UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

CS411 - SQL: Transactions, Constraints, and Triggers

illinois.edu

FUD

- Fear
- Uncertainty
- Doubt

Suggestions

- You're learning SQL
  - install MySQL and start playing around
- You learned Python
  - install Django and start playing around
    - Pinterest, Instagram, The Onion,

Announcements

- Project Track 1
  - Stage 1 due today
  - Stage 2 due Friday
  - Stage 3 due Monday
- HW1 due Saturday

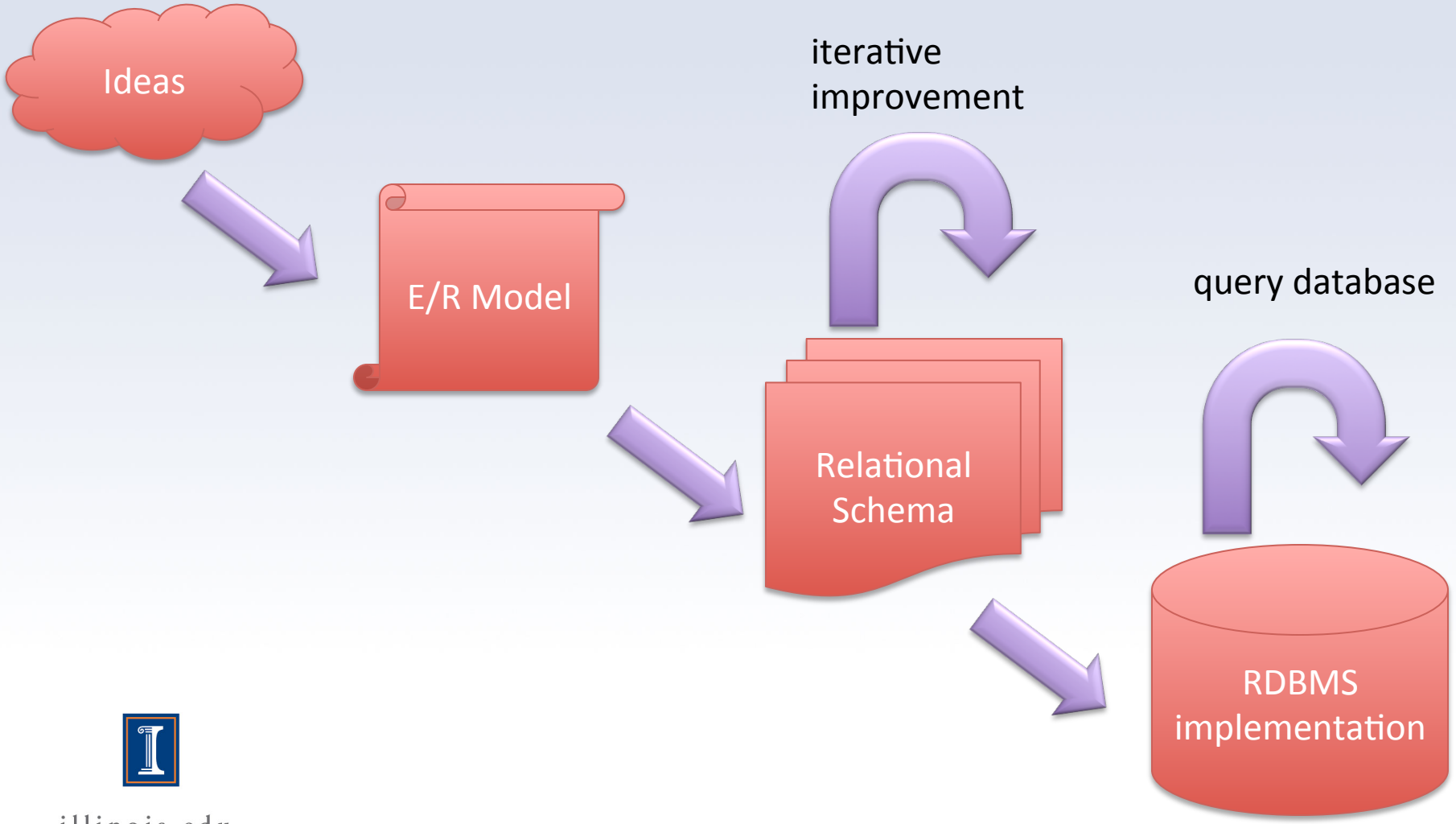
Announcements

- DB programming tutorial session
  - tomorrow at 6:00pm in SC0216
- Midterm 1 March 1st

Review

- What is a weak entity set?
- What ways did we have to translate “isa” relationships into relations?
- What new things did we learn to do with SQL?

