052400-1 VU Information Management and Systems Engineering (2025W)

Milestone 1

Group 5

Student 1: Aziz Iftekher - 12338137

Student 2: Baur Lennard - 12018378

[*11-11-2025*], Vienna

Contents

1 Milestone 1 **3**
1.1 Team - Conceptual Modeling . . . 3
1.1.1 Describe the Application Domain . . . 3
1.1.2 Logical Design – ER Diagram in Chen Notation . . . 4
1.1.3 Relational Modeling – SQL CREATE Statements . . . 4
1.2 Individual - Student 1 . . . 6
1.2.1 Use Case Definition and Design . . . 6
1.2.2 Textual Description . . . 6
1.2.3 Graphical Representation . . . 7
1.3 Analytics Report . . . 7
1.3.1 Concept . . . 7
1.3.2 Proof of Concept . . . 11
1.4 NoSQL Design . . . 17
1.4.1 Design Overview . . . 17
1.4.2 Expected Execution and Possible Changes . . . 20
1.4.3 Five Rules of Thumb . . . 21
1.5 Individual - Student 2 . . . 23
1.5.1 Use Case Definition and Design . . . 23
1.5.2 Textual Description . . . 23
1.5.3 Graphical Representation . . . 24
1.6 Analytics Report . . . 24
1.6.1 Concept . . . 24
1.6.2 Proof of Concept . . . 25
1.7 NoSQL Design . . . 27
1.7.1 Design Overview . . . 27
1.7.2 Expected Execution and Possible Changes . . . 29
1.7.3 Five Rules of Thumb . . . 29

1. Milestone 1

1.1. Team - Conceptual Modeling

1.1.1. Describe the Application Domain

Our idea is to develop a comprehensive management system for technology conferences and hackathons that streamlines event organization, participant management, and project evaluation. Therefore, our domain encompasses all aspects of organizing tech events, from venue management and sponsor coordination to participant registration and hackathon submission evaluation.

The Tech Conference and Hackathon Management System facilitates the organization and execution of technology events. The system manages HackathonEvent [event_id, name, start_date, end_date, event_type, max_participants] that take place at specific Venues [venue_id, name, address, capacity, facilities]. Each Venue hosts multiple HackathonEvent, while each HackathonEvent is hosted at one Venue. HackathonEvent organize multiple Workshops [workshop_id, title, description, duration, skill_level, max_attendees], providing educational sessions for attendees. Each Workshop belongs to one HackathonEvent. Sponsors [sponsor_id, company_name, industry, website, contribution_amount] support multiple HackathonEvent through financial contributions, and each HackathonEvent can be supported by multiple Sponsors.

Our system distinguishes between different types of people through inheritance. Person [person_id, first_name, last_name, email, phone] serves as the base entity with two specializations: Participants [registration_date, t_shirt_size, dietary_restrictions] who attend and compete, and Judges [expertise_area, years_experience, organization] who evaluate submissions.

Participants can take on an Innovation Manager role, where experienced Participants manage multiple other Participants, while each managed Participant is managed by one Innovation Manager. Participants create multiple Submissions [submission_id, project_name, description, submission_time, technology_stack, repository_url] during hackathons, and each Submission can be created by multiple Participants forming a team. Judges evaluate multiple Submissions, and each Submission is evaluated by multiple Judges, providing scores and feedback.

The system tracks Registrations [registration_number, registration_timestamp, payment_status, ticket_type] as a weak entity that depends on both a Participant and an HackathonEvent, identified by a composite key (*participant_id, event_id*) that ensures its existence is tied to both entities. Each Participant registers for multiple HackathonEvents through Registrations, and each HackathonEvent receives multiple Registrations from different Participants.

Requirements	Realization (Name the involved elements)
Entities	
Person	person_id, first_name, last_name, email, phone
Participant	person_id, registration_date, t_shirt_size, dietary_restrictions
Judge	person_id, expertise_area, years_experience, organization
HackathonEvent	event_id, venue_id,name, start_date, end_date, event_type, max_participants
Venue	venue_id, name, address, capacity, facilities
Workshop	workshop_id,title,description,duration,skill_level,max_attendees
Sponsor	sponsor_id,company_name,contribution_amount,industry, website
Submission	submission_id,project_name,description,submission_time, technology_stack,repository_url
Registration (Weak)	registration_id,participant_id, event_id, registration_number, registration_timestamp, payment_status,ticket_type
Relationships	
Participant – Participant	manages (1:n)
Venue – HackathonEvent	hosts (1:n)
HackathonEvent – Workshop	organizes (1:n)
Sponsor – HackathonEvent	supports (m:n)
Participant – Submission	creates (m:n)
Judge – Submission	evaluates (m:n)
Participant – HackathonEvent	registers (m:n via Registration)
Person – {Participant, Judge}	IS-A inheritance

1.1.2. Logical Design – ER Diagram in Chen Notation

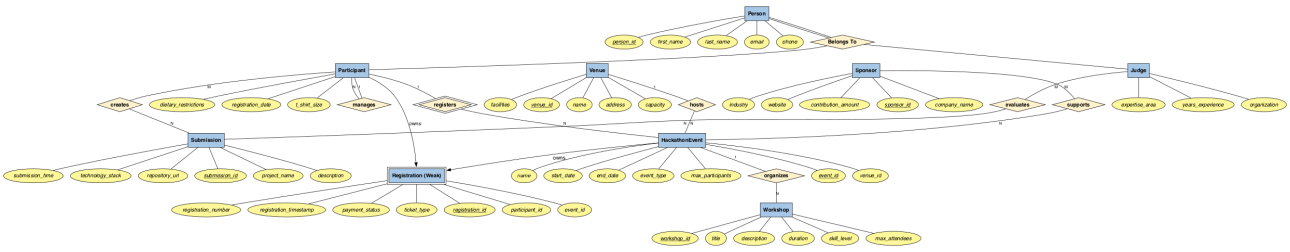

Figure 1: Chen Entity-Relationship (ER) Diagram

1.1.3. Relational Modeling – SQL CREATE Statements

The following relational schema was derived from the Chen ER diagram of the *Tech Conference and Hackathon Management System*. Each entity and relationship was translated into SQL table structures with explicit primary and foreign key constraints.M:N relationships were converted into associative tables, and the weak entity *Registration* uses a composite key (*person_id*, *event_id*) to maintain referential integrity. All relations are normalized to Third Normal Form (3NF).

Person(person_id, first_name, last_name, email, phone)

Participant(*person_id*, registration_date, t_shirt_size, dietary_restrictions, *manager_id*)

- FK: person_id → Person(person_id)
- FK: manager_id → Participant(person_id) (recursive)

Judge(*person_id*, expertise_area, years_experience, organization)

- FK: person_id → Person(person_id)

Venue(venue_id, name, address, capacity, facilities)

HackathonEvent(event_id, name, start_date, end_date, event_type, max_participants, *venue_id*)

- FK: venue_id → Venue(venue_id)

Workshop(workshop_id, title, description, duration, skill_level, max_attendees, *event_id*)

- FK: event_id → HackathonEvent(event_id)

Sponsor(sponsor_id, company_name, industry, website, contribution_amount)

Submission(submission_id, project_name, description, submission_time, technology_stack, repository_url)
Registration (Weak)(*person_id*, *event_id*, registration_number, registration_timestamp, payment_status, ticket_type)

- FK: person_id → Participant(person_id)
- FK: event_id → HackathonEvent(event_id)

Supports(*sponsor_id*, *event_id*)

- FK: sponsor_id → Sponsor(sponsor_id)
- FK: event_id → HackathonEvent(event_id)

Creates(*person_id*, *submission_id*)

- FK: person_id → Participant(person_id)
- FK: submission_id → Submission(submission_id)

Evaluates(*person_id*, *submission_id*, score, feedback)

- FK: person_id → Judge(person_id)
- FK: submission_id → Submission(submission_id)

Differences and Normalization Note:

- All M:N relationships from the Chen model (*supports*, *creates*, *evaluates*) were converted into associative tables.
- The recursive 1:N relationship in *Participant* was modeled using a self-referencing foreign key (*manager_id*).
- The weak entity *Registration* employs a composite primary key (*person_id*, *event_id*) to ensure dependency on both owner entities.
- The final relational schema is in 3NF, eliminating redundancy and maintaining referential integrity.

1.2. Individual - Student 1

Student 1: Aziz Iftekher - 12338137

1.2.1. Use Case Definition and Design

I am using **use case:** "Submit Hackathon Project" **Version Type:** Version 2 – IS-A Relationship (Participant inherits from Person; Submission lies outside the hierarchy)

1.2.2. Textual Description

Use Case Name: Submit Hackathon Project

Goal/Purpose: Allow a registered participant to submit a hackathon project for evaluation by judges.

Trigger: The participant selects the option "Submit Project" from their dashboard within the Hackathon Management System.

