -- Get event capacity
 SELECT ven.capacity
 INTO v_event_capacity
 FROM events evn
 JOIN venues ven ON evn.venue = ven.venue_id
 WHERE evn.event_id = p_event_id;

 -- Count sold tickets in that category
 SELECT COUNT(*)
 INTO v_sold
 FROM tickets tk
 WHERE tk.event = p_event_id;

 -- Check if tickets are sold out
 IF v_sold < v_event_capacity THEN
   SIGNAL SQLSTATE '45002'
     SET MESSAGE_TEXT =
       'Tickets are not sold out for this event.';
 END IF;

 START TRANSACTION;

 -- Try to get a ticket from resale queue
 SELECT rq.ticket, t.client
 INTO v_ticket_id, v_seller_id
  FROM resale_queue rq
 JOIN tickets t
 ON t.ticket_id = rq.ticket
   WHERE t.event = p_event_id AND t.category = p_category_id
 ORDER BY rq.listed_at ASC
 LIMIT 1
 FOR UPDATE;

 -- If ticket found, process transaction
 IF v_ticket_id IS NOT NULL THEN

   -- Get the current owner of the ticket
   INSERT INTO ticket_transactions
     (`ticket`, `buyer`, `seller`)
   VALUES
     (v_ticket_id, p_client_id, v_seller_id);

   -- Update ticket ownership
   UPDATE tickets t
   SET t.client = p_client_id,
       t.payment_method = p_payment_method_id
   WHERE t.ticket_id = v_ticket_id;

   -- Remove ticket from resale queue
   DELETE FROM `resale_queue`
   WHERE `ticket` = v_ticket_id;

   COMMIT;
 ELSE
   ROLLBACK;
   -- Rollback the transaction as no ticket was processed
   -- If no ticket found, insert client into resale waitlist
   INSERT INTO resale_waitlist
     (event, client, category, payment_method)
   VALUES
     (p_event_id, p_client_id,
      p_category_id, p_payment_method_id);
 END IF;
END ||
```

```
/*------------------*/
/*---- TRIGGERS ----*/
/*------------------*/

/**
 * Checks if a new added festival is on the same
 * location as last year.
 */
CREATE TRIGGER `different_location_each_year`
BEFORE INSERT ON `festivals`
FOR EACH ROW
BEGIN
 DECLARE prev_location INT;

 SELECT `location` INTO prev_location
 FROM festivals
 ORDER BY festival_id DESC
 LIMIT 1;

 IF prev_location = NEW.location THEN
   SIGNAL SQLSTATE '45000'
     SET MESSAGE_TEXT =
       'location should be different each year';
 END IF;
END ||


/**
 * Checks if a new added performance is within the event's
 * time period and if the break durations are correct.
 */
CREATE TRIGGER `check_performance_time`
BEFORE INSERT ON `performances`
FOR EACH ROW
BEGIN
 DECLARE event_start DATETIME;
 DECLARE event_end DATETIME;
 DECLARE performance_start DATETIME;
 DECLARE performance_end DATETIME;
 DECLARE break_before TIME DEFAULT NULL;

 -- Check if performance fits within event time range
 SELECT `date_time`,ADDTIME(`date_time`, `duration`)
 INTO event_start, event_end
 FROM `events`
 WHERE event_id = NEW.`event`;

 SET performance_start = NEW.date_time;
 SET performance_end = ADDTIME(NEW.`date_time`,
NEW.`duration`);

 IF performance_start < event_start
   OR performance_end > event_end THEN
   SIGNAL SQLSTATE '45000'
     SET MESSAGE_TEXT =
       'Performance time is outside the bounds of the event';
 END IF;

 -- Check constraints for the break before the performance
 SELECT TIMEDIFF(NEW.date_time, p.date_time + p.duration)
   INTO break_before
 FROM performances p
 WHERE p.event = NEW.event
 AND p.sequence_num = NEW.sequence_num - 1;
  IF NEW.sequence_num > 1 THEN
   IF break_before IS NULL THEN
     SIGNAL SQLSTATE '45000'
       SET MESSAGE_TEXT =
         'previous performance of this event is missing';
   ELSEIF break_before < '00:05:00'
     OR break_before > '00:30:00' THEN
     SIGNAL SQLSTATE '45000'
       SET MESSAGE_TEXT =
         'break duration should be between 5 and 30 mins';
   END IF;
 END IF;
END ||
```

```
/**
 * Checks if the date of the added event is within the
 * dates of the festival
 */
CREATE TRIGGER `check_event_date_within_festival`
BEFORE INSERT ON `events`
FOR EACH ROW
BEGIN
 DECLARE fest_start DATE;
 DECLARE fest_end DATE;

 SELECT first_day, last_day
 INTO fest_start, fest_end
 FROM festivals
 WHERE festival_id = new.festival;

