MSiA-413 Introduction to Databases and Information Retrieval

Lecture 6
Extended ER Diagrams, SELECT Query Steps

Instructor: Nikos Hardavellas

Slides adapted from Steve Tarzia, R. Ramakrishnan and J. Gehrke

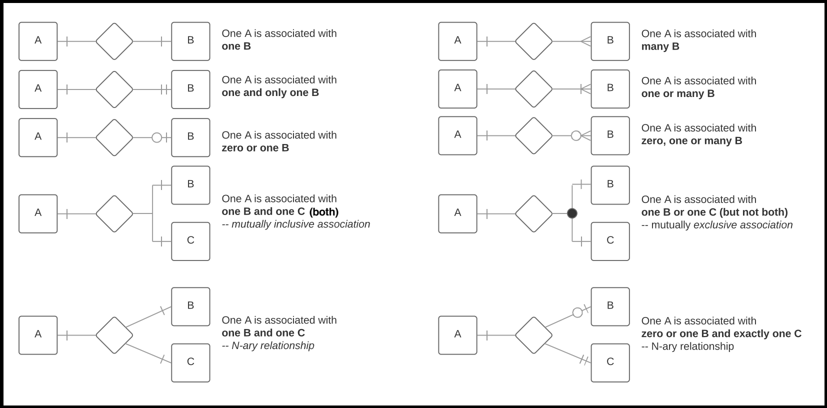
Last Lecture

- Three policies for handling deletion of foreign keys
  - Restrict, Cascade, set NULL
- Table relationships
  - One to Many: most foreign keys
  - One to One: primary key is foreign key. Used for *subset tables*
  - Many to Many: implemented with a *linking table*
- Introduced ER diagrams
  - Cardinality & participation constraints
  - Crow's foot notation
  - Schema vs. instance
- Introduced SQL
  - Syntax diagrams and grammars
  - SELECT queries with filtering, sorting, limiting, arithmetic, and grouping

Overview of Database Design

- Requirements analysis
- *Conceptual design*: (*ER Model* is used at this stage)
  - What are the *entities* and *relationships* (and *events*) in the system we are describing?
    - It is typical to define only entities and relationships; events can be considered as entities
  - What information about these entities and relationships should we store in the database?
    - These are the "*attributes*" of the entity or relationship
  - Which of the attributes are primary *keys*?
    - Uniquely distinguish items that belong to the same entity
    - Primary keys are indicted in ER diagrams by **underlying** them
  - What are the *integrity constraints* or rules of engagement that hold?
    - Cardinality? Participation?
  - A database *schema* in the ER Model can be represented pictorially (*ER diagrams*)
- Logical design
  - Can map an ER diagram into a relational schema (DBMS data model)
- Physical design
  - File types, indexes, disk layout

ER Diagrams cheat sheet



One A is associated with **one B**

One A is associated with **one and only one B**

One A is associated with **zero or one B**

One A is associated with **one B and one C (both)** -- *mutually inclusive association*

One A is associated with **one B and one C** -- *N-ary relationship*

One A is associated with **many B**

One A is associated with **one or many B**

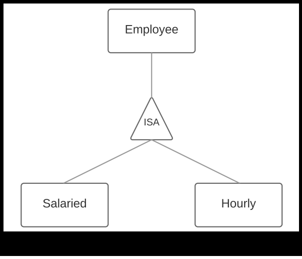
One A is associated with **zero, one or many B**

One A is associated with **one B or one C (but not both)** -- mutually *exclusive association*

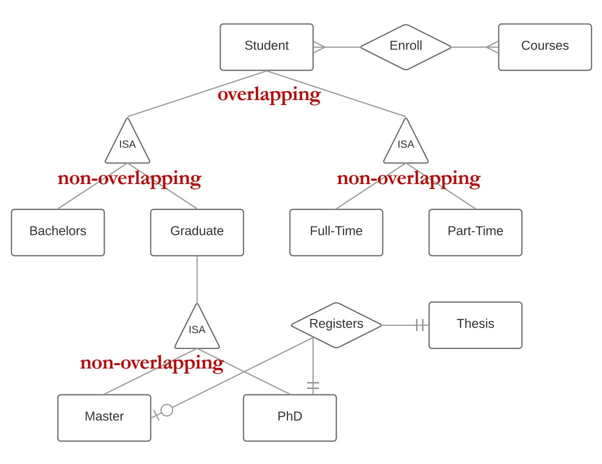
One A is associated with **zero or one B and exactly one C** -- N-ary relationship

Extended ERDs: ISA ("is a") Hierarchies

- As in C++, or other programming languages, attributes are inherited
- If we declare A **ISA** B, every A entity is also considered to be a B entity
- Covering vs. overlapping constraints
  - *Covering constraints*: does every employee have to be either salaried or hourly?
  - *Overlapping constraints*: can Joe be a salaried employee as well as an hourly employee?