Precondition(s):

1. These conditions are assumed to be true before the use case starts.
2. The participant is logged in (a valid *Person* and *Participant* record exists).
3. The participant is registered for an active *HackathonEvent*.
4. The event's project submission period is open.
5. Once these conditions hold, the use case begins at "Click *Submit Project.*".

Main Flow: Submit a hackathon project?

1. Participant navigates to the registered hackathon event dashboard and selects *"Submit Project."*.
2. Participant fills in the project details, project name, description, technology stack, and repository URL.
3. System validates input and creates a new record in the *Submission* table.
4. System creates a relationship record in the *Creates* table linking the *participant* (IS-A *Person*) to the new *Submission*.
5. System confirms successful submission and displays a success message to the participant.

Postcondition(s):

1. A new project record is inserted into the *Submission* table with participant-provided details.
2. A corresponding link record between *Participant* and *Submission* is created in the *Creates* table.
3. As a result, the participant's submission becomes available for viewing or updating until the event deadline.

Entities Involved: ***Person*** serves as the superclass in the **IS-A hierarchy**, from which ***Participant*** is derived as a **specialized entit**y performing the project submission. The use case also involves ***Submission***, an **independent entity** that stores project details, and the **associative relationship *Creates***, which links each *participant* to their corresponding *Submission*.

1.2.3. Graphical Representation

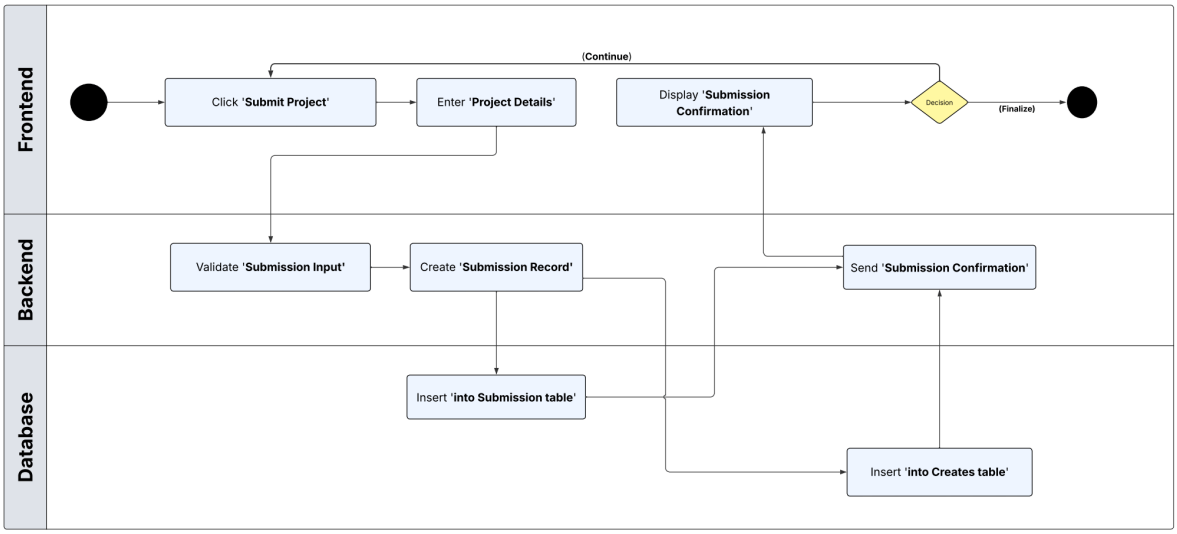

Figure 2: Activity Diagram: Submit Project for Evaluation

1.3. Analytics Report

1.3.1. Concept

Analytics Report Description

This analytics report focuses on tracking and analyzing **hackathon project submissions** created by participants during different events. It aims to help event organizers and innovation managers understand submission patterns, participant engagement, and technology usage across all projects submitted within a specific time frame.

Purpose:

- Identify which participants have submitted projects.
- Analyze technology stacks used across submissions.
- Measure engagement (time from registration to submission).
- Track submission volume trends over time.
- Assess participant productivity (number of projects per participant).

Filter Field: The report uses `submission_time` as the filter field, allowing analysis within specific date ranges such as during an event, the last week etc.

Business Value:

- Track real-time project submissions during hackathon events.
- Identify active and highly engaged participants.
- Analyze trends in technologies used across all submissions.

- Monitor participant engagement timelines.
- Support resource allocation and event planning based on participation data.

Entities Involved

The analytics report involves the following **three entities**:

- **Person**: Stores basic participant information (`person_id`, `first_name`, `last_name`, `email`, `phone`)
- **Participant**: Links persons to their participant-specific data (`person_id`, `registration_date`, `t_shirt_size`, `dietary_restrictions`)
- **Submission**: Contains project details created by the "Submit Hackathon Project" use case (`submission_id`, `project_name`, `description`, `submission_time`, `technology_stack`, `repository_url`)

Relationship Table Used (not counted as entity):

- **Creates** (m:n): Links Participants to their Submissions

Entity	Attribute Included
Person	person_id, first_name, last_name, email
Participant	registration_date, t_shirt_size, dietary_restrictions
Submission	submission_id, project_name, description, submission_time, technology_stack, repository_url

Table 1: Data from each entity included in the analytics

Calculated Fields:(Can be computed by SQL query)

- `days_since_registration`: shows the time difference between registration and project submission.
- `total_submissions_by_participant`: shows the total number of projects submitted by each participant.

How the Use Case Affects

Before Use Case Execution:

- When a participant has NOT yet submitted a project, they will not appear in the report
- The Submissions table will have fewer records
- The filtered query (e.g., submissions after '2025-11-08') will return fewer results

After Use Case Execution ("Submit Hackathon Project"):

- A new record is inserted into the **Submission** table
- A new record is inserted into the **Creates** table (linking Participant to Submission)
- The query will NOW include this new submission in the results
- The `total_submissions_by_participant` count will increase for this participant
- **Result:** The query output changes, showing the newly submitted project

It demonstrates that the analytics report is **directly impacted** by the use case execution, fulfilling the requirement that "results should change after executing the use case."

SQL Query Statement

```
-- Analytics Report: Hackathon Submission Statistics
-- Filter: Submissions created after a specific date
-- Focuses on data created by "Submit Hackathon Project" use case

SELECT
    s.submission_id,
    s.project_name,
    s.description,
    s.submission_time,
    s.technology_stack,
    s.repository_url,

    -- Participant Information (from Person and Participant entities)
    p.person_id,
    p.first_name AS participant_first_name,
    p.last_name AS participant_last_name,
    p.email AS participant_email,
    pt.registration_date,
    pt.t_shirt_size,
    pt.dietary_restrictions,

    -- Time-based statistics
    DATEDIFF(CURRENT_DATE, pt.registration_date) AS days_since_registration,

    -- Submission count per participant
    (SELECT COUNT(*)
     FROM Creates c2
     WHERE c2.person_id = pt.person_id) AS total_submissions_by_participant

FROM Submission s

-- Join to get the participant who created this submission
INNER JOIN Creates c ON s.submission_id = c.submission_id
INNER JOIN Participant pt ON c.person_id = pt.person_id
INNER JOIN Person p ON pt.person_id = p.person_id

-- FILTER FIELD: Only include submissions after November 8, 2025
WHERE s.submission_time >= '2025-11-08 00:00:00'

ORDER BY
    s.submission_time DESC,
    p.last_name ASC;
```

Query Explanation

SELECT Clause:

- Retrieves all relevant submission details from the **Submission** entity
- Includes participant information from **Person** and **Participant** entities
- Calculates `days_since_registration` to show engagement timeline

- Uses a subquery to count `total_submissions_by_participant`

FROM and JOIN Clauses:

- Starts from **Submission** (the core entity affected by the use case)
- **INNER JOIN** with **Creates** to link submissions to participants (m:n relationship)
- **INNER JOIN** with **Participant** and **Person** to get participant details

WHERE Clause (FILTER FIELD):

- Filters submissions based on `submission_time >= '2025-11-08 00:00:00'`
- This allows organizers to analyze recent submissions or submissions within a specific event period
- The date can be adjusted based on reporting needs

GROUP BY Clause:

- Not required in this query as aggregations are handled via subquery

ORDER BY Clause:

- Orders by submission time (most recent first)
- Secondary sort by participant last name (alphabetical)

Expected Output Columns

Column Name	Description	Source Entity
submission_id	Unique submission identifier	Submission
project_name	Name of the submitted project	Submission
description	Project description	Submission
submission_time	Timestamp of submission	Submission
technology_stack	Technologies used in project	Submission
repository_url	GitHub/GitLab repository link	Submission
person_id	Participant's unique ID	Participant/Person
participant_first_name	Participant's first name	Person
participant_last_name	Participant's last name	Person
participant_email	Participant's email	Person
registration_date	When participant registered	Participant
t_shirt_size	Participant's t-shirt size	Participant
dietary_restrictions	Participant's dietary needs	Participant
days_since_registration	Days from registration to now	Calculated
total_submissions_by_participant	Count of all submissions	Calculated

