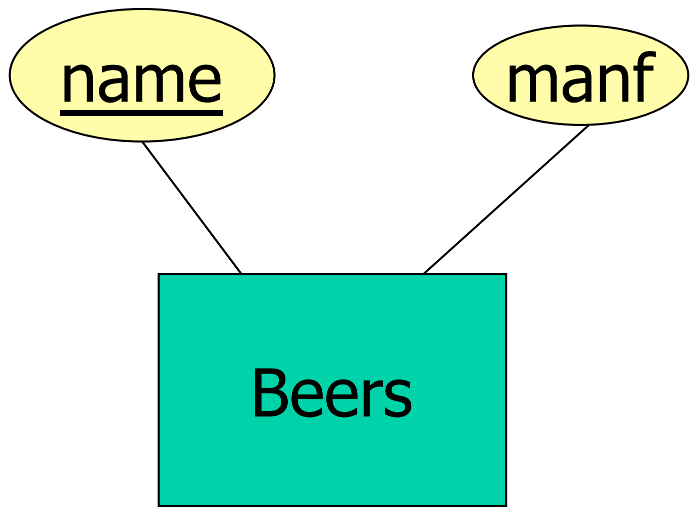
From E/R Diagrams to Relations

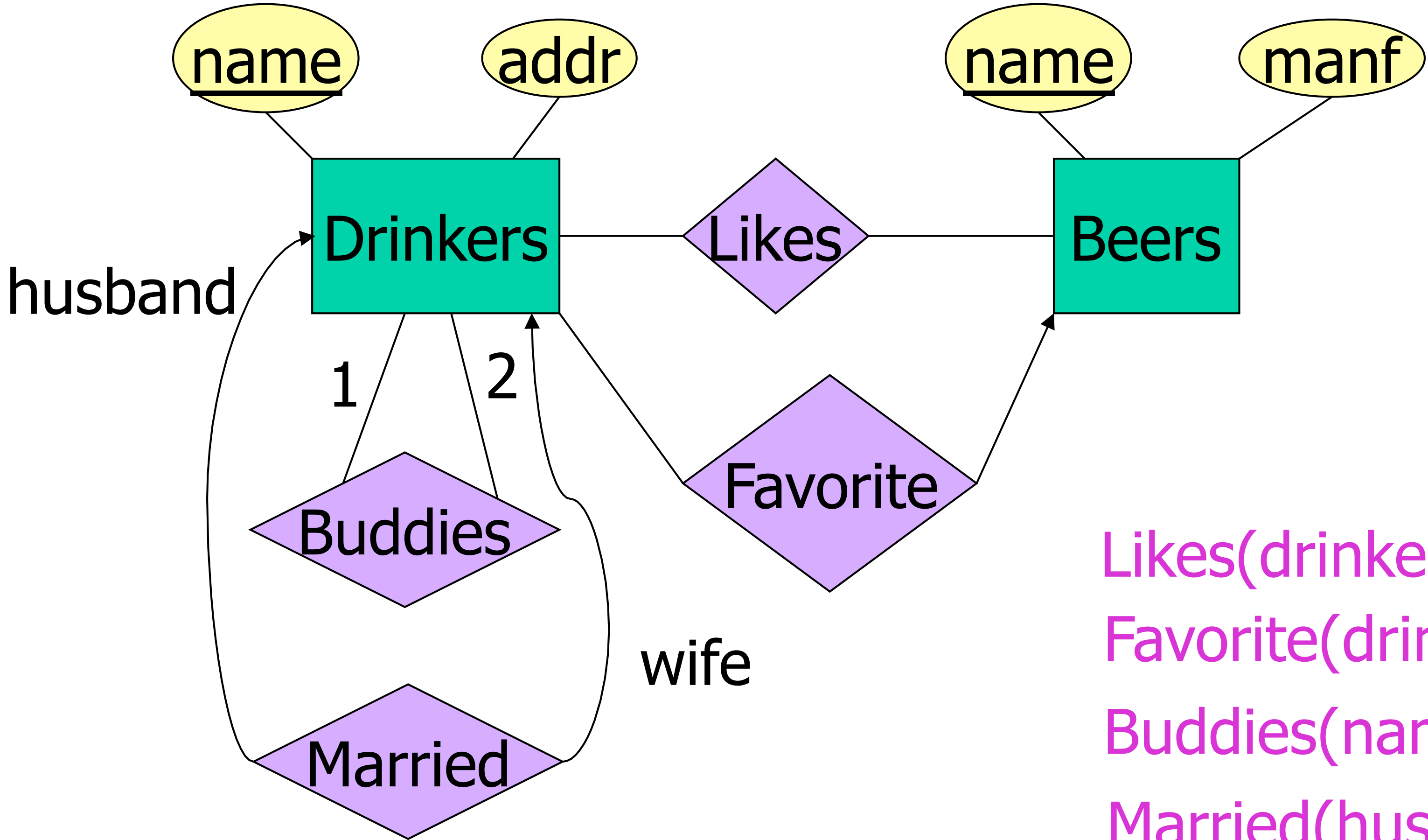
- Entity set → relation
  - Attributes → attributes
- Relationships → relations whose attributes are only:
  - The keys of the connected entity sets
  - Attributes of the relationship itself

