The component of a database system that prevents both students from being given the last seat in the CCE 223 course is called **concurrency control**. Concurrency control mechanisms ensure that multiple transactions (operations) can occur simultaneously without leading to inconsistencies in the database.

In this scenario, if both students attempt to register for the last seat at the same time, the database system needs to ensure that only one registration is successful. This is typically managed using **locking mechanisms** or **transaction isolation levels**.

For example, when the first student initiates the registration process, the database system might place a lock on the seat record, preventing other transactions from modifying it until the first transaction is complete. If the second student tries to register while the seat is locked, their transaction will either wait until the lock is released or be rejected if the seat is already taken.

This ensures that only one student can successfully register for the last seat, maintaining the integrity of the database.

what is atomicity?

**Atomicity** is a property of database transactions that ensures that a transaction is treated as a single, indivisible unit. This means that either all the operations within the transaction are successfully completed, or none of them are. If any part of the transaction fails, the entire transaction is rolled back, and the database is left unchanged.

In the context of the CCE 223 course registration example, atomicity ensures that if a student's registration process involves multiple steps (e.g., checking seat availability, updating the seat count, and enrolling the student), all these steps must complete successfully. If any step fails (e.g., due to a system error or a constraint violation), the entire transaction is rolled back, and the student is not registered. This prevents partial updates that could lead to inconsistencies in the database.

isolation

**Isolation** is a property of database transactions that ensures that the operations of one transaction are invisible to other transactions until the first transaction is completed. This means that even if multiple transactions are executed concurrently, each transaction operates as if it were the only one running in the system. Isolation helps prevent issues such as dirty reads, non-repeatable reads, and phantom reads.

There are different levels of isolation that databases can enforce, each providing a different balance between consistency and performance:

1. **Read Uncommitted**: The lowest isolation level. Transactions can see uncommitted changes made by other transactions, which can lead to dirty reads.
2. **Read Committed**: Ensures that any data read is committed at the moment it is read. This prevents dirty reads but can still result in non-repeatable reads or phantom reads.
3. **Repeatable Read**: Guarantees that if a transaction reads the same data multiple times, it will see the same values each time. This prevents dirty reads and non-repeatable reads but not phantom reads.
4. **Serializable**: The highest isolation level. It ensures complete isolation from other transactions, as if transactions were executed serially one after the other. This prevents dirty reads, non-repeatable reads, and phantom reads.

In the context of the CCE 223 course registration example, isolation ensures that if two students are trying to register for the last seat simultaneously, each student's transaction is isolated from the other until it is completed. This prevents scenarios where one student might see an outdated seat availability status due to the concurrent actions of the other student.

ACID property

The **ACID properties** are a set of principles that ensure reliable processing of database transactions. ACID stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**. These properties are crucial for maintaining the integrity and reliability of data in a database system, especially in environments where multiple transactions occur concurrently.

1. **Atomicity**:
   * Ensures that a transaction is treated as a single, indivisible unit.
   * Either all operations within the transaction are completed successfully, or none are.
   * If any part of the transaction fails, the entire transaction is rolled back.
2. **Consistency**:
   * Ensures that a transaction brings the database from one valid state to another valid state.
   * All data is consistent with defined rules, such as constraints, triggers, and cascades.
   * Prevents corruption of data by ensuring that only valid data is written to the database.
3. **Isolation**:
   * Ensures that the operations of one transaction are isolated from those of other transactions.
   * Transactions are executed as if they were running serially, even though they may be running concurrently.
   * Prevents issues like dirty reads, non-repeatable reads, and phantom reads.
4. **Durability**:
   * Ensures that once a transaction has been committed, it remains committed even in the event of a system failure.
   * Changes made by the transaction are permanently stored in the database.
   * Typically achieved through techniques like write-ahead logging and database backups.

In the context of the CCE 223 course registration example, the ACID properties ensure that:

* **Atomicity**: The registration process is completed fully or not at all.
* **Consistency**: The database remains in a valid state, with no overbooking of seats.
* **Isolation**: Each student's registration is isolated from others, preventing conflicts.
* **Durability**: Once a student is registered, the registration is permanent and will not be lost due to system failures.

image.png

PNG 243.77KB

**b) DBMS Three-Schema Architecture**

The **Three-Schema Architecture** in DBMS is a framework used to separate the user applications and the physical database. It consists of three levels:

1. **External Schema (View Level)**:
   * This is the highest level of abstraction and describes how users view the data.
   * Different users can have different views of the same data.
   * Example: In a university database, students might see their grades, while professors see course enrollments.
2. **Conceptual Schema (Logical Level)**:
   * This level describes the structure of the entire database for a community of users.
   * It hides the physical storage details and concentrates on describing entities, data types, relationships, and constraints.
   * Example: A conceptual schema might define entities like Student, Course, and Enrollment, along with their relationships.
3. **Internal Schema (Physical Level)**:
   * This is the lowest level of abstraction and describes how the data is physically stored in the database.
   * It deals with data storage structures, access methods, and indexes.
   * Example: The internal schema might specify that student records are stored in a B-tree structure on disk.