Table 2: Expected output columns from the analytics query

Impact of Use Case Execution

Scenario: Participant "Iftekher Aziz" submits a project titled "Landing Page Design" **BEFORE Use Case Execution:(Example)**

```
Query returns: 5 submissions (from other participants)
Aziz's submission: NOT present in results
```

AFTER Use Case Execution:

```
Query returns: 6 submissions
New row appears:
- submission_id: 6
- project_name: Landing Page Design
- participant_first_name: Iftekher
- participant_last_name: Aziz
- submission_time: 2025-11-10 14:30:00
- technology_stack: HTML, CSS, Tailwind, JS, ReactJS
- days_since_registration: 26
- total_submissions_by_participant: 1
```

1.3.2. Proof of Concept

Description:

This proof of concept demonstrates that the analytics query results change before and after executing the "Submit Hackathon Project" use case. All testing was performed in MariaDB using the exact table schema from section 1.1.3.
Three separate SQL files were created to ensure reproducibility:

- **Student1_InsertData_Initial.sql**: Contains CREATE TABLE statements and initial dummy data (5 participants, 4 submissions before Aziz submits)
- **Student1_Querystatement.sql**: Contains the analytics query that filters submissions after November 8, 2025
- **Student1_InsertData_UseCase.sql**: Contains INSERT statements that mimic the "Submit Hackathon Project" use case execution (Aziz submits his project)

Testing Methodology

The proof of concept follows this execution sequence:

1. Execute `Student1_InsertData_Initial.sql` to create tables and populate initial dummy data
2. Execute `Student1_Querystatement.sql` to view results BEFORE use case execution
3. Execute `Student1_InsertData_UseCase.sql` to simulate Aziz submitting his project
4. Re-execute `Student1_Querystatement.sql` to view results AFTER use case execution.

Student1_InsertData_Initial.sql

This step shows the execution of the SQL script **Student1_InsertData_Initial.sql** in MariaDB using Visual Studio Code.

In this file, I created the core tables required for my use case (Person, Participant, Submission, and Creates) exactly as defined in section 1.1.3 (Relational Design). The script first drops existing tables in the correct order to avoid foreign key conflicts, then recreates each table with the appropriate primary and foreign key relationships.

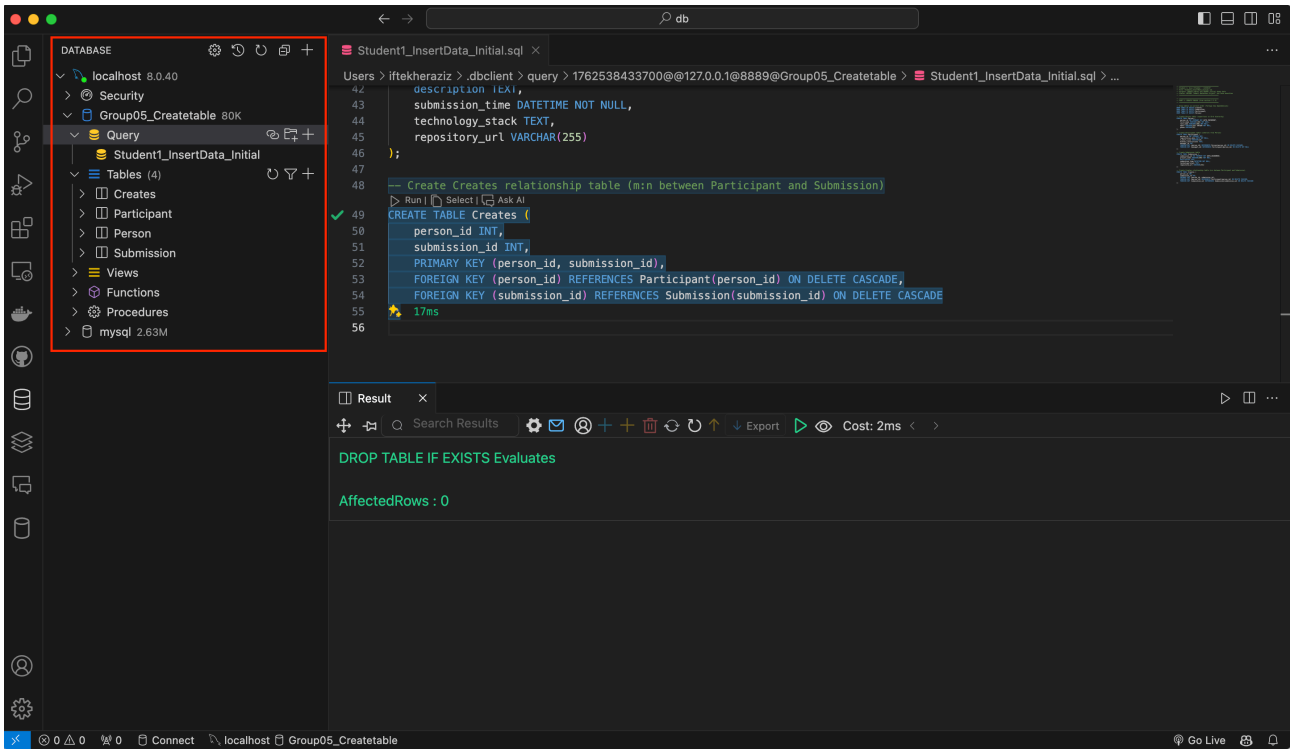

Figure 3: Execution of `Student1_InsertData_Initial.sql` in MariaDB showing table creation for the *Submit Hackathon Project* use case.

This setup forms the foundation for inserting dummy data and running the analytics report later. The screenshot confirms that the tables were successfully created before executing the "**Submit Hackathon Project**" use case.

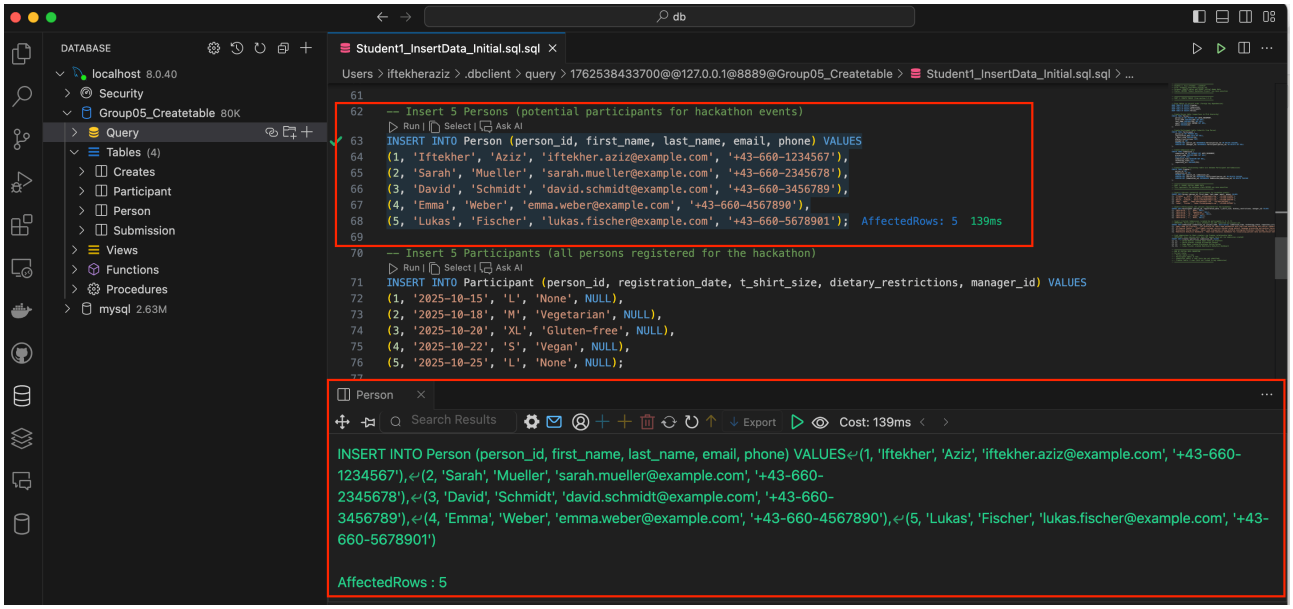

Figure 4: Insertion of initial dummy data into the *Person* table before executing the use case.

This step shows the execution of the SQL script that inserts initial dummy data into the **Person** table. Five sample participants were added to represent users registered in the hackathon management system.

Each record includes a unique `person_id`, along with first name, last name, email, and phone details. These entries serve as the foundational data for participants before executing the “Submit Hackathon Project” use case. The output confirms successful insertion of five records, establishing the initial state of the database for the analytics report.