Entity Set → Relation



Relation: Beers(name, manf)

Relationship → Relation



Likes(drinker, beer)

Favorite(drinker, beer)

Buddies(name1, name2)

Married(husband, wife)

Combining Relations

- OK to combine into one relation:
  1. The relation for an entity-set *E*
  2. The relations for many-one relationships of which *E* is the "many"
- Example: Drinkers(name, addr) and Favorite(drinker, beer) combine to make Drinker1(name, addr, favBeer)

Risk with Many-Many Relationships

- Combining Drinkers with Likes would be a mistake. It leads to redundancy, as:

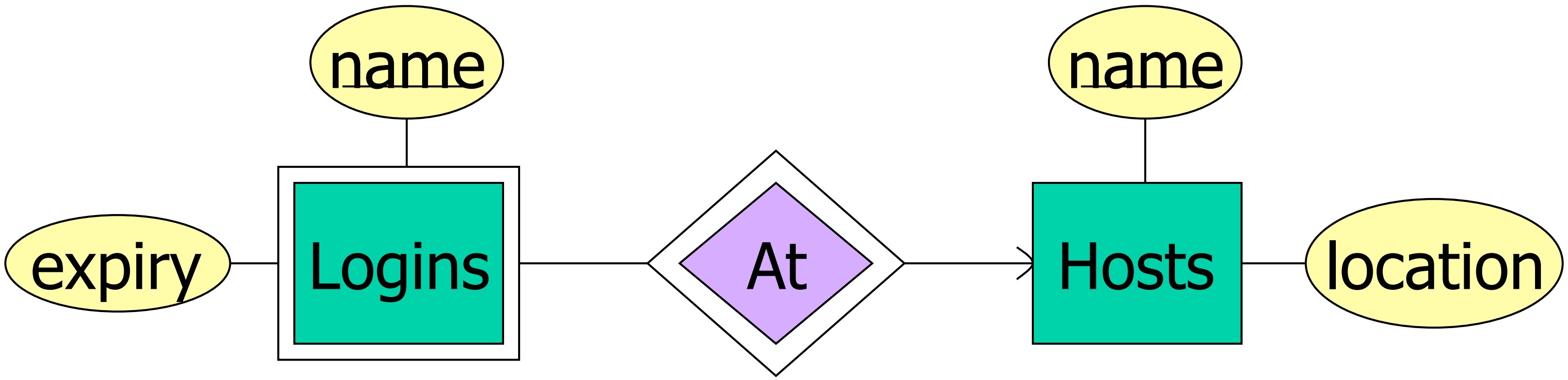
name	addr	beer
Peter	Campusvej	Od.Cl.
Peter	Campusvej	Erd.W.

