H3

The University of Manchester

The Relational Languages

Fundamentals of Databases
Alvaro A A Fernandes, SCS, UoM

[COMP23111 2014-2015 Lecture 03 of 12]

Acknowledgements

- These slides are adaptations (mostly minor, but some major) of material authored and made available to instructors by **A. Silberschatz, H. F. Korth, and S. Sudarshan** to accompany their textbook **Database System Concepts, 6th Edition, McGraw-Hill, 2006, 978-0-07-352332-3** and **H. Garcia-Molina, J.D. Ullman, and J. Widom** to accompany their textbook **Database Systems: The Complete Book, 2nd Edition, Pearson Prentice Hall, 2009, 978-0-13-135428-9**
- Copyright remains with them, whom I thank.
- All errors are my responsibility.

In Previous Lectures

- We learned that data is an enterprise asset and that DBMSs are crucial to manage it well.
- We learned the importance of adopting different levels of abstraction in designing and implementing databases.
- We learned how data models lead to schemas and instances that enable a logical view of the data.
- We learned about the main components of DBMSs and the various architectures used to deploy them.
- We started learning about the relational approach to data modelling.
- We started learning SQL, focussing on its DDL and DML capabilities.

In This Lecture

- What is an algebra?
- What is a relational algebra?
- What are the basic operations of relational algebra?
- What are the most important derived operations?
- What are the most important operations to have in an extended relational algebra?
- How do we relate SQL to relational algebra?
- What are the main constructs and query types in SQL?

The University of Manchester

What is an algebra?

What is an Algebra?

- An algebra is a mathematical structure consisting of:
  - Operands: variables or values, typically drawn from a single sort (i.e., one type only), from which new values can be constructed
  - Operators: symbols denoting procedures that construct new values from given values and with closure
- Equivalences between expressions follow from operator semantics (e.g., commutativity, associativity)
- By closure we mean that evaluating any well-formed expression yields an element of the same type as the inputs, i.e., from the set of operands

A Familiar Example

- Take as operands, the real numbers and an infinite set of letters used to denote them.
- Take as operators: addition (+), subtraction (-), multiplication (*), division (/), all binary.
- We can construct some example expressions.
- Addition and multiplication are commutative and associative.
- Closure holds.

```
2+2

4*(3/2)

(4*3)/2

x+1

2/(4*x)

x+y = y+x

(x*y)*z = x*(y*z)
```

Relational Algebras

- A relational algebra (RA) is an algebra whose operands are relation names (i.e., variables that denote relation instances).

- Relational-algebraic operators are designed to do the most common things that we need to do with relations in a database.

- They can take parameters (e.g., a predicate, or a list of attribute names).

- The result is an algebra that can be used as a query language over relational databases.

result is a **new** relation
(with new name, schema and instance)

$$\textbf{JaneEyreInstructors} = \pi_{\text{teaches.i\_n}}(\sigma_{\text{takes.s\_n='Jane Eyre'}}(\textbf{teaches} \bowtie_{\text{course}} \textbf{takes}))$$

parameters

operands are relation names

operators

Relational Algebras

- When evaluated against a database instance, a relational-algebraic expression yields a relation that is the result of the query.
- The evaluation is grounded on the relation instances denoted by variables in the expression.
- One special characteristic is that, because attributes and relations have names (i.e., there is the notion of a relation schema), we need to:
  - Take into account the schemas of the input relations, and
  - Infer (and perhaps explicitly act to specify) the schema of the output relation.

result is a **new** relation
(with new name, schema and instance)

$$\textbf{JaneEyreInstructors} = \pi_{\text{teaches.i\_n}}(\sigma_{\text{takes.s\_n='Jane Eyre'}}(\text{teaches} \bowtie_{\text{course}} \text{takes}))$$

parameters

operands are relation names

operators

Some Terminology

- A relation R with schema $R = \{a_1{:}t_1, \ldots, a_n{:}t_n\}$ has arity (or degree) n
- An instance of R is a set of tuples conforming to the schema of R
- By conformance to a schema we mean that
  - every tuple must have the schema-defined arity (i.e., number of columns) and
  - the value of each schema-defined attribute a_i in every tuple must have the schema-defined type t_i
- The cardinality (of a given instance of R) is the number of tuples in the instance, which we denote by $|R|$

Core Relational Algebra

Core Relational Algebra

- Some relational-algebraic operators are primitive and some are derivable.
- Consider, by analogy, addition and multiplication.
- One can derive multiplication from addition, whereas addition is not derivable from the usual arithmetic operators.
- Addition is, in some sense, primitive, whereas multiplication is not.
- We define a core RA by taking a set of primitive operators only
- Then, we specify the derivation of additional ones from our choice of primitives.

Core Relational Algebra: Union, (Intersection) and Difference

- Because relations are sets of tuples, the binary set-theoretic operators **Union** and **Difference** are part of the core RA.

- **Intersection** is derivable from union and difference: A∩B = ((A∪B)\(A\B))\(B\A) .