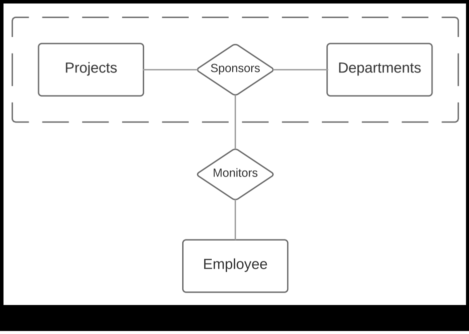


Complex ISA hierarchies



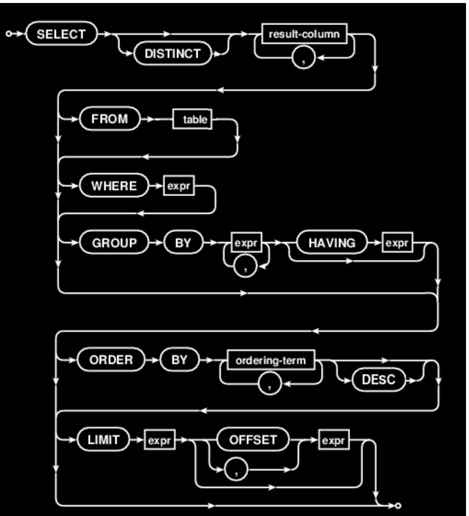
Aggregation

- Used to model a relationship involving another *relationship*
- Allows us to *treat a relationship as an entity* for purposes of participation in (other) relationships



Part 2: SELECT Query Steps

SQLite SELECT Syntax



For example:

```
SELECT FirstName, LastName
FROM customers
WHERE City = "Paris";
```

SELECT queries are series of filtering and manipulation steps

1. The `FROM` expression gives the starting point – a full table
   The final result will be a subset or aggregation of this
2. The `WHERE` expression keeps only those rows passing some test
   This expression can be very complex, but it must be something than can be evaluated on each row, one at a time
3. `GROUP BY` combines rows if something about them is the same
4. The `SELECT` result-columns are computed, including aggregation
   At this point we have thrown out the columns we don’t need
5. `HAVING` expression keeps only the aggregated rows passing a test
6. `ORDER BY` sorts what’s left
7. `LIMIT` truncates the results to just a certain number of rows

SELECT steps (abbreviated)

1. FROM chooses the table of interest
2. WHERE throws out irrelevant rows
3. GROUP BY identifies rows to combine
4. SELECT tells what values to return (allowing math and aggregation)
5. HAVING throws out irrelevant rows (after aggregation)
6. ORDER BY sorts
7. LIMIT throws out rows based on their position in the results

Each step gets closer to the specific result you want

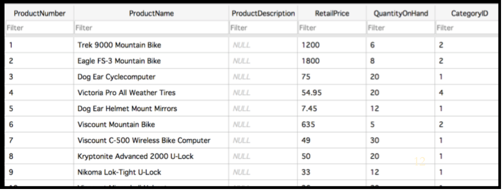
What is the average price of a bike car rack?

1. FROM chooses the table of interest
2. WHERE throws out irrelevant rows
3. GROUP BY identifies rows to combine
4. SELECT tells what values to return (allowing math and aggregation)
5. HAVING throws out irrelevant rows (after aggregation)
6. ORDER BY sorts
7. LIMIT throws out rows based on their position in the results

Products table has the price info, so we start there:

This placeholder will change in step 4

```
SELECT *
  FROM Products;
```



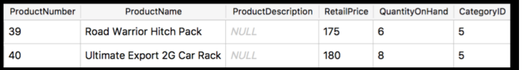
ProductNumber	ProductName	ProductDescription	RetailPrice	QuantityOnHand	CategoryID
Filter	Filter	Filter	Filter	Filter	Filter
1	Trek 9000 Mountain Bike	NULL	1200	6	2
2	Eagle FS-3 Mountain Bike	NULL	1800	8	2
3	Dog Ear Cyclecomputer	NULL	75	20	1
4	Victoria Pro All Weather Tires	NULL	54.95	20	4
5	Dog Ear Helmet Mount Mirrors	NULL	7.45	12	1
6	Viscount Mountain Bike	NULL	635	5	2
7	Viscount C-500 Wireless Bike Computer	NULL	49	30	1
8	Kryptonite Advanced 2000 U-Lock	NULL	50	20	1
9	Nikoma Lok-Tight U-Lock	NULL	33	12	1

What is the average price of a bike car rack?