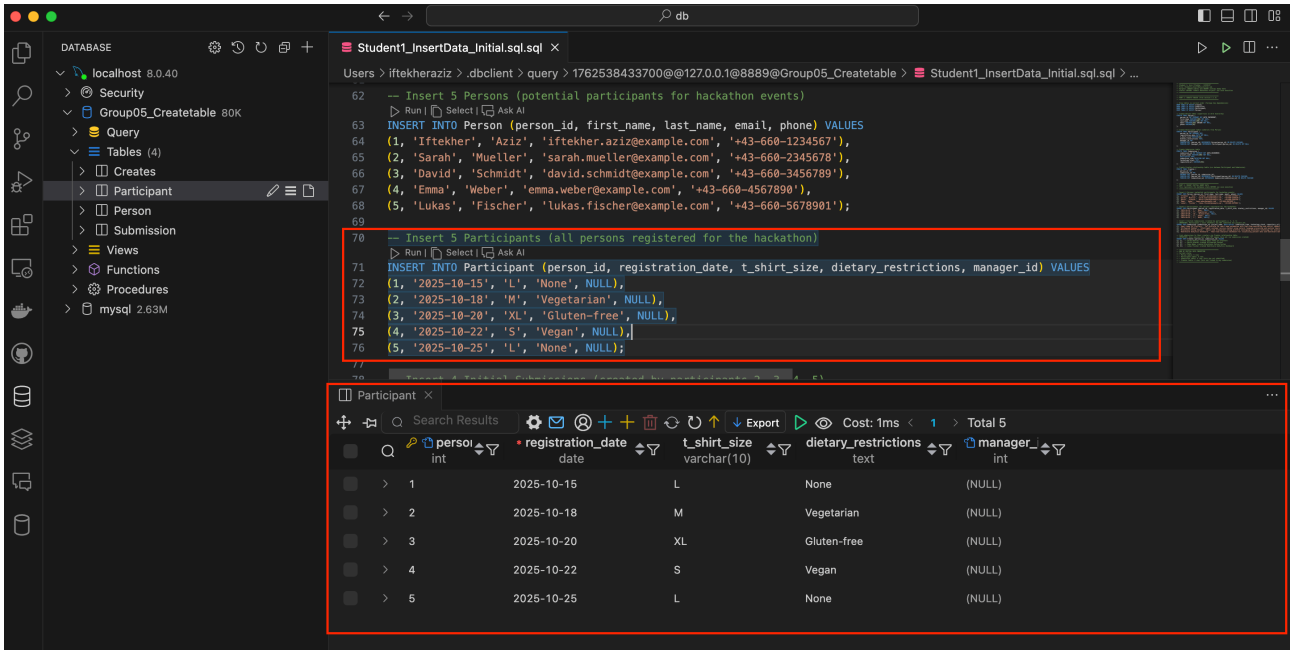

Figure 5: Insertion of initial dummy data into the *Participant* table showing all registered hackathon participants.

This step inserts five records into the **Participant** table, linking each participant to their corresponding *Person* entry. Each record includes registration details such as `registration_date`, `t_shirt_size`, and `dietary_restrictions`, reflecting individual participant preferences and event readiness. These records represent participants who are officially registered for the hackathon before executing the “Submit Hackathon Project” use case. The output confirms successful insertion of five participants, maintaining relational consistency with the *Person* table.

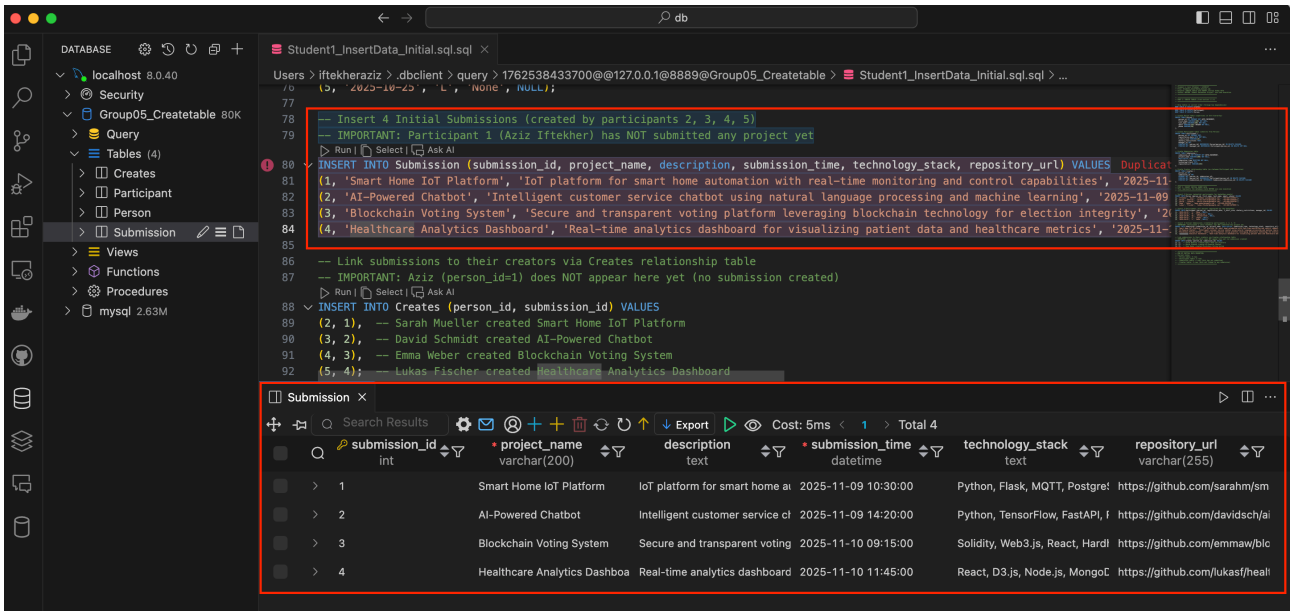

Figure 6: Insertion of initial dummy data into the *Submission* table representing projects created by participants 2–5 before use case execution.

This step inserts four initial records into the **Submission** table, representing hackathon projects created by participants 2, 3, 4, and 5. Each record includes the project's name, detailed description, submission timestamp, technology stack, and repository link. It is important to note that **Participant 1 (Aziz Iftekher)** has not yet submitted any project, ensuring a clear state difference before executing the "Submit Hackathon Project" use case. These submissions simulate existing project data within the system, forming the analytical foundation for monitoring and evaluating participant engagement.

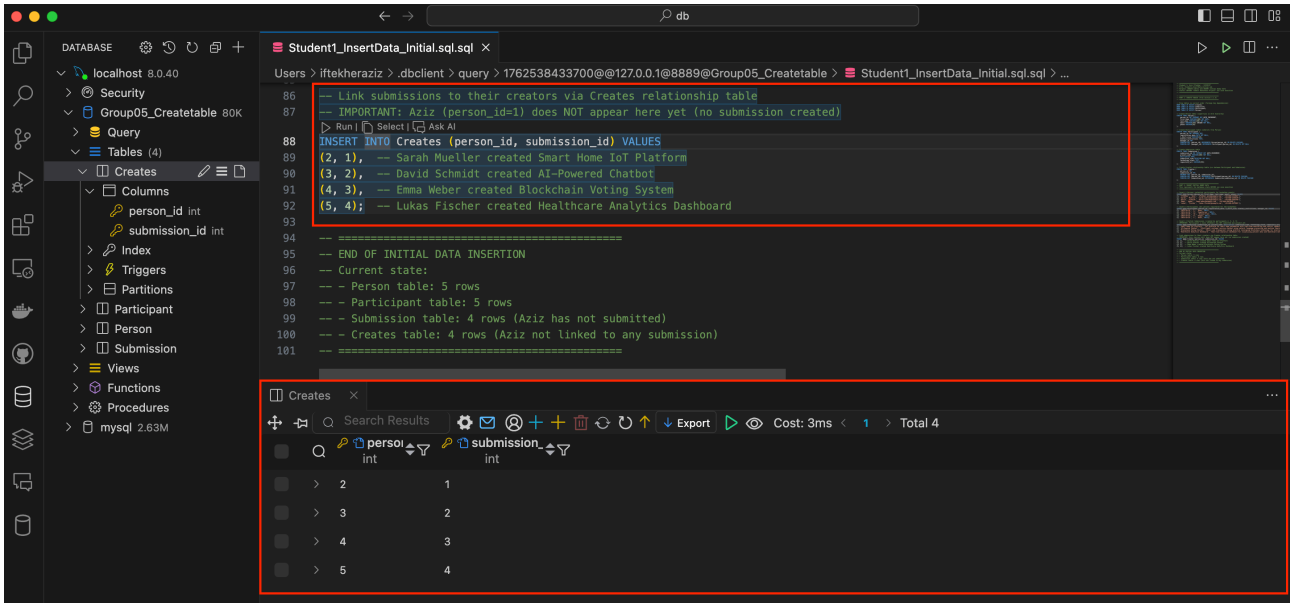

Figure 7: Linking participants to their submitted projects through the *Creates* relationship table before executing the use case.