- In the case of RA, for set-theoretic operations to be well-formed, both operands must have compatible relation schemas.

- The schemas of two relations *R* and *R'* are *compatible* if they have the same arity and for every attribute of type t_i in *R* there is a corresponding attribute of type t_i in *R'* .

2	s	5	b	y
3	z	6	n	r
7	z	4	b	y
8	a	3	n	t

1	a	4	v	t
2	s	5	b	y
4	s	4	c	t
5	s	7	v	r
6	a	5	c	r
7	z	4	b	y

Union

Difference

1	a	4	v	t
2	s	5	b	y
3	z	6	n	r
4	s	4	c	t
5	s	7	v	r
6	a	5	c	r
7	z	4	b	y
8	a	3	n	t

3	z	6	n	r
8	a	3	n	t

What about Intersection?

Core Relational Algebra: Union, (Intersection) and Difference

- Notice that these operations have the potential to yield a result with different cardinality than either operand, but the arity of the result does not differ from that of the operands.

- The cardinality may be

  - larger (in the case of **Union**)

  - smaller (in the case of **Intersection** and **Difference**)

2	s	5	b	y
3	z	6	n	r
7	z	4	b	y
8	a	3	n	t

A(R) = 5
|R| = 4

1	a	4	v	t
2	s	5	b	y
4	s	4	b	y
5	s	7	v	r
6	a	5	c	r
7	z	4	b	y

A(S) = 5
|S| = 6

Union

Difference

1	a	4	v	t
2	s	5	b	y
3	z	6	n	r
4	s	4	b	y
5	s	7	v	r
6	a	5	c	r
7	z	4	b	y
8	a	3	n	t

A(RuS) = 5
|RuS| = 8

3	z	6	n	r
8	a	3	n	t

A(R-S) = 5
|R-S| = 2

Core Relational Algebra: (Cartesian) Product

- The binary set-theoretic (**Cartesian**) **product** operator is also in the core RA.
- It concatenates every tuple of one operand with every tuple of the other to produce tuples in the result.
- It creates compositions/ associations of relations.
- It normally yields a result with larger cardinality and larger arity than either operand.

2	s	5	b	y
3	z	6	n	r
7	z	4	b	y
8	a	3	n	t

A(R) = 5
|R| = 4

3	z	6	n	r
8	a	3	n	t

A(S) = 5
|S| = 2

Product

2	s	5	b	y	3	z	6	n	r
3	z	6	n	r	3	z	6	n	r
7	z	4	b	y	3	z	6	n	r
8	a	3	n	t	3	z	6	n	r
2	s	5	b	y	8	a	3	n	t
3	z	6	n	r	8	a	3	n	t
7	z	4	b	y	8	a	3	n	t
8	a	3	n	t	8	a	3	n	t

A(RxS) = 10
|RxS| = 8

Core Relational Algebra: Selection

- **Selection** is a unary operator that picks only certain rows from its operand.
- It has the potential to yield a result with smaller cardinality than either operand, but the arity of the result does not differ.

A	B	C	D	E
1	a	4	v	t
2	s	5	b	y
3	z	6	n	r
4	s	4	b	y
5	s	7	v	r
6	a	5	c	r
7	z	4	b	y
8	a	3	n	t

A(R) = 5
|R| = 8

S = Selection[B≠'s'](R)

A	B	C	D	E
1	a	4	v	t
3	z	6	n	r
6	a	5	c	r
7	z	4	b	y
8	a	3	n	t

A(S) = 5
|S| = 5

Core Relational Algebra: Projection

- **Projection** is an operator that picks only certain columns from its operand.
- It has the potential to yield a result with smaller arity than either operand, and may also make the cardinality smaller (why?)

A	B	C	D	E
1	a	4	v	t
2	s	5	b	y
3	z	6	n	r
4	s	4	b	y
5	s	7	v	r
6	a	5	c	r
7	z	4	b	y
8	a	3	n	t

A(R) = 5
|R| = 8

S = Projection[B,D,E](R)

B	D	E
a	v	t
s	b	y
z	n	r
s	v	r
a	c	r
z	b	y
a	n	t

A(S) = 3
|S| = 7

Core Relational Algebra: Renaming

- **Renaming** is a unary operator that determines the name that one or more attributes in the operand will have the result relation.

- It changes neither the arity nor the cardinality of the result relation.

- Its use is a formal necessity and it is often convenient in practice.

A	B	C	D	E
1	a	4	v	t
2	s	5	b	y
3	z	6	n	r
4	s	4	b	y
5	s	7	v	r
6	a	5	c	r
7	z	4	b	y
8	a	3	n	t

A(R) = 5
|R| = 8

S = Renaming[B→X,E→Y](R)

A	X	C	D	Y
1	a	4	v	t
2	s	5	b	y
3	z	6	n	r
4	s	4	b	y
5	s	7	v	r
6	a	5	c	r
7	z	4	b	y
8	a	3	n	t

A(S) = 5
|S| = 8

The University of Manchester

Extending the Core Relational Algebra

Derivable RA Operators

- Besides **Intersection**, the most useful derivable operators are:
  - **Joins**, in many different forms
  - **Division**
- The core RA extended with a set of useful derived operators is referred to as an extended RA.