Redundancy

Handling Weak Entity Sets

- Relation for a weak entity set must include attributes for its complete key (including those belonging to other entity sets), as well as its own, nonkey attributes
- A supporting relationship is redundant and yields no relation (unless *it* has attributes)

Example: Weak Entity Set → Relation



Hosts(hostName, location)

Logins(loginName, hostName, expiry)

~~At(loginName, hostName, hostName2)~~

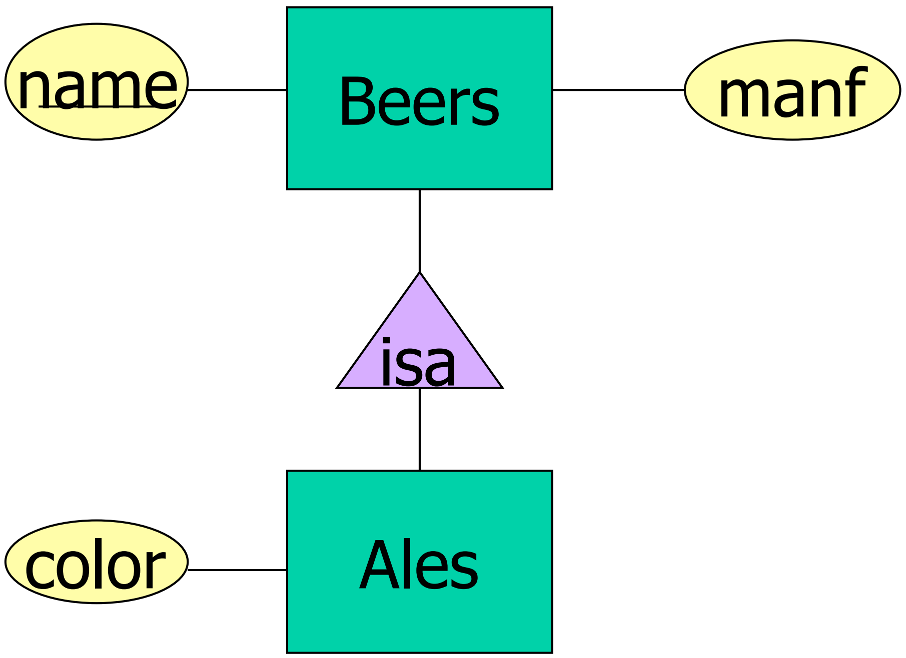
At becomes part of Logins

Must be the same

Subclasses: Three Approaches

1. *Object-oriented* : One relation per subset of subclasses, with all relevant attributes
2. *Use nulls* : One relation; entities have NULL in attributes that don't belong to them
3. *E/R style* : One relation for each subclass:
   - Key attribute(s)
   - Attributes of that subclass

Example: Subclass → Relations



Object-Oriented

name	manf
Odense Classic	Albani

Beers

name	manf	color
HC Andersen	Albani	red

Ales

Good for queries like "find the color of ales made by Albani"

E/R Style

name	manf
Odense Classic	Albani
HC Andersen	Albani

Beers

name	color
HC Andersen	red

Ales

Good for queries like "find all beers (including ales) made by Albani"

Using Nulls

name	manf	color
Odense Classic	Albani	NULL
HC Andersen	Albani	red

Beers

Saves space unless there are *lots* of attributes that are usually NULL

Summary 6

More things you should know:

- Entities, Attributes, Entity Sets,
- Relationships, Multiplicity, Keys
- Roles, Subclasses, Weak Entity Sets
- Design guidelines
- E/R diagrams $\rightarrow$ relational model

The Project

Purpose of the Project

- To try in practice the process of designing and creating a relational database application
- This process includes:
  - development of an E/R model
  - transfer to the relational model
  - normalization of relations
  - implementation in a DBMS
  - programming of an application

Project as (part of) the Exam

- The project is the exam for HA(Dat) students
- Part of the exam for IMADA students
- The project must be done *individually*
- No cooperation is allowed beyond what is explicitly stated in the description

Subject of the Project

- *To create an electronic inventory for a computer store*
- Keep information about complete computer systems and components
- System should be able to
  - calculate prices for components and computer systems
  - make lists of components to order from the distributor

Objects of the System

- component: name, kind, price
  - kind is one of CPU, RAM, graphics card, mainboard, case
  - CPU: socket, bus speed
  - RAM: type, bus speed
  - mainboard: CPU socket, RAM type, on-board graphics?, form factor
  - case: form factor

Objects of the System

- computer system: catchy name, list of components
  - requires a case, a mainboard, a CPU, RAM, optionally a graphics card
  - sockets, bus speed, RAM type, and form factor must match
  - if there is no on-board graphics, a graphics card must be included

Objects of the System

- current stock: list of components and their current amount
- minimum inventory: list of components, their allowed minimum amount, and their preferred amount after restocking