Design Process



Dynamic Databases

- We can now alter the data in our databases
- DBMSs can be accessed by multiple users
- This can cause ***problems***

A tale of two users

- Bob and Jill live in Chicago
- They both want to see Beyonce perform
- There is only one seat left, seat 102
- They both have access to the concert database

A tale of two users

Bob

```
SELECT seatNum
FROM Concert
WHERE act="Beyonce"
AND City="Chicago"
AND owner="NOBODY"

Result: 102
```

Jill

```
SELECT seatNum
FROM Concert
WHERE act="Beyonce"
AND City="Chicago"
AND owner="NOBODY"

Result: 102
```

A tale of two users

Bob

```
UPDATE Concert
SET owner=“Bob”
WHERE seatNum=102
AND act=“Beyonce”
AND City=“Chicago”

Result: Seat 102
belongs to Bob
```

Jill

```
UPDATE Concert
SET owner=“Jill”
WHERE seatNum=102
AND act=“Beyonce”
AND City=“Chicago”

Result: Seat 102
belongs to Jill
```

Tragedy!

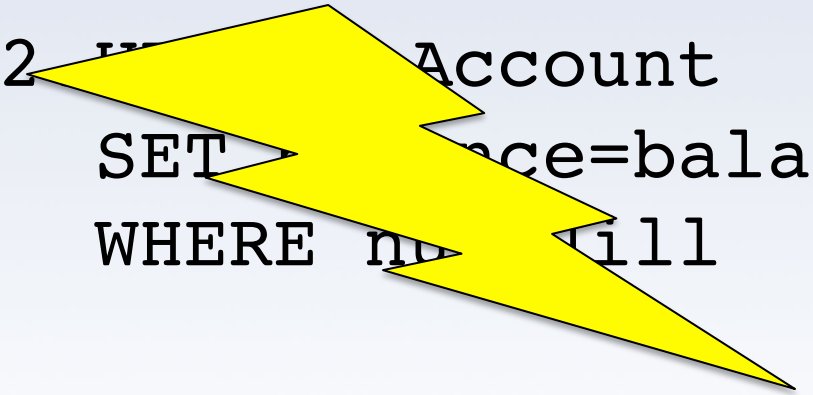
- Both Bob and Jill think they'll be sitting in seat 102
- Only Jill's name is in the database
- Bob is heartbroken

Bob's bad day

- Jill offers to sell the ticket to Bob for $500
- Bob uses an online banking system to transfer the money to Jill's account
- The system executes the following queries

Bob's bad day

```
1. UPDATE Account
   SET balance=balance-500
   WHERE name=Bob
2. UPDATE Account
   SET balance=balance+500
   WHERE name=Bill
```



Bob's bad day

- The second query never got executed
- Bob's account has $500 less
- Jill's account balance is the same
- Jill refuses to give Bob the ticket

Poor Bob!

- What went wrong?
- Cast your mind back to day 1…

ACID test

- Atomicity
- Consistency
- Isolation
- Durability

ACID test

- Atomicity - “all or nothing”
- Consistency - “maintain constraints”
- Isolation - “don’t interfere”
- Durability - “don’t lose anything”

ACID test

- **Atomicity - “all or nothing”**
- Consistency - “maintain constraints”
- **Isolation - “don’t interfere”**
- Durability - “don’t lose anything”

Isolation

- In the Great Ticket Race, Bob's queries should have happened before Jill's
- Bob's queries should have been executed ***sequentially***
- Bob's queries should have been ***isolated*** from Jill's queries