This step establishes the relationship between participants and their submitted projects through the **Creates** table. Each record links a participant (`person_id`) with a corresponding project (`submission_id`)

created in the hackathon. Participants 2, 3, 4, and 5 are successfully linked to their respective projects, while **Participant 1 (Aziz Iftekher)** does not appear here, as no project has been submitted yet. This table serves as a bridge in the many-to-many (*m:n*) relationship between *Participant* and *Submission*, ensuring relational integrity before the execution of the "Submit Hackathon Project" use case.

Student1_Querystatement.sql

This step shows the execution of the SQL script **Student1_Querystatement.sql** in MariaDB using Visual Studio Code.

This step executes the analytics report query that retrieves statistics on hackathon project submissions along with participant information. The query joins data from the **Submission**, **Creates**, **Participant**, and **Person** tables to display project details, participant profiles, and calculated engagement metrics such as `days_since_registration` and `total_submissions_by_participant`.

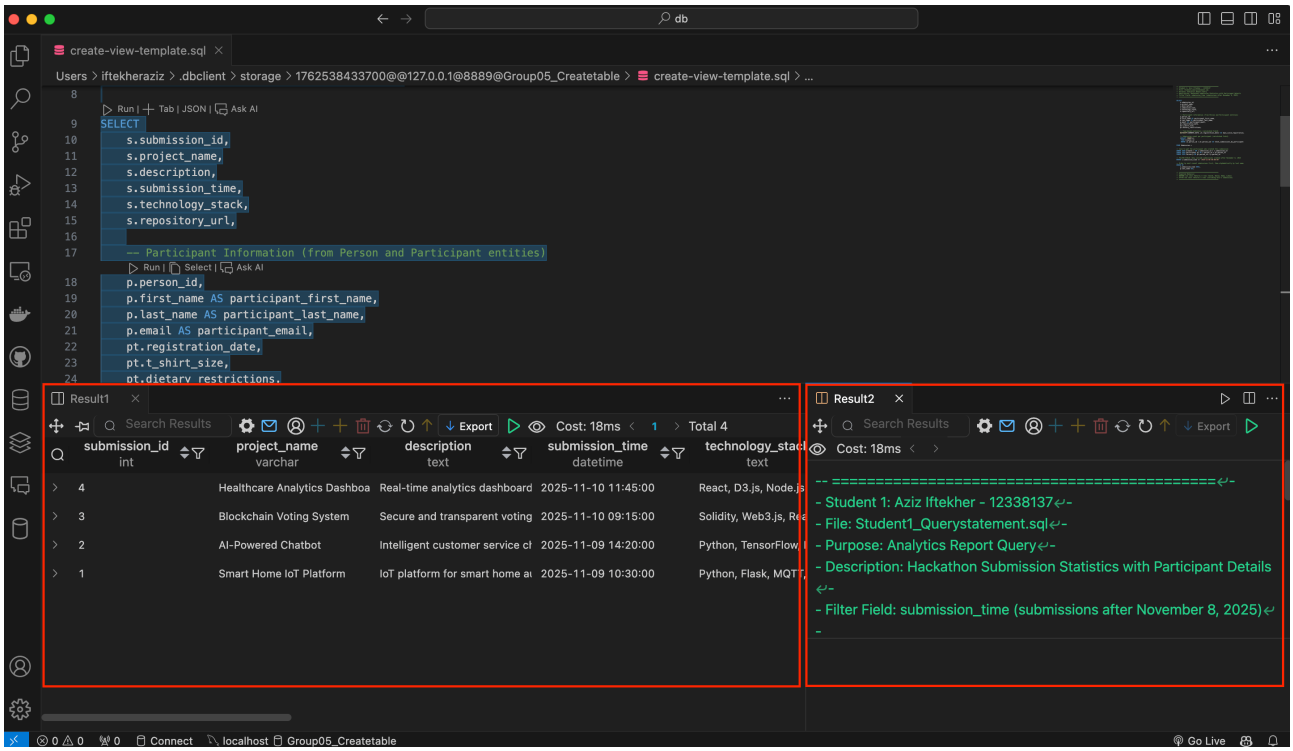

Figure 8: Execution of the analytics query showing initial results before the "Submit Hackathon Project" use case execution.

The filter field `submission_time` ensures that only projects submitted after `2025-11-08` are included. Before the "Submit Hackathon Project" use case is executed, the query returns four rows corresponding to participants 2–5 (Sarah, David, Emma, Lukas). After the use case is executed, a fifth record will appear for **Participant 1 (Aziz Iftekher)**, verifying that the report dynamically updates with new submission data.

Student1_InsertData_UseCase..sql

This step shows the execution of the SQL script **Student1_InsertData_UseCase..sql** in MariaDB using Visual Studio Code.

This step simulates the execution of the “Submit Hackathon Project” use case. In this scenario, **Participant 1 (Aziz Iftekher)** submits a new project titled *“Landing Page Design”*.

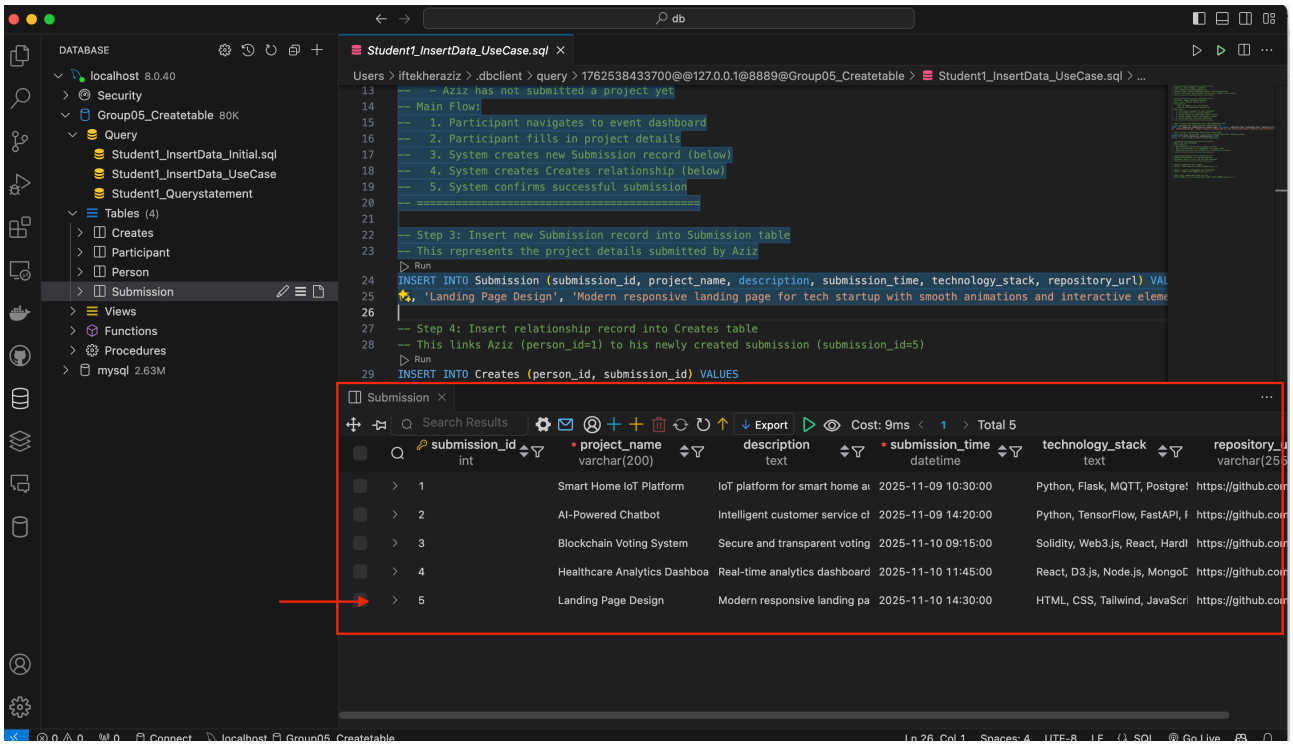

Figure 9: Insertion of a new record into the *Submission* table representing the project submitted by Participant 1 (Aziz Iftekher).

A new record is inserted into the **Submission** table, containing the project name, description, submission timestamp, technology stack, and repository URL. This insertion represents the system’s action of persisting a new project submission to the database once a participant completes the submission form. The addition of this record marks the change in the system state—transitioning Aziz from having no submission to being an active contributor in the analytics report.

This step finalizes the execution of the “Submit Hackathon Project” use case by inserting a relationship record into the **Creates** table. It establishes a link between **Participant 1 (Aziz Iftekher)** and his newly created project (`submission_id = 5`), titled *“Landing Page Design”*.