 -- Check if the event date is within the festival's range
 IF DATE(NEW.`date_time`) < fest_start OR
DATE(NEW.`date_time`) > fest_end THEN
   SIGNAL SQLSTATE '45000'
     SET MESSAGE_TEXT =
       'Event date must be within the corresponding
        festival dates';
 END IF;
END ||

/**
 * Checks if a newly added event in the "events" table
 * overlaps with an existing event in the same venue.
 */
CREATE TRIGGER `deny_overlapping_events`
BEFORE INSERT ON `events`
FOR EACH ROW BEGIN
 IF EXISTS (
   SELECT 1
   FROM `events` AS evn
   WHERE
     evn.`venue` = new.`venue`
     AND check_time_overlap(
       NEW.`date_time`,
       NEW.`duration`,
       evn.`date_time`,
       evn.`duration`
     )
   LIMIT 1
 ) THEN
   SIGNAL SQLSTATE '45000'
   SET MESSAGE_TEXT =
     'The new event overlaps with an existing one.';
 END IF;
END ||
```

```
/**
 * Checks if newly added performances contain performers that
 * appear in other performances at the same time.
 */
CREATE TRIGGER `check_artist_availability`
BEFORE INSERT ON `performances`
FOR EACH ROW BEGIN
 IF EXISTS (
   SELECT 1
   FROM `performances` AS per
   WHERE
     per.`performer` IN (
       SELECT new.`performer` AS `performer`
       UNION
       (
         SELECT `group_artist`.`artist` AS `performer`
         FROM `group_artist`
         WHERE `group_artist`.`group` = new.`performer`
       )
       UNION
       (
         SELECT `group_artist`.`group` AS `performer`
         FROM `group_artist`
         WHERE
           `group_artist`.`artist` IN (
             SELECT `group_artist`.`artist` AS `performer`
             FROM `group_artist`
             WHERE `group_artist`.`group` = new.`performer`
           )
           OR `group_artist`.`artist` = new.`performer`
       )
     )
     AND check_time_overlap(
       new.`date_time`,
       new.`duration`,
       per.`date_time`,
       per.`duration`
     )
   LIMIT 1
 ) THEN
   SIGNAL SQLSTATE '45000' SET
     MESSAGE_TEXT =
       'Performer is not available at the selected time.';
 END IF;
END ||

/**
 * Checks if the added entries in the "group_artist" table are
 * valid => "artist" must be an artist and "group" must be a
 * group.
 */
CREATE TRIGGER `check_group_artist_integrity`
BEFORE INSERT ON `group_artist`
FOR EACH ROW BEGIN
 DECLARE group_type BOOLEAN;
 DECLARE artist_type BOOLEAN;

 SELECT `is_group` INTO group_type
 FROM `performers`
 WHERE `performer_id` = new.`group`;

 SELECT `is_group` INTO artist_type
 FROM `performers`
 WHERE `performer_id` = new.`artist`;
  IF group_type = FALSE THEN
   SIGNAL SQLSTATE '45000'
     SET MESSAGE_TEXT =
       'The performer "group" is not valid group.';
 END IF;

 IF artist_type = TRUE THEN
   SIGNAL SQLSTATE '45000'
     SET MESSAGE_TEXT =
       'The performer "artist" is not valid
        individual artist.';
 END IF;
END ||

/**
 * Checks if an artist has participated in the previous
 * 3 years
 */
CREATE TRIGGER `check_consecutive_participations`
BEFORE INSERT ON `performances`
FOR EACH ROW BEGIN
 DECLARE y1 BOOLEAN DEFAULT FALSE;
 DECLARE y2 BOOLEAN DEFAULT FALSE;
 DECLARE y3 BOOLEAN DEFAULT FALSE;
 IF EXISTS (
   SELECT 1
   FROM `performances` AS p
   WHERE
     YEAR(p.`date_time`) = YEAR(NEW.`date_time`) - 1
     AND p.`performer` = new.`performer`
   LIMIT 1
 ) THEN
   SET y1 = TRUE;
 END IF;

 IF EXISTS (
   SELECT 1
   FROM `performances` AS p
   WHERE
     YEAR(p.`date_time`) = YEAR(NEW.`date_time`) - 2
     AND p.`performer` = new.`performer`
   LIMIT 1
 ) THEN
   SET y2 = TRUE;
 END IF;

 IF EXISTS (
   SELECT 1
   FROM `performances` AS p
   WHERE
     YEAR(p.`date_time`) = YEAR(NEW.`date_time`) - 3
     AND p.`performer` = new.`performer`
   LIMIT 1
 ) THEN
   SET y3 = TRUE;
 END IF;

 IF y1 AND y2 AND y3 THEN
   SIGNAL SQLSTATE '45000'
     SET MESSAGE_TEXT =
       'Performer has participated in the last 3 years.';
 END IF;
END ||
-- #endregion

DELIMITER ;

COMMIT;
```