A happy tale

Bob:

```
SELECT seatNum FROM Concert WHERE act="Beyonce"
AND City="Chicago" AND owner="NOBODY"

UPDATE Concert SET owner="Bob" WHERE seatNum=102
AND act="Beyonce" AND City="Chicago"
```

Jill:

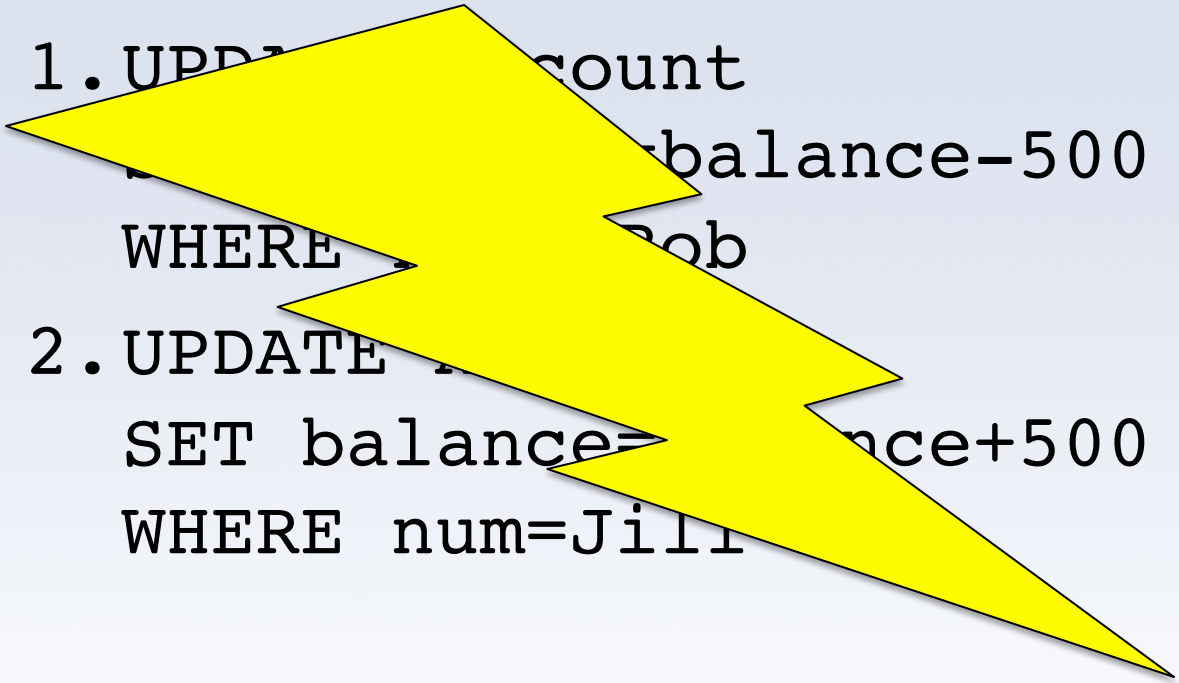
```
SELECT seatNum FROM Concert WHERE act="Beyonce"
AND City="Chicago" AND owner="NOBODY"
```

Result: Nothing!

Atomicity

- In the Online Banking Debacle, either:
  - Both queries should execute
  - Neither query should execute
- They should execute as a unit, or ***atomically***

Bob's okay day



Transactions

- Group queries together into one logical task
- Transactions are guaranteed to execute ***atomically***
- We can control the level of ***isolation***

Transactions

- First, we START the transaction
- Then we specify the queries we want to be a part of the transaction
- Then we COMMIT the transaction

Example

```
START TRANSACTION;

SELECT seatNum FROM Concert WHERE act="Beyonce"
AND City="Chicago" AND owner="NOBODY";

UPDATE Concert SET owner="Bob" WHERE seatNum=102
AND act="Beyonce" AND City="Chicago";

COMMIT;
```

Bob owns seat 102

Transactions

- We can decide not to COMMIT the transaction
- Instead, we can ROLLBACK the transaction
- It's like the transaction never happened

Example

```
START TRANSACTION;

SELECT seatNum FROM Concert WHERE act="Beyonce"
AND City="Chicago" AND owner="NOBODY";

UPDATE Concert SET owner="Bob" WHERE seatNum=102
AND act="Beyonce" AND City="Chicago";

ROLLBACK;
```

NOBODY owns seat 102

Read phenomena

- When we relax total isolation (serializable) different read interactions are possible
- Consider two transactions T_1 and T_2
- Depending on what T_2 writes, what does T_1 see?

