Comprehensive Database Design Report for CampusFlow

Koorosh Asil Gharehbaghi (Student ID: 40124463)
Yasin Ahmadi (Student ID: 40115203)
Department of Computer Science, Faculty of Mathematics
K. N. Toosi University of Technology

June 2025

Abstract

This comprehensive report provides a detailed summary of the relational database design for **CampusFlow**, a university management system. It thoroughly describes the core entities, their attributes, and the intricate relationships established to manage academic operations efficiently. The report delves into the foundational principles guiding key design decisions, the rigorous normalization process applied to achieve at least Third Normal Form (3NF), and the strategic implementation of advanced SQL features, including views, stored procedures, scalar functions, and triggers. The overarching goal of this design is to ensure robust data integrity, minimize redundancy, optimize data access, and provide a scalable foundation for future university system enhancements.

1 Introduction

The digital transformation of educational institutions necessitates a well-structured and reliable data management system. The primary objective of the **CampusFlow** database design is to create a robust, scalable, and efficient information repository for managing critical academic data within a university environment. This system is designed to support various operations, including student enrollment, course scheduling, instructor assignments, and departmental management. The design emphasizes data consistency, integrity, and the ease of retrieval for reporting and analytical purposes. This document details the architectural decisions, the schema, and the advanced functionalities implemented to meet these objectives.

2 Database Schema: Structure and Normalization

The CampusFlow database schema is comprised of seven interconnected tables, each meticulously designed to encapsulate a specific entity within the university context. This modular approach is fundamental to achieving high levels of data integrity and minimizing redundancy. The schema's adherence to normalization principles ensures efficient data storage and retrieval.

2.1 Table Definitions and Attribute Details

Each table is defined with specific attributes, primary keys (PKs) for unique identification, and foreign keys (FKs) to establish relationships, thereby enforcing referential integrity.

- **Department**
  - `DepartmentID` (`INT`, PK): A unique numerical identifier for each academic department. This serves as the primary key, ensuring that each department record is distinct.
  - `DepartmentName` (`NVARCHAR(50)`, `NOT NULL`, `UNIQUE`): The official name of the department. This field is mandatory and unique to prevent duplicate department entries.
  - `OfficeLocation` (`NVARCHAR(50)`): The physical location of the department's main office. This field can be null if a specific office location is not yet assigned.
- **Instructor**
  - `InstructorID` (`INT`, PK): A unique numerical identifier for each instructor.
  - `FirstName` (`NVARCHAR(50)`, `NOT NULL`): The instructor's first name.
  - `LastName` (`NVARCHAR(50)`, `NOT NULL`): The instructor's last name.
  - `Email` (`NVARCHAR(100)`, `NOT NULL`, `UNIQUE`): The instructor's unique university email address. This is critical for communication and identification.
  - `Salary` (`DECIMAL(10,2)`): The instructor's annual salary, stored with two decimal places for precision.
  - `DepartmentID` (`INT`, FK): A foreign key referencing `DepartmentID` in the `Department` table. This establishes a "teaches in" relationship, indicating which department the instructor belongs to. It is `NOT NULL` to ensure every instructor is associated with a department.
- **Student**
  - `StudentID` (`INT`, PK): A unique numerical identifier for each student.
  - `FirstName` (`NVARCHAR(50)`, `NOT NULL`): The student's first name.
  - `LastName` (`NVARCHAR(50)`, `NOT NULL`): The student's last name.
  - `Email` (`NVARCHAR(100)`, `NOT NULL`, `UNIQUE`): The student's unique university email address. Essential for communications.

- `MajorDeptID` (`INT`, FK): A foreign key referencing `DepartmentID` in the `Department` table. This indicates the student's declared major. It can be NULL if a student has not yet declared a major.
- `EnrollmentYear` (`SMALLINT`, `NOT NULL`): The academic year in which the student first enrolled in the university.