**c) Simplified Database System Environment**

A simplified database system environment typically includes the following components:

1. **Database**: The actual data stored in a structured format.
2. **DBMS Software**: Manages the database and provides an interface for users and applications.
3. **Users**: End-users, database administrators, and application developers.
4. **Applications**: Software that interacts with the database through the DBMS.

**Sketch**:

Copy

+-------------------+

| Users |

+-------------------+

|

v

+-------------------+

| Applications |

+-------------------+

|

v

+-------------------+

| DBMS Software |

+-------------------+

|

v

+-------------------+

| Database |

+-------------------+

**Research Topics in Database Systems**:

* Big Data Management
* NoSQL Databases
* Cloud Database Systems
* Blockchain and Databases
* Data Privacy and Security
* Machine Learning in Databases
* Real-Time Data Processing

**d) Difference Between DBMS and RDBMS**

**DBMS (Database Management System)**:

* Manages databases on the computer network and system hard disks.
* Data is stored in a hierarchical or navigational form.
* Does not support relationships between tables.
* Example: File systems, XML.

**RDBMS (Relational Database Management System)**:

* Manages databases that are based on the relational model.
* Data is stored in tables, and relationships between tables are established using keys.
* Supports ACID properties for transaction management.
* Example: MySQL, PostgreSQL, Oracle.

**Filesystem DBMS Example**:

* A filesystem DBMS might store data in flat files, such as CSV files.
* Example: A simple student database stored in a CSV file where each line represents a student record with fields like ID, Name, and Grade.

This architecture and differentiation help in understanding how database systems are structured and how they manage data efficiently.

image.png

PNG 704.39KB

**a) Component Preventing Both Students from Getting the Last Seat**

The component of a database system that prevents both students from being given the last seat in the CCB 224 course is **concurrency control**. Concurrency control mechanisms ensure that multiple transactions can occur simultaneously without leading to inconsistencies.

**Example**: When two students attempt to register for the last seat simultaneously, the database system uses locking mechanisms to ensure that only one transaction can modify the seat record at a time. If the first student's transaction locks the seat, the second student's transaction will either wait or be rejected, ensuring only one student gets the seat.

**b) Why Studying Databases? Purposes of Database Systems Courses**

Studying databases is essential because:

1. **Data Management**: Efficiently store, retrieve, and manage large volumes of data.
2. **Data Integrity**: Ensure accuracy and consistency of data through constraints and validation rules.
3. **Data Security**: Protect data from unauthorized access and breaches.
4. **Decision Making**: Provide reliable data for analysis and decision-making processes.
5. **Application Development**: Support the development of data-driven applications.

**Purposes of Database Systems Courses**:

* Understand fundamental database concepts and models.
* Learn to design and implement databases.
* Gain skills in querying and managing data using SQL.
* Explore advanced topics like concurrency control, transaction management, and database security.

**c) Levels of Abstraction in a University Database System**

1. **Physical Level**: Describes how data is stored on physical storage devices (e.g., hard disks). Example: Storage of student records in a specific file format.
2. **Logical Level**: Describes the structure of the entire database, including tables, relationships, and constraints. Example: Defining entities like Student, Course, and Enrollment.
3. **View Level**: Describes how different users view the data. Example: Students see their grades, while professors see course enrollments.

**d) Emergence of NoSQL Systems and Contrast with Traditional Databases**

**NoSQL Systems Emerged**:

* To handle large volumes of unstructured or semi-structured data.
* To provide scalability and flexibility for web applications and big data.
* To support distributed computing environments.

**Contrast with Traditional Databases**:

* **Data Model**: NoSQL uses non-relational models (e.g., document, key-value, graph).
* **Scalability**: NoSQL systems are designed for horizontal scaling.
* **Consistency**: NoSQL often sacrifices strict consistency for availability and partition tolerance (CAP theorem).
* **Schema**: NoSQL is schema-less, allowing flexible data structures.