Read phenomena

- Three types:
  1. Phantom reads
  2. Nonrepeatable reads
  3. Dirty reads

Read phenomena

- To illustrate these phenomena, I will use the following example table

id	name	age
1	Joe	20
2	Jill	25

- Flagrantly stolen from the Wikipedia entry on isolation

Phantom Reads

- If we execute T_1 follow by T_2 then repeat T_1, the rows returned are different
- Values of old rows remain the same
- We're seeing different tuples, but the values of the ones we read remain the same

Phantom Reads

- If we execute T_1 follow by T_2 then repeat T_1, the rows returned are different

```
T1: Select * FROM Users
```

id	name	age
1	Joe	20
2	Jill	25

```
T2: INSERT INTO Users VALUES (3,'Bob',27)
```

```
T1: Select * FROM Users
```

id	name	age
1	Joe	20
2	Jill	25
3	Bob	27

Nonrepeatable Reads

- If we execute T_1 follow by T_2 then repeat T_1, the values returned are different
- In the middle of processing T_1, T_2 could be modifying the values of the tuples we've seen

Nonrepeatable Reads

- If we execute T_1 follow by T_2 then repeat T_1, the values returned are different

```
T1: Select * FROM Users
```

id	name	age
1	Joe	20
2	Jill	25

```
T2: UPDATE Users SET age=21 WHERE id=1
```

```
T1: Select * FROM Users
```

id	name	age
1	Joe	21
2	Jill	25

Dirty Reads

- If we execute T_1 follow by T_2 then repeat T_1, the results are different ***even if we roll back*** T_2
- T_1 was allowed to read data that T_2 updated, but hadn't committed

Nonrepeatable Reads

- If we execute T_1 follow by T_2 then repeat T_1, the values returned are different

```
T1: Select * FROM Users
```

id	name	age
1	Joe	20
2	Jill	25

```
T2: UPDATE Users SET age=21 WHERE id=1
```

```
T1: Select * FROM Users
```

id	name	age
1	Joe	21
2	Jill	25

```
T2: ROLLBACK;
```

Isolation Levels

- Four levels of isolation can be specified:
  1. Serializable
  2. Repeatable read
  3. Read committed
  4. Read uncommitted

Isolation Levels

Isolation Level	Phantom	Nonrepeatable	Dirty
Serializable	No	No	No
Repeatable Read	Yes	No	No
Read committed	Yes	Yes	No
Read uncommitted	Yes	Yes	Yes

Read only transactions

- Transactions that only read data won't interfere with other transactions
- If we tell the DBMS when this happens, it can schedule transaction better
- We have a way to specify this:

```
SET TRANSACTION READ ONLY;
```

Moving on…

- Transactions ensure ***atomicity*** and ***isolation***
- What about ***consistency***?

ACID test

- Atomicity ☺
- **Consistency**
- Isolation ☺
- Durability

Keys

- We've already seen how to create keys for relations

```
CREATE TABLE Jedi(
   name CHAR(30) PRIMARY KEY,
   saberID CHAR(30),
   homeWorld char(30)
);
```

Foreign-Keys

- Foreign-Keys are how we establish referential integrity
  - an attribute in our table refers to some other attribute in another relation
  - Used when we want to join tables together

Foreign-Keys

- When we declare foreign keys:
  1. The referenced attribute must be declared either UNIQUE or as the PRIMARY KEY
  2. Values for this attribute must appear as the value of the referenced atribute

Foreign-Keys

- Here is how they are declared in SQL:

```
CREATE TABLE Jedi(
   name CHAR(30) PRIMARY KEY,
   saberID CHAR(30) REFERENCES
   Saber(id),
   homeWorld char(30) REFERENCES
   Planet(name)
);
```