1. FROM chooses the table of interest
2. WHERE throws out irrelevant rows
3. GROUP BY identifies rows to combine
4. SELECT tells what values to return (allowing math and aggregation)
5. HAVING throws out irrelevant rows (after aggregation)
6. ORDER BY sorts
7. LIMIT throws out rows based on their position in the results

We only need the bike rack products, so we filter on CategoryID = 5

```
SELECT *
   FROM Products
   WHERE CategoryID = 5;
```



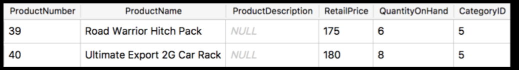
ProductNumber	ProductName	ProductDescription	RetailPrice	QuantityOnHand	CategoryID
39	Road Warrior Hitch Pack	NULL	175	6	5
40	Ultimate Export 2G Car Rack	NULL	180	8	5

What is the average price of a bike car rack?

1. FROM chooses the table of interest
2. WHERE throws out irrelevant rows
3. GROUP BY identifies rows to combine
4. SELECT tells what values to return (allowing math and aggregation)
5. HAVING throws out irrelevant rows (after aggregation)
6. ORDER BY sorts
7. LIMIT throws out rows based on their position in the results

A GROUP BY statement is not needed because we will group *all* of the rows together

```
SELECT *
   FROM Products
   WHERE CategoryID = 5;
```



ProductNumber	ProductName	ProductDescription	RetailPrice	QuantityOnHand	CategoryID
39	Road Warrior Hitch Pack	NULL	175	6	5
40	Ultimate Export 2G Car Rack	NULL	180	8	5

Grouping

The GROUP BY clause combines multiple rows and lets you perform aggregation math functions

```
SELECT AlbumId,
    SUM(Milliseconds/1000/60) AS AlbumMinutes
  FROM tracks GROUP BY AlbumId ORDER BY AlbumMinutes;
```

Result:

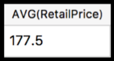
AlbumId	*AlbumMinutes*
340	0.86300000
345	1.11065000
318	1.68821667
…	…

What is the average price of a bike car rack?

1. `FROM` chooses the table of interest
2. `WHERE` throws out irrelevant rows
3. `GROUP BY` identifies rows to combine
4. `SELECT` tells what values to return (allowing math and aggregation)
5. `HAVING` throws out irrelevant rows (after aggregation)
6. `ORDER BY` sorts
7. `LIMIT` throws out rows based on their position in the results

We want the RetailPrice column, and we want to aggregate all the rows with the average function

```
SELECT AVG(RetailPrice)
    FROM Products
    WHERE CategoryID = 5;
```



AVG(RetailPrice)
177.5

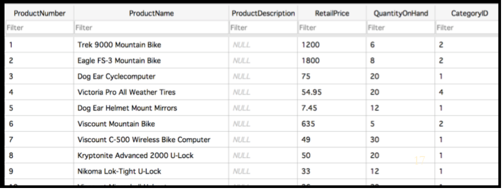
What is the average price of each product type?

1. FROM chooses the table of interest
2. WHERE throws out irrelevant rows
3. GROUP BY identifies rows to combine
4. SELECT tells what values to return (allowing math and aggregation)
5. HAVING throws out irrelevant rows (after aggregation)
6. ORDER BY sorts
7. LIMIT throws out rows based on their position in the results

Products table has the price info, so we start there:

This placeholder will change in step 4

```
SELECT *
   FROM Products;
```



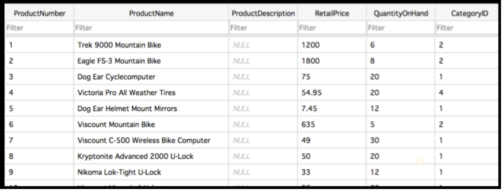
ProductNumber	ProductName	ProductDescription	RetailPrice	QuantityOnHand	CategoryID
Filter	Filter	Filter	Filter	Filter	Filter
1	Trek 9000 Mountain Bike	NULL	1200	6	2
2	Eagle FS-3 Mountain Bike	NULL	1800	8	2
3	Dog Ear Cyclecomputer	NULL	75	20	1
4	Victoria Pro All Weather Tires	NULL	54.95	20	4
5	Dog Ear Helmet Mount Mirrors	NULL	7.45	12	1
6	Viscount Mountain Bike	NULL	635	5	2
7	Viscount C-500 Wireless Bike Computer	NULL	49	30	1
8	Kryptonite Advanced 2000 U-Lock	NULL	50	20	1
9	Nikoma Lok-Tight U-Lock	NULL	33	12	1

What is the average price of each product type?

1. FROM chooses the table of interest
2. WHERE throws out irrelevant rows
3. GROUP BY identifies rows to combine
4. SELECT tells what values to return (allowing math and aggregation)
5. HAVING throws out irrelevant rows (after aggregation)
6. ORDER BY sorts
7. LIMIT throws out rows based on their position in the results

We want all products, so no need to filter

```
SELECT *
   FROM Products;
```



ProductNumber	ProductName	ProductDescription	RetailPrice	QuantityOnHand	CategoryID
Filter	Filter	Filter	Filter	Filter	Filter
1	Trek 9000 Mountain Bike	NULL	1200	6	2
2	Eagle FS-3 Mountain Bike	NULL	1800	8	2
3	Dog Ear Cyclecomputer	NULL	75	20	1
4	Victoria Pro All Weather Tires	NULL	54.95	20	4
5	Dog Ear Helmet Mount Mirrors	NULL	7.45	12	1
6	Viscount Mountain Bike	NULL	635	5	2
7	Viscount C-500 Wireless Bike Computer	NULL	49	30	1
8	Kryptonite Advanced 2000 U-Lock	NULL	50	20	1
9	Nikoma Lok-Tight U-Lock	NULL	33	12	1

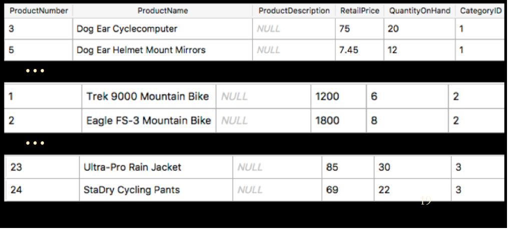
What is the average price of each product type?

1. FROM chooses the table of interest
2. WHERE throws out irrelevant rows
3. GROUP BY identifies rows to combine
4. SELECT tells what values to return (allowing math and aggregation)
5. HAVING throws out irrelevant rows (after aggregation)
6. ORDER BY sorts
7. LIMIT throws out rows based on their position in the results

The GROUP BY statement groups together all rows of the same product category

Note: syntax below only for illustration

```
SELECT *
   FROM Products
   GROUP BY CategoryID;
```



ProductNumber	ProductName	ProductDescription	RetailPrice	QuantityOnHand	CategoryID
3	Dog Ear Cyclecomputer	NULL	75	20	1
5	Dog Ear Helmet Mount Mirrors	NULL	7.45	12	1

…

1	Trek 9000 Mountain Bike	NULL	1200	6	2
2	Eagle FS-3 Mountain Bike	NULL	1800	8	2

…

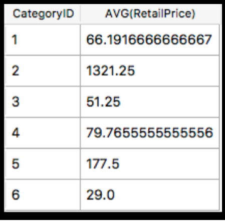
23	Ultra-Pro Rain Jacket	NULL	85	30	3
24	StaDry Cycling Pants	NULL	69	22	3

What is the average price of each product type?

1. FROM chooses the table of interest
2. WHERE throws out irrelevant rows
3. GROUP BY identifies rows to combine
4. SELECT tells what values to return (allowing math and aggregation)
5. HAVING throws out irrelevant rows (after aggregation)
6. ORDER BY sorts
7. LIMIT throws out rows based on their position in the results

We want the CategoryID and RetailPrice columns, and we want to aggregate the rows in each group with the average function

```
SELECT CategoryID,
             AVG(RetailPrice)
   FROM Products
   GROUP BY CategoryID;
```



CategoryID	AVG(RetailPrice)
1	66.1916666666667
2	1321.25
3	51.25
4	79.7655555555556
5	177.5
6	29.0