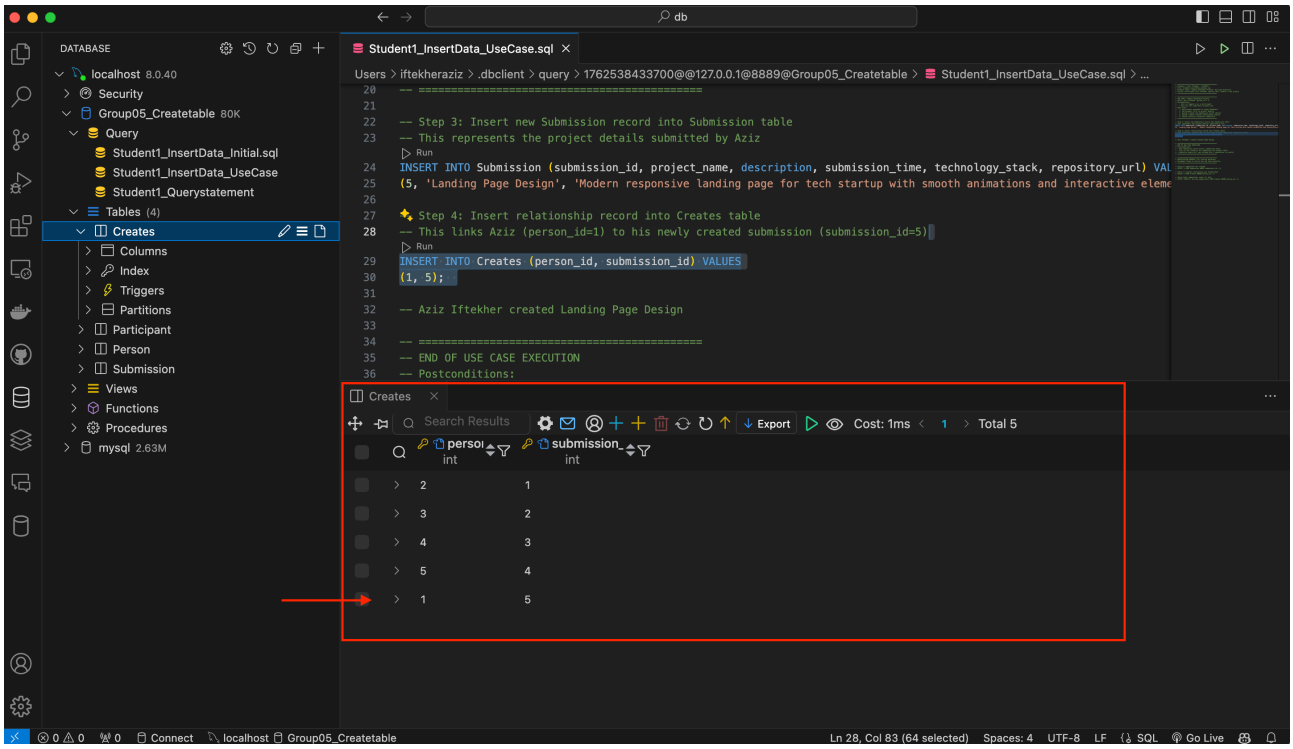

Figure 10: Linking Participant 1 (Aziz Iftekher) to the new project in the *Creates* table, completing the “Submit Hackathon Project” use case.

This insertion confirms that the participant is now formally connected to their submission in the database. After this step, the **Creates** table expands from 4 to 5 records, completing the use case execution and ensuring that the analytics report query now includes Aziz’s submission. The postconditions are fulfilled — a new submission record exists, a participant–submission link has been established, and the analytics report output dynamically updates to reflect the change.

1.4. NoSQL Design

1.4.1. Design Overview

Based on the **“Submit Hackathon Project”** use case from Task 1.2, I propose a document-oriented MongoDB structure where submission documents embed participant information for optimal query performance.

Entities Involved

The following entities from the relational model are represented in the NoSQL design:

1. **Person** – Identification data (person_id, first_name, last_name, email, phone)
2. **Participant** – Registration details (registration_date, t_shirt_size, dietary_restrictions)
3. **Submission** – Project information (submission_id, project_name, description, submission_time, technology_stack, repository_url)
4. **Creates** – Many-to-many relationship linking participants to submissions

Design Strategy

I chose an **embedded document approach** where participant data is denormalized within each submission document. Submissions serve as root documents with participant information embedded inside.

Rationale:

- Submissions are always queried with participant details (for dashboards, leaderboards, evaluations)
- Eliminates joins and multiple lookups
- Participant data is relatively static after registration
- Supports team submissions with an embedded array (typical team size: 1-5 members)

Entity Mapping

Relational Entity	NoSQL Representation
Person	Embedded in `created_by` object
Participant	Embedded in `created_by` object
Submission	Root document
Creates (m:n)	`team_members` array

Table 3: Mapping of relational entities to NoSQL structure

JSON Example Document

Aziz Iftekher's "Landing Page Design" submission:

```
{
  "_id": ObjectId("673a1b2c3d4e5f6a7b8c9d0e"),
  "submission_id": 5,
  "project_name": "Landing Page Design",
  "description": "Modern responsive landing page for tech startup
                  with smooth animations and interactive elements",
  "submission_time": ISODate("2025-11-10T14:30:00Z"),
  "technology_stack": [
    "HTML",
    "CSS",
    "Tailwind",
    "JavaScript",
    "ReactJS"
  ],
  "repository_url": "https://github.com/iftekheraziz/landing-page-design",

  "created_by": {
    "person_id": 1,
    "first_name": "Iftekher",
    "last_name": "Aziz",
    "email": "iftekher.aziz@example.com",
    "phone": "+43-660-1234567",
    "registration_date": ISODate("2025-10-15T00:00:00Z"),
    "t_shirt_size": "L",
    "dietary_restrictions": "None"
  },

```

```
  "team_members": [
    {
      "person_id": 1,
      "first_name": "Iftekher",
      "last_name": "Aziz",
      "role": "Full-stack Developer"
    }
  ],

  "metadata": {
    "created_at": ISODate("2025-11-10T14:30:00Z"),
    "updated_at": ISODate("2025-11-10T14:30:00Z"),
    "status": "submitted",
    "version": 1
  }
}
```

Multi-Author Example

Team submission with multiple participants:

```
{
  "_id": ObjectId("673a1b2c3d4e5f6a7b8c9d0f"),
  "submission_id": 6,
  "project_name": "Blockchain Voting System",
  "description": "Secure voting platform using blockchain technology",
  "submission_time": ISODate("2025-11-10T16:45:00Z"),
  "technology_stack": ["Solidity", "Web3.js", "React", "Hardhat"],
  "repository_url": "https://github.com/team-blockchain/voting-system",

  "created_by": {
    "person_id": 4,
    "first_name": "Emma",
    "last_name": "Weber",
    "email": "emma.weber@example.com",
    "phone": "+43-660-4567890",
    "registration_date": ISODate("2025-10-22T00:00:00Z"),
    "t_shirt_size": "S",
    "dietary_restrictions": "Vegan"
  },

  "team_members": [
    {
      "person_id": 4,
      "first_name": "Emma",
      "last_name": "Weber",
      "role": "Smart Contract Developer"
    },
    {
      "person_id": 3,
      "first_name": "David",
      "last_name": "Schmidt",
      "role": "Frontend Developer"
    }
  ],

  "metadata": {
    "created_at": ISODate("2025-11-10T16:45:00Z"),
```

```
38      "updated_at": ISODate("2025-11-10T16:45:00Z"),
39      "status": "submitted",
40      "version": 1
41    }
42  }
```

1.4.2. Expected Execution and Possible Changes

Expected Execution of the Use Case

When a participant executes the **"Submit Hackathon Project"** use case, the following happens:

Typical execution pattern:

- Each participant submits 1-3 projects during a hackathon event
- Each submission gets viewed 50-200 times (by judges, organizers, other participants)
- Submissions are rarely modified after creation
- This creates a read-to-write ratio of about 100:1

Database operations:

- **SQL approach:** Requires 2-6 separate INSERT statements (one for Submission table, 1-5 for Creates table depending on team size)
- **MongoDB approach:** Requires only 1 insertOne() operation that includes all submission and participant data

Why This Design Works Well

My embedded document design fits this use case because:

1. **Writes are simple:** Creating a submission happens in one database operation instead of multiple related inserts
2. **Reads are fast:** When displaying a submission, I get all the data (project details plus participant info) in one query without joining tables
3. **Common queries are efficient:** Showing a list of submissions with participant names requires just one find() operation
4. **Data stays together:** Since submissions are almost always viewed with participant details, keeping them in the same document makes sense

How Changes Would Affect the Design

If reads increase significantly (e.g., 10x more views per submission):
This would actually make my embedded design even better because:

- More reads = single-query retrieval becomes more valuable
- No joins means consistent fast response times even under load

I would only need to add:

- Database indexes on submission_time and technology_stack for faster filtering
- Maybe a caching layer if reads become extremely frequent

The document structure itself wouldn't need to change.