- **Course**
  - `CourseID` (`INT`, PK): A unique numerical identifier for each course offered.
  - `CourseName` (`NVARCHAR(50)`, `NOT NULL`): The official name of the course.
  - `Credits` (`DECIMAL(3,1)`, `NOT NULL`): The number of academic credits awarded upon successful completion of the course, stored with one decimal place.
  - `Description` (`NVARCHAR(200)`): A brief summary of the course content. This field can be NULL.
  - `DepartmentID` (`INT`, FK): A foreign key referencing `DepartmentID` in the `Department` table. This signifies which department offers the course. It is `NOT NULL`.

- **Prerequisite**
  - `CourseID` (`INT`, PK, FK): A foreign key referencing the `CourseID` of the course that requires a prerequisite. Part of the composite primary key.
  - `PrereqCourseID` (`INT`, PK, FK): A foreign key referencing the `CourseID` of the prerequisite course. Part of the composite primary key.
  - **Composite Primary Key**: The combination of (`CourseID`, `PrereqCourseID`) forms the primary key, ensuring that each prerequisite relationship is uniquely defined. This table resolves the many-to-many relationship between courses and their prerequisites.

- **Section**
  - `SectionID` (`INT`, PK): A unique numerical identifier for each specific offering (section) of a course.
  - `CourseID` (`INT`, FK): A foreign key referencing `CourseID` in the `Course` table, linking the section to its course. It is `NOT NULL`.
  - `Semester` (`NVARCHAR(10)`, `NOT NULL`): The academic semester (e.g., 'Fall', 'Spring', 'Summer') in which the section is offered.
  - `Year` (`SMALLINT`, `NOT NULL`): The academic year in which the section is offered.
  - `InstructorID` (`INT`, FK): A foreign key referencing `InstructorID` in the `Instructor` table, indicating the instructor assigned to teach this section. It is `NOT NULL`.
  - `Capacity` (`INT`, `NOT NULL`): The maximum number of students allowed to enroll in this section.
  - `CurrentEnrollmentCount` (`INT`, `DEFAULT 0`): A newly added attribute (for advanced features) that tracks the live count of students currently enrolled in this specific section. This is initialized to 0.

- **Enrollment**

- `EnrollmentID` (`INT`, PK): A unique numerical identifier for each enrollment transaction.
- `StudentID` (`INT`, FK): A foreign key referencing `StudentID` in the `Student` table, indicating the student involved in the enrollment. It is `NOT NULL`.
- `SectionID` (`INT`, FK): A foreign key referencing `SectionID` in the `Section` table, indicating the section the student is enrolled in. It is `NOT NULL`.
- `EnrollmentDate` (`DATE`, `NOT NULL`): The date when the student enrolled in the section.
- `Grade` (`NVARCHAR(2)`): The final grade received by the student in the section (e.g., 'A+', 'B', 'C-'). This field is nullable as a grade is not assigned at the time of enrollment.
- **Unique Constraint**: While not explicitly a composite PK, a unique constraint on (`StudentID`, `SectionID`) could be added to prevent a student from enrolling in the same section multiple times. This is implicitly handled by the 'EnrollStudentInSection' stored procedure.

2.2 Normalization to Third Normal Form (3NF)

The CampusFlow database design rigorously adheres to normalization principles, specifically achieving at least the Third Normal Form (3NF). This process is crucial for minimizing data redundancy, improving data integrity, and enhancing database flexibility.

✓ **First Normal Form (1NF)**: All tables meet 1NF criteria. Each column contains atomic (indivisible) values, and there are no repeating groups of columns. For example, a student's name is split into `FirstName` and `LastName`, and each record has a unique primary key.

✓ **Second Normal Form (2NF)**: 2NF requires that all non-key attributes in a table are fully functionally dependent on the entire primary key. In our design, all tables with composite primary keys (e.g., `Prerequisite`) ensure that all non-key attributes (if any existed) would depend on the full composite key. For tables with single-column primary keys (like `Department`, `Instructor`, `Student`, `Course`, `Section`, `Enrollment`), 2NF is inherently satisfied as there is only one key to depend on.