Foreign-Keys

- Don't have to be from another table:

```
CREATE TABLE Jedi(
   name CHAR(30) PRIMARY KEY,
   master char(30) REFERENCES
   Jedi(name)
);
```

Foreign-Key Enforcement

- Foreign keys are enforced whenever a tuple is:
  - inserted
  - deleted
  - updated
- By default, changes that violate foreign-key constraint are rejected

Example

```
INSERT IN   Jedi
VALUES (  Maul   1  34,"Dathomir")
```

Jedi

name	saberID	homeWorld
Yoda	1828	Degoba
Luke	2333	Tatooine
Vader	1818	Tatooine
Obi-wan	1033	Stewjan

Planet

name
Degoba
Tatooine
Stewjan

Other policies

- We can specify other enforcement policies
  1. Cascade - if the attribute is modified, so is the referenced attribute
  2. Set-Null - if the attribute is modified, the referenced attribute is set to null

Example

```
CREATE TABLE Jedi(
   name CHAR(30) PRIMARY KEY,
   saberID CHAR(30) REFERENCES Saber(id)
      ON DELETE SET NULL
      ON UPDATE CASCADE,
   homeWorld char(30) REFERENCES Planet(name)
);
```

Example

```
DELETE FROM Jedi
WHERE name="Vader"
```

Jedi

name	saberID	homeWorld
Yoda	1828	Degoba
Luke	2333	Tatooine
~~Vader~~	~~1818~~	~~Tatooine~~
Obi-wan	1033	Stewjan

saber

id	color
1828	green
2333	green
~~1818~~ NULL	red
1033	blue

Example

```
UPDATE Jedi
SET saberID=1212 WHERE
name="Vader"
```

Jedi

name	saberID	homeWorld
Yoda	1828	Degoba
Luke	2333	Tatooine
Vader	~~1818~~ 1212	Tatooine
Obi-wan	1033	Stewjan

Saber

id	color
1828	green
2333	green
~~1818~~ 1212	red
1033	blue

Deferred Checking

- Having constraints checked all the time is annoying
  - Must insert into referenced table before inserting into table with foreign key
  - What if there is a cyclic dependency?

Example

```
CREATE TABLE Chicken(
   cID INT PRIMARY KEY,
   eID INT REFERENCES Egg(eID)
);

CREATE TABLE Egg(
   eID INT PRIMARY KEY,
   cID INT REFERENCES Chicken(cID)
);
```

Example

```
CREATE TABLE Chicken(
   cID INT PRIMARY KEY,
   eID INT REFERENCES Egg(eID)
);

CREATE TABLE Egg(
   eID INT PRIMARY KEY,
   cID INT REFERENCES Chicken(cID)
);
```

Example

```
CREATE TABLE Chicken(
   cID INT PRIMARY KEY,
   eID INT REFERENCES Egg(eID)
);

CREATE TABLE Egg(
   eID INT PRIMARY KEY,
   cID INT REFERENCES Chicken(cID)
);
```

Deferred Checking

- Better way: only check after an entire transaction completes
- ***Defer*** constraint checking

Example

```
CREATE TABLE Chicken(
   cID INT PRIMARY KEY,
   eID INT REFERENCES Egg(eID)
   INITIALLY DEFERRED DEFERRABLE
);

CREATE TABLE Egg(
   eID INT PRIMARY KEY,
   cID INT REFERENCES Chicken(cID)
   INITIALLY DEFERRED DEFERRABLE
);
```

Example

```
CREATE TABLE Chicken(
   cID INT PRIMARY KEY,
   eID INT REFERENCES Egg(eID)
   INITIALLY DEFERRED DEFERRABLE
);

CREATE TABLE Egg(
   eID INT PRIMARY KEY,
   cID INT REFERENCES Chicken(cID)
   INITIALLY DEFERRED DEFERRABLE
);
```

```
START TRANSACTION;
INSERT INTO Chicken
VALUES (1,2);
INSERT INTO Egg
VALUES (2,1);
COMMIT;
```

Deferred Checking

- This can be done for any kind of constraint
- What other kinds of constraints?
- Let's learn about more kinds of constraints

Attribute Constraints

- Create constraints for a column of our table
  - Can disallow NULL values
  - Can enforce constrains on values