Intended Use of the System

- Print a daily price list for components and computer systems
- Give quotes for custom orders
- Print out a list of components for restocking on Saturday morning (computer store restocks his inventory every Saturday at his distributor)

Selling Price

- Selling price for a component is the price + 30%
- Selling price for a computer system is sum of the selling prices of the components rounded up to next ’99‘
- Rebate System:
  - total price is reduced by 2% for each additional computer system ordered
  - maximal 20% rebate

Example: Selling Price

- computer system for which the components are worth DKK 1984
- the selling price of the components is 1984*1.3 = 2579.2
- It would be sold for DKK 2599
- Order of 3 systems: DKK 7485, i.e., DKK 2495 per system
- Order of 11, 23, or 42 systems: DKK 2079 per system

Functionality of the System

- List of all components in the system and their current amount
- List of all computer systems in the the system and how many of each could be build from the current stock
- Price list including all components and their selling prices grouped by kind all computers systems that could be build from the current stock including their components and selling price

Functionality of the System

- Price offer given the computer system and the quantity
- Sell a component or a computer system by updating the current stock
- Restocking list including names and amounts of all components needed for restocking to the preferred level

Limitations for the Project

- No facilities for updating are required except for the Selling mentioned explicitly
- Only a simple command-line based interface for user interaction is required
  - Choices by the user can be input by showing a numbered list of alternatives or by prompting for component names, etc.
- You are welcome to include update facilities or make a better user interface but *this will not influence the final grade!*

Tasks

1. Develop an appropriate E/R model
2. Transfer to a relational model
3. Ensure that all relations are in 3NF (decompose and refine the E/R model)

---

4. Implement in PostgreSQL DBMS (ensuring the constraints hold)
5. Program in Java or PHP an application for the user interaction providing all the functionality described above

Test Data

- Can be made up as you need it
- At least in the order of 8 computer systems and 30 components
- Sharing data with other participants in the course is explicitly allowed and *encouraged*

Formalities

- Printed report of 10-15 pages
  - design choices and reasoning
  - structure of the final solution
  - Must include:
    - A diagram of your E/R model
    - Schemas of your relations
    - Arguments showing that these are in 3NF
    - Central parts of your SQL code + explanation
    - A (very) short user manual for the application.
  - no documentation of testing is required

Milestones

- There are two stages:
  1. Tasks 1-3, deadline March 6
     Preliminary report describing design choices, E/R model, resulting relational model (will be commented on and handed back)
  2. Tasks 4-5, deadline March 20
     Final report as correction and extension of the preliminary report
- Grade for the project will be based both on the preliminary and on the final report

Implementation IMADA

- Java with JDBC as database interface
- SQL and Java code handed in by “aflever DM505“ command
- Database for testing must be available on the PostgreSQL server
- Testing during grading will use your program and the data on that server

Implementation HA(Dat)

- PHP with web interface
- SQL and PHP code handed in by WebDAV to the PostgreSQL server
- Database for testing must be available on the PostgreSQL server
- Testing during grading will use your website and the data on that server

Constraints

Constraints and Triggers

- A *constraint* is a relationship among data elements that the DBMS is required to enforce
  - Example: key constraints
- *Triggers* are only executed when a specified condition occurs, e.g., insertion of a tuple
  - Easier to implement than complex constraints

Kinds of Constraints

- Keys
- Foreign-key, or referential-integrity
- Value-based constraints
  - Constrain values of a particular attribute
- Tuple-based constraints
  - Relationship among components
- Assertions: any SQL boolean expression

Review: Single-Attribute Keys

- Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute
- Example:

```
CREATE TABLE Beers (
  name   CHAR(20) PRIMARY KEY,
  manf   CHAR(20)
);
```

Review: Multiattribute Key

- The bar and beer together are the key for Sells:

```
CREATE TABLE Sells (
        bar         CHAR(20),
        beer        VARCHAR(20),
        price       REAL,
        PRIMARY KEY (bar, beer)
  );
```

Foreign Keys

- Values appearing in attributes of one relation must appear together in certain attributes of another relation
- Example: in Sells(bar, beer, price), we might expect that a beer value also appears in Beers.name

Expressing Foreign Keys

- Use keyword REFERENCES, either:
  1. After an attribute (for one-attribute keys)
  2. As an element of the schema:

  FOREIGN KEY (<list of attributes>)

  REFERENCES <relation> (<attributes>)
- Referenced attributes must be declared PRIMARY KEY or UNIQUE

Example: With Attribute

```
CREATE TABLE Beers (
  name    CHAR(20) PRIMARY KEY,
  manf    CHAR(20));
CREATE TABLE Sells (
  bar     CHAR(20),
  beer    CHAR(20) REFERENCES Beers(name),
  price   REAL );
```

Example: As Schema Element

```
CREATE TABLE Beers (
  name    CHAR(20) PRIMARY KEY,
  manf    CHAR(20) );
CREATE TABLE Sells (
  bar     CHAR(20),
  beer    CHAR(20),
  price   REAL,
  FOREIGN KEY(beer) REFERENCES
     Beers(name));
```

Enforcing Foreign-Key Constraints

- If there is a foreign-key constraint from relation *R* to relation *S*, two violations are possible:
  1. An insert or update to *R* introduces values not found in *S*
  2. A deletion or update to S causes some tuples of *R* to "dangle"

Actions Taken

- Example: suppose *R* = Sells, *S* = Beers
- An insert or update to Sells that introduces a non-existent beer must be rejected
- A deletion or update to Beers that removes a beer value found in some tuples of Sells can be handled in three ways (next slide)

Actions Taken

1. *Default:* Reject the modification
2. *Cascade:* Make the same changes in Sells
   - Deleted beer: delete Sells tuple
   - Updated beer: change value in Sells
3. *Set NULL:* Change the beer to NULL

Example: Cascade

- Delete the Od.Cl. tuple from Beers:
  - Then delete all tuples from Sells that have beer = 'Od.Cl.'
- Update the Od.Cl. tuple by changing 'Od.Cl.' to 'Odense Classic':
  - Then change all Sells tuples with beer = 'Od.Cl.' to beer = 'Odense Classic'

Example: Set NULL

- Delete the Od.Cl. tuple from Beers:
  - Change all tuples of Sells that have beer = ’Od.Cl.’ to have beer = NULL
- Update the Od.Cl. tuple by changing ’Od.Cl.’ to ’Odense Classic’:
  - Same change as for deletion

Choosing a Policy

- When we declare a foreign key, we may choose policies SET NULL or CASCADE independently for deletions and updates
- Follow the foreign-key declaration by:

ON [UPDATE, DELETE][SET NULL CASCADE]

- Two such clauses may be used
- Otherwise, the default (reject) is used

Example: Setting Policy

```
CREATE TABLE Sells (
  bar    CHAR(20),
  beer   CHAR(20),
  price  REAL,
  FOREIGN KEY(beer)
      REFERENCES Beers(name)
      ON DELETE SET NULL
      ON UPDATE CASCADE
);
```

Attribute-Based Checks

- Constraints on the value of a particular attribute
- Add CHECK(<condition>) to the declaration for the attribute
- The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery

Example: Attribute-Based Check

```
CREATE TABLE Sells (
  bar    CHAR(20),
  beer   CHAR(20)    CHECK (beer IN
           (SELECT name FROM Beers)),
  price  INT CHECK (price <= 100)
);
```

Timing of Checks

- Attribute-based checks are performed only when a value for that attribute is inserted or updated
  - Example: `CHECK (price <= 100)` checks every new price and rejects the modification (for that tuple) if the price is more than 100
  - Example: `CHECK (beer IN (SELECT name FROM Beers))` not checked if a beer is deleted from Beers (unlike foreign-keys)

Tuple-Based Checks

- CHECK (<condition>) may be added as a relation-schema element
- The condition may refer to any attribute of the relation
  - But other attributes or relations require a subquery
- Checked on insert or update only

Example: Tuple-Based Check

- Only Carlsens Kvarter can sell beer for more than 100:

```
CREATE TABLE Sells (
    bar       CHAR(20),
    beer      CHAR(20),
    price     REAL,
    CHECK (bar = 'C4' OR
                price <= 100)
);
```

Assertions

- These are database-schema elements, like relations or views
- Defined by:

```
CREATE ASSERTION <name>
    CHECK (<condition>);
```