If writes increase significantly (e.g., participants submit 20+ projects each):

The embedded design still works fine for writes. However, a problem appears if participant information starts changing frequently:

- If a participant updates their email, I'd need to update that email in every submission document they created
- With 20 submissions per participant, that's 20 document updates instead of 1

In this case, I would consider a hybrid approach:

- Keep embedding static data (first name, last name, registration date)
- Use a reference for frequently changing data (email, phone)
- This way, most queries still work with one operation, but updates affect fewer documents

If team sizes grow large (20-50 members per submission):

My current design embeds all team members in an array. This becomes problematic with large teams because:

- The document size grows significantly
- Querying and updating specific team members becomes slower
- Array operations on 50 elements are less efficient than small arrays

I would need to switch to a referencing approach:

- Store just team member IDs in the submission document
- Keep full team member details in a separate collection
- Accept that queries now require two database operations

Change	Effect on Design	Action Needed
More frequent reads	Design works better	Add indexes
More frequent writes + participant updates	Duplication becomes costly	Switch to hybrid (embed + reference)
Large team sizes	Array too big	Switch to referencing

Table 4: How operation changes affect my NoSQL design

1.4.3. Five Rules of Thumb

Rule 1: Favor embedding unless there is a compelling reason not to.

I used embedding to keep participant data inside each submission document. Since submissions are always viewed together with participant details, embedding reduces the number of queries and improves read performance.

Rule 2: The need to access an object on its own is a compelling reason not to embed it.

Participant data is not frequently accessed separately in this use case. Therefore, embedding it within submissions makes sense. If organizers later need to view or update participant profiles independently, I would separate this data into its own collection.

Rule 3: High-cardinality arrays are a compelling reason not to embed.

Currently, team sizes are small (1–5 members), so embedding the `team_members` array is efficient. If team sizes grow larger (e.g., 20+ members), I would switch to referencing to avoid large array updates and document growth.

Rule 4: Consider the write/read ratio of a collection/document when denormalizing.

The system performs far more reads than writes (about 100:1). Embedding works best here because the data is read often but written only once when a project is submitted.

Rule 5: Structure your data to match the ways that your application queries and updates it.

Most queries fetch full submission details with participant and team data. My structure aligns with these queries, allowing a single `find()` operation to return everything needed for dashboards or analytics.

1.5. Individual - Student 2

Student 2: Baur Lennard - 12018378

1.5.1. Use Case Definition and Design

I am doing **Version 1: Weak Entity** (Registration)

1.5.2. Textual Description

Use Case Name (weak entity): Register Participant for Event
Trigger: A participant clicks the "Register for Event" button on the event details page.
Preconditions:

- The participant must be logged into the system (participant exists in the database).
- The event must exist and be open for registration.
- The event has not reached maximum capacity (current registrations < max_participants).
- The participant has not already registered for this event.

Main Flow:

1. The participant selects an event from the available events list.
2. The system displays event details including available spots.
3. The participant fills in the registration form (ticket type, payment information).
4. The system validates the registration data and checks the capacity constraints.
5. The system creates a new Registration record with a unique registration_number.
6. The system updates the event's current participant count.
7. The system sends a confirmation email to the participant.
8. The system displays a success message with registration details.

Postconditions:

- A new Registration entry is created in the database.
- The registration is linked to both the Participant and Event (weak entity dependency).
- The participant receives a unique registration_number for reference.
- The event's registered participant count is incremented.
- Payment status is set to "pending" or "completed" based on payment method.

Entities: Participant (from Person IS-A), Event, Registration (weak entity)

1.5.3. Graphical Representation

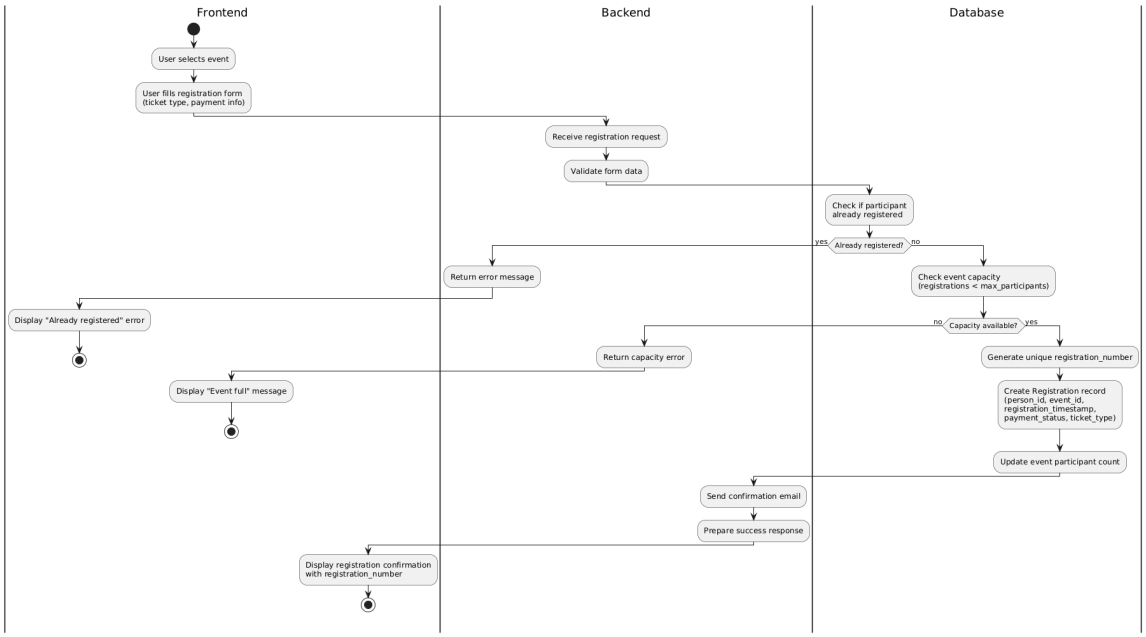

Figure 11: Activity Diagram: Register Participant for Event

1.6. Analytics Report

1.6.1. Concept

The analytics report provides event organizers with comprehensive statistics about participant registrations across hackathon events. The report analyzes registration patterns, payment completion rates, and venue capacity utilization to support event planning and management decisions.

Entities Involved

The analytics report queries data from five entities:

1. **Person** – Stores basic participant information (`person_id`, `first_name`, `last_name`, `email`, `phone`)
2. **Participant** – Extended participant details including registration dates and manager relationships (`person_id`, `registration_date`, `t_shirt_size`, `dietary_restrictions`, `manager_id`)
3. **HackathonEvent** – Event information including dates, type, and capacity constraints (`event_id`, `name`, `start_date`, `end_date`, `event_type`, `max_participants`, `venue_id`)
4. **Venue** – Location details for hosting events (`venue_id`, `name`, `address`, `capacity`, `facilities`)
5. **Registration (Weak Entity)** – The registration records created by the use case (`person_id`, `event_id`, `registration_number`, `registration_timestamp`, `payment_status`, `ticket_type`)

Note: Registration is the weak entity that depends on both **Participant** and **HackathonEvent**.

Filter Field

The report uses `event_type` as the primary filter field, allowing organizers to analyze registrations for specific event categories (e.g., *Hackathon*, *Conference*, *Workshop*). An additional time-based filter on `registration_timest` ensures only recent registrations are included.

Report Output

For each event matching the filter criteria, the report displays:

- Event identification and scheduling details (ID, name, dates, type)
- Capacity metrics (maximum participants, total registrations, utilization percentage)
- Venue information (name, address, capacity)
- Payment statistics (completed vs. pending payments)
- List of registered participants with their ticket types
- Capacity utilization percentage showing how full each event is

Impact of Use Case Execution

When the *"Register Participant for Event"* use case is executed, new **Registration** records are inserted into the database. These new registrations directly impact the analytics report by:

- Increasing the `total_registrations` count for the affected events
- Adding new participant names to the `registered_participants` list
- Updating payment status statistics (pending vs. completed)
- Increasing the `capacity_percentage` calculation
- Modifying the ordering of events (if sorted by registration count)

1.6.2. Proof of Concept

To verify the analytics report works correctly, I tested it in MariaDB using dummy data. The proof of concept follows these steps:

Step 1: Database Setup

First, I created all necessary tables using the SQL CREATE statements from section 1.1.3. The analytics report requires five tables: Person, Participant, Venue, HackathonEvent, and Registration.