✓ **Third Normal Form (3NF)**: 3NF builds upon 2NF by eliminating transitive dependencies. A transitive dependency occurs when a non-key attribute is dependent on another non-key attribute. In CampusFlow, this is diligently avoided. For instance, in the `Instructor` table, instructor details are directly dependent on `InstructorID`. Information about the department (`DepartmentName`, `OfficeLocation`) is not redundantly stored within the `Instructor` table; instead, a foreign key `DepartmentID` links to the separate `Department` table. This ensures that department-specific details are managed and updated in one central location, preventing inconsistencies. The same logic applies across all entities, maintaining a clean and robust data model.

This stringent adherence to 3NF significantly reduces the potential for data anomalies (insertion, update, and deletion anomalies) and makes the database easier to maintain and extend.

3 Key Design Decisions and Underlying Assumptions

The architectural choices for the CampusFlow database were guided by principles of data integrity, scalability, and ease of management.

- **Separate Entities for Each Concept**:

  → **Decision**: Each distinct concept (Department, Instructor, Student, Course, Section, Enrollment, Prerequisite) is represented by its own table.

  → **Justification**: This modular approach aligns with the single responsibility principle. It simplifies updates, avoids redundant data storage, and improves readability of the schema. For example, department details are stored once in the `Department` table, and other tables reference it via `DepartmentID`.

- **Use of Surrogate Primary Keys**:

  → **Decision**: Integer IDs (e.g., `DepartmentID`, `InstructorID`, `StudentID`) are used as primary keys instead of natural keys (like names or emails).

  → **Justification**: Surrogate keys are stable (they don't change), universally unique, and compact, making joins more efficient. Natural keys, while sometimes seemingly intuitive, can be prone to changes (e.g., a student's email might change) or might not always be unique, leading to data integrity issues.

- **Foreign Keys for Relationship Enforcement**:

  → **Decision**: Relationships between tables are explicitly defined using foreign key constraints.

  → **Justification**: This enforces referential integrity at the database level. For instance, it prevents assigning an instructor to a non-existent department or enrolling a student in a non-existent section. This ensures data consistency and prevents orphaned records.

- **Resolution of Many-to-Many Relationships**:

  → **Decision**: Many-to-many relationships are resolved using junction (or associative) tables. The relationship between `Student` and `Section` is handled by `Enrollment`, and between `Course` and `Course` (for prerequisites) by `Prerequisite`.

  → **Justification**: This is a standard relational database practice. Junction tables allow for additional attributes specific to the relationship (e.g., `EnrollmentDate`, `Grade` in `Enrollment`), which cannot be stored if a direct many-to-many relationship were attempted.

- **Realistic Data Constraints**:

  → **Decision**: `NOT NULL` constraints are applied to essential fields, and `UNIQUE` constraints are used where uniqueness is a business rule (e.g., `Email`, `DepartmentName`).

  → **Justification**: These constraints enforce data quality and integrity at the point of data entry, preventing incomplete or illogical records.

- **Capacity Management and `CurrentEnrollmentCount`**:
  - → **Decision**: The `Section` table includes a `Capacity` field (maximum allowed students) and a `CurrentEnrollmentCount` field. A trigger (`trg_UpdateSectionEnrollmentCount_A` is used to automatically update `CurrentEnrollmentCount`.
  - → **Assumptions**: We assume 'Capacity' represents the *maximum* number of students a section can hold, not the remaining seats. We also assume that 'CurrentEnrollmentCount' needs to be kept in real-time sync with new enrollments for quick checks and reporting.
  - → **Justification**: This provides immediate access to enrollment statistics without needing to run aggregation queries on the `Enrollment` table every time, which can improve performance for frequent checks. The trigger ensures data consistency automatically.
- **Nullable Grade Field**:
  - → **Decision**: The `Grade` field in the `Enrollment` table is nullable.
  - → **Justification**: This accommodates the real-world scenario where a student enrolls in a section, but a grade is only assigned after the course is completed.

4 Advanced SQL Features Implemented

To further enhance the database's functionality, efficiency, and manageability, several advanced SQL features have been incorporated. These features abstract complex logic, improve security, and ensure data consistency.