**e) Different Users for the University Database Schema**

1. **Students**:
   * **Applications**: Course registration, grade viewing.
   * **User Category**: End-users.
   * **Interface**: Web or mobile application with a user-friendly interface.
2. **Professors**:
   * **Applications**: Course management, grade submission.
   * **User Category**: End-users.
   * **Interface**: Web application with administrative features.
3. **Administrators**:
   * **Applications**: Database maintenance, user management.
   * **User Category**: Database administrators.
   * **Interface**: Command-line tools or administrative dashboards.
4. **Academic Advisors**:
   * **Applications**: Student progress tracking, course planning.
   * **User Category**: End-users.
   * **Interface**: Web application with reporting features.
5. **Developers**:
   * **Applications**: Application development, database integration.
   * **User Category**: Application developers.
   * **Interface**: Integrated Development Environments (IDEs) and database management tools.

This structured approach helps in understanding the roles and needs of different users interacting with the university database system.

for question a. which property is used from acid property?

For question **a)**, the property from the **ACID** properties that is primarily used to prevent both students from being given the last seat is **Isolation**.

**Explanation:**

* **Isolation** ensures that concurrent transactions (e.g., two students trying to register for the same seat) are executed in a way that they do not interfere with each other. It ensures that one transaction is not able to see the intermediate state of another transaction until it is completed.
* In this case, when two students attempt to register for the last seat simultaneously, the database system uses isolation to ensure that one student's registration process is completed before the other begins. This prevents both students from being able to register for the same seat at the same time.

**How Isolation Works in This Scenario:**

1. **Transaction 1 (Student A)** starts and locks the seat record.
2. **Transaction 2 (Student B)** attempts to access the same seat record but is blocked or delayed until Transaction 1 completes.
3. If Transaction 1 successfully registers for the seat, Transaction 2 will either fail (if the seat is now taken) or be allowed to proceed if the seat is still available.

Thus, **Isolation** ensures that only one student can successfully register for the last seat, maintaining data consistency and preventing conflicts.

if two students is pressing the registration button exactly same time then how to control it ?

When two students press the registration button at exactly the same time, the database system uses **concurrency control mechanisms** to handle the situation and ensure that only one student successfully registers for the last seat. Here's how it works:

**1. Locking Mechanism:**

* The database system places a **lock** on the seat record as soon as one student's transaction begins.
* If the second student tries to access the same seat while it is locked, their transaction will either:
  + **Wait** until the lock is released (if the first transaction fails or completes).
  + **Fail** immediately if the seat is already taken by the first student.

**Example**:

* Student A's transaction locks the seat and proceeds to register.
* Student B's transaction is blocked until Student A's transaction completes.
* If Student A successfully registers, Student B's transaction will fail because the seat is no longer available.

**2. Atomicity and Isolation (ACID Properties):**

* **Atomicity** ensures that the registration process is treated as a single, indivisible operation. If any part of the transaction fails (e.g., the seat is already taken), the entire transaction is rolled back.
* **Isolation** ensures that the two transactions do not interfere with each other. Each transaction operates as if it is the only one running, even though they are happening concurrently.

**3. Transaction Management:**

* The database system processes transactions in a **serializable** order, meaning it ensures that the final result is as if the transactions occurred one after the other, even if they were executed concurrently.
* The system uses techniques like **timestamp ordering** or **optimistic concurrency control** to manage simultaneous requests.

**4. Example Workflow:**

* **Step 1**: Both Student A and Student B press the registration button at the same time.
* **Step 2**: The database system assigns a timestamp or priority to each transaction (e.g., Student A's request is processed first).
* **Step 3**: Student A's transaction locks the seat, checks availability, and registers for the seat.
* **Step 4**: Student B's transaction attempts to access the seat but finds it locked or already taken.
* **Step 5**: Student B's transaction either waits (if the seat is still being processed) or fails (if the seat is already taken).

**5. Handling in Real-World Systems:**

* **Web Applications**: The application layer (e.g., the registration system) may also implement additional checks to handle high concurrency, such as:
  + Queueing requests and processing them one at a time.
  + Displaying a message to the second student indicating that the seat is no longer available.
* **Database Systems**: The DBMS ensures that the seat record is updated atomically and consistently, even under high concurrency.