- Condition may refer to any relation or attribute in the database schema

Example: Assertion

- In Sells(bar, beer, price), no bar may charge an average of more than 100

```
CREATE ASSERTION NoRipoffBars CHECK (
  NOT EXISTS (
    SELECT bar FROM Sells
    GROUP BY bar
    HAVING 100 < AVG(price)
  ));
```

Bars with an average price above 100

Example: Assertion

- In Drinkers(name, addr, phone) and Bars(name, addr, license), there cannot be more bars than drinkers

```
CREATE ASSERTION LessBars CHECK (
  (SELECT COUNT(*) FROM Bars) <=
  (SELECT COUNT(*) FROM Drinkers)
);
```

Timing of Assertion Checks

- In principle, we must check every assertion after every modification to any relation of the database
- A clever system can observe that only certain changes could cause a given assertion to be violated
  - Example: No change to Beers can affect FewBar; neither can an insertion to Drinkers

Triggers

Triggers: Motivation

- Assertions are powerful, but the DBMS often cannot tell when they need to be checked
- Attribute- and tuple-based checks are checked at known times, but are not powerful
- Triggers let the user decide when to check for any condition

Event-Condition-Action Rules

- Another name for "trigger" is *ECA rule*, or *event-condition-action* rule
- *Event:* typically a type of database modification, e.g., "insert on Sells"
- *Condition:* Any SQL boolean-valued expression
- *Action:* Any SQL statements

Preliminary Example: A Trigger

- Instead of using a foreign-key constraint and rejecting insertions into Sells(bar, beer, price) with unknown beers, a trigger can add that beer to Beers, with a NULL manufacturer

Example: Trigger Definition

```
CREATE TRIGGER BeerTrig
  AFTER INSERT ON Sells
  REFERENCING NEW ROW AS NewTuple
  FOR EACH ROW
  WHEN (NewTuple.beer NOT IN
    (SELECT name FROM Beers))
  INSERT INTO Beers(name)
    VALUES(NewTuple.beer);
```

- AFTER INSERT ON Sells — The event
- WHEN (NewTuple.beer NOT IN (SELECT name FROM Beers)) — The condition
- INSERT INTO Beers(name) VALUES(NewTuple.beer); — The action

Options: CREATE TRIGGER

- CREATE TRIGGER <name>
- or CREATE OR REPLACE TRIGGER <name>
  - Useful if there is a trigger with that name and you want to modify the trigger

Options: The Event

- AFTER can be BEFORE
  - Also, INSTEAD OF, if the relation is a view
    - A clever way to execute view modifications: have triggers translate them to appropriate modifications on the base tables
- INSERT can be DELETE or UPDATE
  - And UPDATE can be UPDATE . . . ON a particular attribute

Options: FOR EACH ROW

- Triggers are either "row-level" or "statement-level"
- FOR EACH ROW indicates row-level; its absence indicates statement-level
- *Row level triggers:* execute once for each modified tuple
- *Statement-level triggers:* execute once for a SQL statement, regardless of how many tuples are modified

Options: REFERENCING

- INSERT statements imply a new tuple (for row-level) or new table (for statement-level)
  - The “table” is the set of inserted tuples
- DELETE implies an old tuple or table
- UPDATE implies both
- Refer to these by

[NEW OLD][TUPLE TABLE] AS <name>

Options: The Condition

- Any boolean-valued condition
- Evaluated on the database as it would exist before or after the triggering event, depending on whether BEFORE or AFTER is used
  - But always before the changes take effect
- Access the new/old tuple/table through the names in the REFERENCING clause

Options: The Action

- There can be more than one SQL statement in the action
  - Surround by BEGIN . . . END if there is more than one
- But queries make no sense in an action, so we are really limited to modifications

Another Example

- Using Sells(bar, beer, price) and a unary relation RipoffBars(bar), maintain a list of bars that raise the price of any beer by more than 10

The Trigger

```
CREATE TRIGGER PriceTrig
  AFTER UPDATE OF price ON Sells
  REFERENCING
    OLD ROW AS ooo
    NEW ROW AS nnn
  FOR EACH ROW
  WHEN (nnn.price > ooo.price + 10)
  INSERT INTO RipoffBars
    VALUES (nnn.bar);
```

The event – only changes to prices

Updates let us talk about old and new tuples

We need to consider each price change

Condition: a raise in price > 10

When the price change is great enough, add the bar to RipoffBars