- **View: `StudentEnrollmentDetails`**
  - **Purpose**: To provide a simplified, consolidated view of student academic records by joining data from six different tables: `Student`, `Enrollment`, `Section`, `Course`, `Instructor`, and `Department`.
  - **Technical Details**: The view is defined using a series of `JOIN` clauses to link these tables on their respective foreign keys. It selects specific attributes from each table and renames them with aliases for clarity (e.g., `s.FirstName AS StudentFirstName`). A `LEFT JOIN` is used for `Department` for the major, accommodating students who might not have a major yet.
  - **Benefit**:
    - **Simplification**: Users and applications can query this single view instead of writing complex multi-table join queries, reducing development time and error potential.
    - **Data Security**: By granting access to the view rather than the underlying tables, specific columns can be excluded, enhancing data security.
    - **Consistency**: Ensures that all consumers of this data see a consistent representation of combined enrollment information.
- **Stored Procedure: `EnrollStudentInSection`**

- **Purpose**: To encapsulate the business logic required for enrolling a student into a course section, including critical validation checks.
- **Technical Details**: The procedure takes `@StudentID`, `@SectionID`, and `@EnrollmentDate` as input. It first queries the `Section` and `Enrollment` tables to calculate available capacity and check for existing enrollments. It uses `IF...BEGIN...END` blocks for conditional logic and returns integer status codes (`0` for success, negative for specific errors). A new `EnrollmentID` is generated (using `ISNULL(MAX(EnrollmentID), 0) + 1`), and the new record is inserted into the `Enrollment` table.
- **Benefit**:
  - **Data Integrity**: Centralizes enrollment logic, ensuring all enrollments adhere to defined rules (capacity, no duplicates) regardless of the application making the request.
  - **Security**: Database permissions can be granted only on the stored procedure, abstracting direct table access.
  - **Performance**: Stored procedures are pre-compiled and cached, leading to faster execution times compared to ad-hoc SQL queries.
  - **Maintainability**: Changes to enrollment rules can be made in one place (the procedure) without affecting calling applications.

- **Scalar Function: `GetInstructorSectionCount`**
  - **Purpose**: To calculate and return the total number of sections an individual instructor is currently assigned to teach.
  - **Technical Details**: This function takes an `@InstructorID` as input. It performs a simple `COUNT` aggregation on the `Section` table, filtered by the provided `InstructorID`, and returns the result. `ISNULL` ensures a 0 is returned if no sections are found.
  - **Benefit**:
    - **Reusability**: The calculation logic is encapsulated, allowing it to be used in various queries, views, or other stored procedures without duplication.
    - **Simplicity**: Queries that need an instructor's section count become much cleaner and easier to read.

- **Trigger: `trg_UpdateSectionEnrollmentCount_AfterEnrollment`**
  - **Purpose**: To automatically maintain the `CurrentEnrollmentCount` in the `Section` table whenever a new student enrollment occurs.
  - **Technical Details**: This 'AFTER INSERT' trigger is defined on the `Enrollment` table. When a new row is inserted into `Enrollment`, the trigger fires. It uses the pseudo-table `INSERTED` to identify the `SectionID` of the newly enrolled student(s) and then increments the `CurrentEnrollmentCount` for those sections in the `Section` table.
  - **Benefit**:
    - **Automated Data Consistency**: Ensures that the `CurrentEnrollmentCount` always reflects the actual number of enrollments, reducing manual updates and preventing data inconsistencies.

- **Reduced Application Logic**: The application layer doesn't need to explicitly update the section count; the database handles it automatically.
- **Performance (for Reads)**: Having a pre-calculated count directly in the `Section` table makes querying for section enrollment numbers very fast.

5 Conclusion

The **CampusFlow** database design represents a robust, well-normalized, and extensible solution for managing academic data within a university. Through careful table structuring, adherence to Third Normal Form, and the strategic implementation of advanced SQL features, the design achieves high levels of data integrity, minimizes redundancy, and facilitates efficient data manipulation and retrieval. The integration of views, stored procedures, functions, and triggers not only streamlines common operations but also provides a solid foundation for future enhancements and analytical capabilities. This design serves as a reliable backbone for the university's academic information system.