By using these mechanisms, the database system ensures that only one student can successfully register for the last seat, even if multiple students attempt to register at the same time. This maintains data integrity and prevents overbooking.

say for example two user preses for same seat in same time ,who will get the seat and how will it be processed and feasible

When two users press the registration button for the same seat at the **exact same time**, the database system uses **concurrency control mechanisms** to determine who gets the seat and how the process is handled. Here's a detailed explanation of how this works and who gets the seat:

**1. How the Database Processes Concurrent Requests**

* The database system uses **locking** or **transaction scheduling** to ensure that only one transaction can modify the seat record at a time.
* When two users press the button simultaneously, the database assigns a **priority** to each transaction (e.g., based on timestamps or the order in which the requests are received by the system).
* The transaction with the **higher priority** (e.g., the one that arrived first, even by milliseconds) is processed first.

**2. Who Gets the Seat?**

* The user whose transaction is processed first will get the seat.
* The second user's transaction will either:
  + **Wait** until the first transaction completes (if the seat is still available).
  + **Fail** immediately if the seat is already taken by the first user.

**3. Feasibility and Fairness**

* The system ensures **fairness** by processing transactions in the order they are received (e.g., based on timestamps).
* If both requests arrive at the **exact same time**, the database system may use internal mechanisms (e.g., transaction IDs or system-level priorities) to decide which transaction to process first.
* In practice, it is highly unlikely for two requests to arrive at the **exact same nanosecond**, so the system will almost always have a way to determine the order.

**4. Step-by-Step Process**

* **Step 1**: Both User A and User B press the registration button at the same time.
* **Step 2**: The database system assigns a timestamp or priority to each request (e.g., User A's request is processed first).
* **Step 3**: User A's transaction locks the seat record, checks availability, and registers for the seat.
* **Step 4**: User B's transaction attempts to access the seat but finds it locked or already taken.
* **Step 5**: User B's transaction either:
  + Waits for User A's transaction to complete (if the seat is still being processed).
  + Fails immediately if the seat is already taken by User A.

**5. Real-World Feasibility**

* **Locking Mechanism**: The database ensures that only one transaction can modify the seat record at a time, preventing conflicts.
* **Atomicity**: If User A's transaction succeeds, User B's transaction will fail, ensuring that only one user gets the seat.
* **User Feedback**: The system will notify User B that the seat is no longer available, providing a seamless user experience.

**6. Example Scenario**

* **User A** and **User B** both click the "Register" button for the last seat in a course at the same time.
* The database system processes User A's request first (based on timestamps or priority).
* User A successfully registers for the seat.
* User B's request is rejected, and they are notified that the seat is no longer available.

**7. Ensuring Fairness**

* To ensure fairness, the system can implement additional measures, such as:
  + **Queueing**: Requests are placed in a queue and processed in the order they are received.
  + **Random Selection**: If two requests arrive at the exact same time, the system randomly selects one to process first.
  + **Retry Mechanism**: If a user's request fails, they can retry for other available seats.

By using these mechanisms, the database system ensures that the seat allocation process is **fair, consistent, and feasible**, even when multiple users attempt to register for the same seat simultaneously.

attribute, as used in the E-R model, can be characterized by the following attribute types.

• **Simple** and **composite** attributes. In our examples thus far, the attributes have been

**simple**; that is, they have not been divided into subparts. **Composite** attributes, on

the other hand, can be divided into subparts (i.e., other attributes). For example, an attribute *name* could be structured as a composite attribute consisting of

*first name*, *middle initial*, and *last name*. Using composite attributes in a design

schema is a good choice if a user will wish to refer to an entire attribute on some

occasions, and to only a component of the attribute on other occasions. Suppose

we were to add an address to the *student* entity-set. The address can be defined

as the composite attribute *address* with the attributes *street*, *city*, *state*, and *postal*

*code*.1 Composite attributes help us to group together related attributes, making the modeling cleaner.

Note also that a composite attribute may appear as a hierarchy. In the composite attribute *address*, its component attribute *street* can be further divided into *street*

*number*, *street name*, and *apartment number*. Figure 6.7 depicts these examples of composite attributes for the *instructor* entity set.

1We assume the address format used in the United States, which includes a numeric postal code called a zip code.

• **Single-valued** and **multivalued** attributes. The attributes in our examples all have

a single value for a particular entity. For instance, the *student ID* attribute for a

specific student entity refers to only one student *ID*. Such attributes are said to be

**single valued**. There may be instances where an attribute has a set of values for

a specific entity. Suppose we add to the *instructor* entity set a *phone number* attribute. An *instructor* may have zero, one, or several phone numbers, and different

instructors may have different numbers of phones. This type of attribute is said to

be **multivalued**. As another example, we could add to the *instructor* entity set an

attribute *dependent name* listing all the dependents. This attribute would be multivalued, since any particular instructor may have zero, one, or more dependents.