Step 2: Initial Data Insertion

I inserted sample data into each table. This initial dataset represents the state of the database before any participant registers for an event. The data includes:

- 3 Venues (Tech Hub Vienna, Innovation Center, Startup Campus)
- 5 Persons who are Participants (Anna, Michael, Sarah, David, Lisa)
- 4 HackathonEvents (including 2 events of type "Hackathon")
- 4 initial Registrations connecting participants to events

All foreign key relationships were maintained to ensure referential integrity between tables.

Step 3: Initial Report Execution

I executed the analytics query to establish a baseline. The query filters events by type "Hackathon" and displays registration statistics for each event.

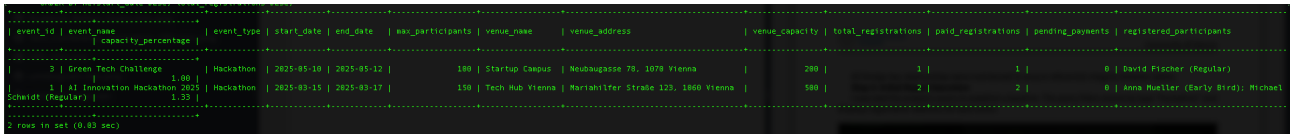

Figure 12: Analytics Report Results - Before Use Case Execution

As shown in Figure 12, the initial results show:

- **Green Tech Challenge**: 1 registration (David Fischer)
- **AI Innovation Hackathon 2025**: 2 registrations (Anna Mueller, Michael Schmidt)

Step 4: Use Case Execution

Next, I executed the use case "Register Participant for Event" by inserting three new Registration records:

- Lisa Wagner registered for AI Innovation Hackathon 2025
- Anna Mueller registered for Green Tech Challenge
- Michael Schmidt registered for Web Development Conference

These INSERT statements simulate participants registering for events through the system.

Step 5: Final Report Execution

After executing the use case, I ran the same analytics query again to observe the changes.

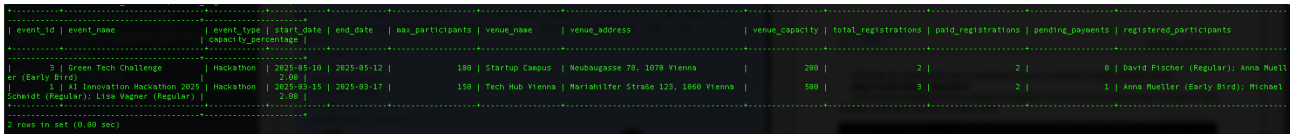

Figure 13: Analytics Report Results - After Use Case Execution

As shown in Figure 13, the results have changed:

- **Green Tech Challenge**: 2 registrations (increased by 1)
- **AI Innovation Hackathon 2025**: 3 registrations (increased by 1)

Verification

The difference between the two query results proves that:

1. The use case successfully creates new Registration records (weak entity)
2. The analytics report correctly reflects these database changes
3. All five entities (Person, Participant, HackathonEvent, Venue, Registration) contribute data to the report
4. The filter field (event_type = 'Hackathon') works as intended

The complete SQL statements are provided in the .zip file

1.7. NoSQL Design

1.7.1. Design Overview

For the use case, I propose an event-centric document based design in MongoDB. This design optimizes for the most common access pattern: retrieving event details along with all registrations and venue information in a single query.

Entities

All five entities from the use case are represented in the NoSQL design:

1. **HackathonEvent** - Main document at the root level
2. **Venue** - Embedded within the event document
3. **Registration** - Embedded as an array within the event document
4. **Participant** - Stored in a separate collection, referenced by participant_id
5. **Person** - Combined with Participant data in the participants collection

Design Rationale

The design uses two main collections: `events` and `participants`.

Venue embedding: The venue information is embedded directly in the event document because each event has exactly one venue, and venue data is relatively stable and small. This eliminates the need for joins when displaying event details.

Registration embedding: Registrations are stored as an array within the event document because they are existence-dependent on events (weak entity relationship). This structure efficiently supports the primary query pattern "show all participants registered for this event" and respects the bounded array size constraint (max_participants).

Participant referencing: Participants are stored in a separate collection and referenced by participant_id rather than embedded. This approach reduces data duplication, as participant information can be reused across multiple events, and maintains a single source of truth for participant data.

JSON Document Examples

The following JSON examples demonstrate the NoSQL data structure:

Events Collection:

```
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "event_id": 1,
  "name": "AI Innovation Hackathon 2025",
  "start_date": ISODate("2025-03-15T00:00:00Z"),
  "end_date": ISODate("2025-03-17T00:00:00Z"),
  "event_type": "Hackathon",
  "max_participants": 150,
  "venue": {
    "venue_id": 1,
    "name": "Tech Hub Vienna",
    "address": "Mariahilfer Straße 123, 1060 Vienna",
    "capacity": 500,
    "facilities": ["WiFi", "Projectors", "Catering"]
```

```
  },
  "registrations": [
    {
      "registration_number": "REG-2025-001",
      "registration_timestamp": ISODate("2025-01-20T10:30:00Z"),
      "payment_status": "completed",
      "ticket_type": "Early Bird",
      "participant_id": 1,
      "participant_name": "Anna Mueller",
      "participant_email": "anna.mueller@email.com"
    },
    {
      "registration_number": "REG-2025-005",
      "registration_timestamp": ISODate("2025-02-25T11:20:00Z"),
      "payment_status": "pending",
      "ticket_type": "Regular",
      "participant_id": 5,
      "participant_name": "Lisa Wagner",
      "participant_email": "lisa.wagner@email.com"
    }
  ],
  "registration_count": 2
}
```

Participants Collection:

```
{
  "_id": ObjectId("507f1f77bcf86cd799439012"),
  "participant_id": 5,
  "first_name": "Lisa",
  "last_name": "Wagner",
  "email": "lisa.wagner@email.com",
  "phone": "+43 690 5678901",
  "registration_date": ISODate("2024-05-12T00:00:00Z"),
  "t_shirt_size": "M",
  "dietary_restrictions": null,
  "manager_id": 4,
  "manager_name": "David Fischer",
  "event_history": [
    {
      "event_id": 1,
      "event_name": "AI Innovation Hackathon 2025",
      "registration_number": "REG-2025-005",
      "registration_date": ISODate("2025-02-25T11:20:00Z")
    }
  ]
}
```

Denormalization Strategy

I denormalized some data to make reads faster. Participant names and emails are copied into the registrations array so I don't need to look them up separately when showing the registration list. I also added an

event history array in participant documents for quick access to their past registrations. This creates some duplicate data, but it's worth it because common queries need fewer database operations.

1.7.2. Expected Execution and Possible Changes

My use case has a 1:2 read-to-write ratio per registration: one query fetches the event document (db.events.findOne), and two updates add the registration to the event's array and the event to the participant's history. This is a write-heavy operation since every registration requires two database updates.

Why This Design Benefits the Use Case

The embedded design retrieves event details, venue information, and all registrations in a single query instead of requiring multiple joins like in SQL. The denormalized registration_count field makes capacity checks instant (registration_count < max_participants) without counting array elements. The registrations array is bounded by max_participants (150), keeping documents small at around 30KB maximum.

Impact of Changing Access Patterns

If read frequency increases significantly (users browsing many events before registering), the current design performs well since everything is already embedded in one document. If write frequency spikes with 100+ concurrent registrations per minute, I would separate registrations into their own collection to avoid document-level lock contention, though this is unlikely for typical hackathon registrations that occur over weeks. If participants frequently update profiles, the denormalized names and emails in registrations become expensive to maintain, but since profile updates are rare (that I assume) compared to registrations, the current design is acceptable.

1.7.3. Five Rules of Thumb

Rule 1: Favor embedding unless there is a compelling reason not to

I embedded venue in events because each event has one venue and the data is small. Registrations are embedded as an array since they are a weak entity with a bounded size (max 150). Event history is embedded in participants for quick access to their registration history.

Rule 2: The need to access an object on its own is a compelling reason not to embed it

Participants are stored in a separate collection because they need independent access for profile updates and are reused across multiple events. If I embedded them in events, updating Anna Mueller's email would require changing multiple event documents. The separate collection maintains a single source of truth.

Rule 3: High-cardinality arrays are a compelling reason not to embed

All my arrays have low cardinality, so embedding is safe. Registrations have a maximum of 150 entries, and event_history typically contains 5-10 events. If team submissions grew to hundreds of items, I would move them to a separate collection to avoid document bloat.

Rule 4: Consider the write/read ratio when denormalizing

I denormalized participant names in registrations because profile updates are rare but registration lists are viewed frequently (1:1000 ratio). I also denormalized registration_count for instant capacity checks. I did not denormalize full profiles (t-shirt size, dietary restrictions) since these change more often and are rarely needed.

Rule 5: Structure your data to match application queries and updates

My event-centric design matches the primary query: users browse events then register. One query (db.events.findOne) returns event details, venue, and all registrations. I did not optimize for "find all participants across events" because that's not how users interact with the system.