Example

```
CREATE TABLE Jedi(
   name CHAR(30) PRIMARY KEY,
   saberColor char(30) NOT NULL
);
```

Example

```
CREATE TABLE Jedi(
   name CHAR(30) PRIMARY KEY,
   side char(30)
      CHECK side in ('Light', 'Dark'),
   saberColor char(30)
      CHECK (color IN
         ('Red','Green','Orange',…))
);
```

Tuple Constraints

- Allow us to enforce constraints on rows of our table
- Can check values of multiple attributes together
- Disadvantage: must be checked whenever a tuple is modified

Example

```
CREATE TABLE Jedi(
   name CHAR(30) PRIMARY KEY,
   side char(30),
   saberColor char(30),
   CHECK (side='Light'
   OR saberColor<>'Red')
);
```

FYI

- Constraints can be named and modified
- Constraint deferring and checking can be much more complicated
- If you use constraints a lot, read the documentation

Assertions

- Constraints only checked when values of one table are updated
- If we want to enforce more complex constraints, we need to use ***Assertions***

Example

```
CREATE ASSERTION Balance CHECK
(
  (SELECT COUNT(*) FROM Jedi
     WHERE side='Light')=
  (SELECT COUNT(*) FROM Jedi
     WHERE side='Dark')
);
```

Comparison of Constraints

Constraint type	Where declared	When activated	Guaranteed to hold
Attribute	With attribute	Attribute update	Not with subqueries
Tuple	With table	Row updated	Not with subqueries
Assertion	With database	Any mentioned table updated	Yes

ACID test

- Atomicity ☺
- Consistency ☺
- Isolation ☺
- Durability 😐

Triggers

- Constraints are
  - inefficient to implement
  - not very flexible
- Triggers enable a dynamic response to an event

Triggers

1. Awakened by an *event*
   - Insertions, deletions, updates
   - Transaction COMMITs
2. Test a *condition*
3. Perform an *action*
   - Any SQL statement is allowed

Sometimes called "ECA rules"

Trigger Syntax

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
    {BEFORE|AFTER} {INSERT|DELETE|UPDATE}
        ON <table_name>
    [REFERENCING
        [NEW AS <new_row_name>] [OLD AS <old_row_name>]]
    [FOR EACH ROW [WHEN (<trigger_condition>)]]
    <trigger_body>
```

Trigger Syntax

- COMPLICATED!
- If you want to write your own, read reference material first
- Let's look at examples
  - See why/how they're used
  - Learn a bit of the syntax

Example

```
CREATE TRIGGER NoFallenJedi
AFTER UPDATE OF side ON Jedi
REFERENCING
   OLD ROW AS oldR,
   NEW ROW AS newR
FOR EACH ROW
WHEN oldR.side="Light" AND newR.side="Dark"
   UPDATE Jedi
   SET side=oldR.side
   WHERE name=newR.name
```

Example

```
CREATE TRIGGER NoFallenJedi
AFTER UPDATE OF side ON Jedi
REFERENCING
   OLD ROW AS oldR,
   NEW ROW AS newR
FOR EACH ROW
WHEN oldR.side="Light" AND newR.side="Dark"
   UPDATE Jedi
   SET side=oldR.side
   WHERE name=newR.name
```

Example

```
CREATE TRIGGER ForceBalance
AFTER UPDATE OF side ON Jedi
REFERENCING
    OLD TABLE AS oldT,
    NEW TABLE AS newT
FOR EACH STATEMENT
WHEN (SELECT COUNT(*) FROM Jedi WHERE side='Light')=(SELECT COUNT(*) FROM JEDI WHERE side='Dark')
BEGIN
    DELETE FROM Jedi
    WHERE (name) IN (SELECT name FROM newT);
    INSERT INTO Jedi (SELECT * FROM OldT);
END;
```

Example

```
CREATE TRIGGER DefaultGreenSaber
BEFORE INSERT OF Jedi
REFERENCING
   NEW ROW AS newR,
   NEW TABLE AS newT
FOR EACH ROW
WHEN newR.saberCol IS NULL
   UPDATE newT SET saberCol=“Green”
```