• **Derived attributes**. The value for this type of attribute can be derived from the values of other related attributes or entities. For instance, let us say that the *instructor*

entity set has an attribute *students advised*, which represents how many students

an instructor advises. We can derive the value for this attribute by counting the

number of *student* entities associated with that instructor.

As another example, suppose that the *instructor* entity set has an attribute *age* that indicates the instructor’s age. If the *instructor* entity set also has an attribute *date of birth*, we can calculate *age* from *date of birth* and the current date. Thus, *age* is a derived attribute. In this case, *date of birth* may be referred to as a *base* attribute, or a *stored* attribute. The value of a derived attribute is not stored but is computed when required.

<https://www.geeksforgeeks.org/types-of-keys-in-relational-model-candidate-super-primary-alternate-and-foreign/>

**6.2.1 Entity Sets**

An **entity** is a “thing” or “object” in the real world that is distinguishable from all other objects. For example, each person in a university is an entity. An entity has a set of properties, and the values for some set of properties must uniquely identify an entity. For instance, a person may have a *person id* property whose value uniquely identifies that person. Thus, the value 677-89-9011 for *person id* would uniquely identify one particular person in the university. Similarly, courses can be thought of as entities, and *course id* uniquely identifies a course entity in the university. An entity may be concrete, such

**6.2 The Entity-Relationship Model 245**

as a person or a book, or it may be abstract, such as a course, a course offering, or a flight reservation.

An **entity set** is a set of entities of the same type that share the same properties, or attributes. The set of all people who are instructors at a given university, for example, can be defined as the entity set *instructor*. Similarly, the entity set *student* might represent the set of all students in the university.

In the process of modeling, we often use the term *entity set* in the abstract, without referring to a particular set of individual entities. We use the term **extension** of the entity set to refer to the actual collection of entities belonging to the entity set. Thus, the set of actual instructors in the university forms the extension of the entity set *instructor*. This distinction is similar to the difference between a relation and a relation instance, which we saw in Chapter 2.

Entity sets do not need to be disjoint. For example, it is possible to define the entity set *person* consisting of all people in a university. A *person* entity may be an *instructor* entity, a *student* entity, both, or neither.

An entity is represented by a set of **attributes**. Attributes are descriptive properties possessed by each member of an entity set. The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the *instructor* entity set are *ID*, *name*, *dept name*, and *salary*. In real life, there would be further attributes, such as street number, apartment number, state, postal code, and country, but we generally omit them to keep our examples simple. Possible attributes of the *course* entity set are *course id*, *title*, *dept name*, and *credits*.

In this section we consider only attributes that are **simple**— those not divided into subparts. In Section 6.3, we discuss more complex situations where attributes can be composite and multivalued.

Each entity has a **value** for each of its attributes. For instance, a particular *instructor* entity may have the value 12121 for *ID*, the value Wu for *name*, the value Finance for *dept name*, and the value 90000 for *salary*.

The *ID* attribute is used to identify instructors uniquely, since there may be more than one instructor with the same name. Historically, many enterprises found it convenient to use a government-issued identification number as an attribute whose value uniquely identifies the person. However, that is considered bad practice for reasons of security and privacy. In general, the enterprise would have to create and assign its own unique identifier for each instructor.

A database thus includes a collection of entity sets, each of which contains any number of entities of the same type. A database for a university may include a number of other entity sets. For example, in addition to keeping track of instructors and students, the university also has information about courses, which are represented by the entity set *course* with attributes *course id*, *title*, *dept name* and *credits*. In a real setting, a university database may keep dozens of entity sets.

An entity set is represented in an E-R diagram by a **rectangle**, which is divided into two parts. The first part, which in this text is shaded blue, contains the name of

**246 Chapter 6 Database Design Using the E-R Model**

|  |
| --- |
| *instructor* |
| *ID name salary* |

|  |
| --- |
| *student* |
| *ID name tot\_cred* |

**Figure 6.1** E-R diagram showing entity sets *instructor* and *student*.

the entity set. The second part contains the names of all the attributes of the entity set. The E-R diagram in Figure 6.1 shows two entity sets *instructor* and *student*. The attributes associated with *instructor* are *ID*, *name*, and *salary*. The attributes associated with *student* are *ID*, *name*, and *tot cred*. Attributes that are part of the primary key are underlined (see Section 6.5).