The University of Manchester

RA Notation and Examples

Selection: Notation and Example

$$S := \sigma_c(R)$$

- *c* is a condition (as in 'if' statements) that refers to attributes of *R*.

- *S* contains all those tuples of *R* that satisfy *c*.

Selection: Notation and Example

Sells		
bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

$$\texttt{JoeMenu:=}\sigma_{\texttt{bar='Joe''s'}}\texttt{(Sells)}$$

JoeMenu		
bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75

Projection: Notation and Example

$S := \pi_L(R)$

- *L* is a list of attributes from the schema of *R*.
- *S* is constructed by
  - looking at each tuple of *R*,
  - extracting the attributes on list *L*, in the order specified, and
  - creating from those components a new tuple for *S* if it is not a duplicate of a tuple already placed in *S*.

Projection: Notation and Example

Sells		
bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

$\text{BeerPrices}:=\pi_{\text{beer,price}}(\text{Sells})$

BeerPrices	
beer	price
Bud	2.50
Miller	2.75
Miller	3.00

(Cartesian) Product: Notation and Example

$R_3 \; := \; R_1 \times R_2$

- Pair each tuple *t* of R_1 with each tuple *t'* of R_2.
- Then, the concatenation *t+t'* is a tuple of R_3.
- The schema of R_3 is the concatenation of the attribute declarations of R_1 and R_2.
- But beware, if an attribute *A* exists in R_1 and R_2, use $R_1.A$ and $R_2.A$ to disambiguate
- If projecting out, use renaming.

(Cartesian) Product: Notation and Example

R_1	
A	B
1	2
3	4

$$R_3 := R_1 \times R_2$$

R_2	
C	D
5	6
7	8
9	10

R_3			
A	B	C	D
1	2	5	6
1	2	7	8
1	2	9	10
3	4	5	6
3	4	7	8
3	4	9	10

Renaming: Notation and Example

$$R_2 := \rho_{R2(A1,\ \ldots,\ An)}(R_1)$$

- Assigns to the resulting relation the name R_2 with attributes named $A_1,\ldots,A_n$ and the same tuples as R_1.

- A simplified notation is

$$R_2(A_1,\ \ldots,\ A_n) := R_1$$

Renaming: Notation and Example

Bars	
name	addr
Joe's	Maple St.
Sue's	River Rd.

$$Barz := \rho_{Bars(bar,addr)}\ (Bars)$$

Barz	
bar	addr
Joe's	Maple St.
Sue's	River Rd.

θ-Join: Notation and Example

$$R_3 \ := \ R_1 \ \bowtie_\theta \ R_2$$

- Equivalent to taking the product

$$R \ := \ R_1 \ \times \ R_2$$

then applying a selection

$$R_3 \ := \ \sigma_\theta(R)$$

where θ can be any Boolean-valued condition

- If the only predicate that occurs in θ is '=', we refer to the θ-join as an equijoin.

θ-Join: Notation and Example

Sells		
bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00

Bars	
name	addr
Joe's	Maple St.
Sue's	River Rd.

$$\texttt{BarInfo := Sells} \bowtie_{\texttt{Sells.bar=Bars.name}} \texttt{Bars}$$

BarInfo				
bar	beer	price	name	addr
Joe's	Bud	2.50	Joe's	Maple St.
Joe's	Miller	2.75	Joe's	Maple St.
Sue's	Bud	2.50	Sue's	River Rd.
Sue's	Coors	3.00	Sue's	River Rd.

(Natural) Join: Notation and Example

$R_3 := R_1 \bowtie R_2$

- A useful join variant that connects two relations by:
  - Checking for equality of attributes of the same name
  - Projecting out one copy only of each pair of equated attributes.
- So, equivalent to an equijoin to which we apply a projection that eliminates duplicate columns.

(Natural) Join: Notation and Example

Sells		
bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00

Barz	
bar	addr
Joe's	Maple St.
Sue's	River Rd.

$$\text{BarInfo} := \text{Sells} \bowtie \text{Barz}$$

BarInfo			
bar	beer	price	addr
Joe's	Bud	2.50	Maple St.
Joe's	Miller	2.75	Maple St.
Sue's	Bud	2.50	River Rd.
Sue's	Coors	3.00	River Rd.

Extended Projection: Notation and Example

- Using the same projection operator, we allow the list *L* to contain arbitrary expressions involving attributes:
  - Deriving attributes values, e.g., using arithmetic with name assignment, e.g.,

$$A+B*C \rightarrow D$$

  - Introducing duplicate occurrences of the same attribute (typically with name disambiguation)

Extended Projection: Notation and Example

R	
A	B
1	2
3	4
5	6
7	8

$$S := \pi_{A+B \rightarrow C, A, A \rightarrow A2}(R)$$

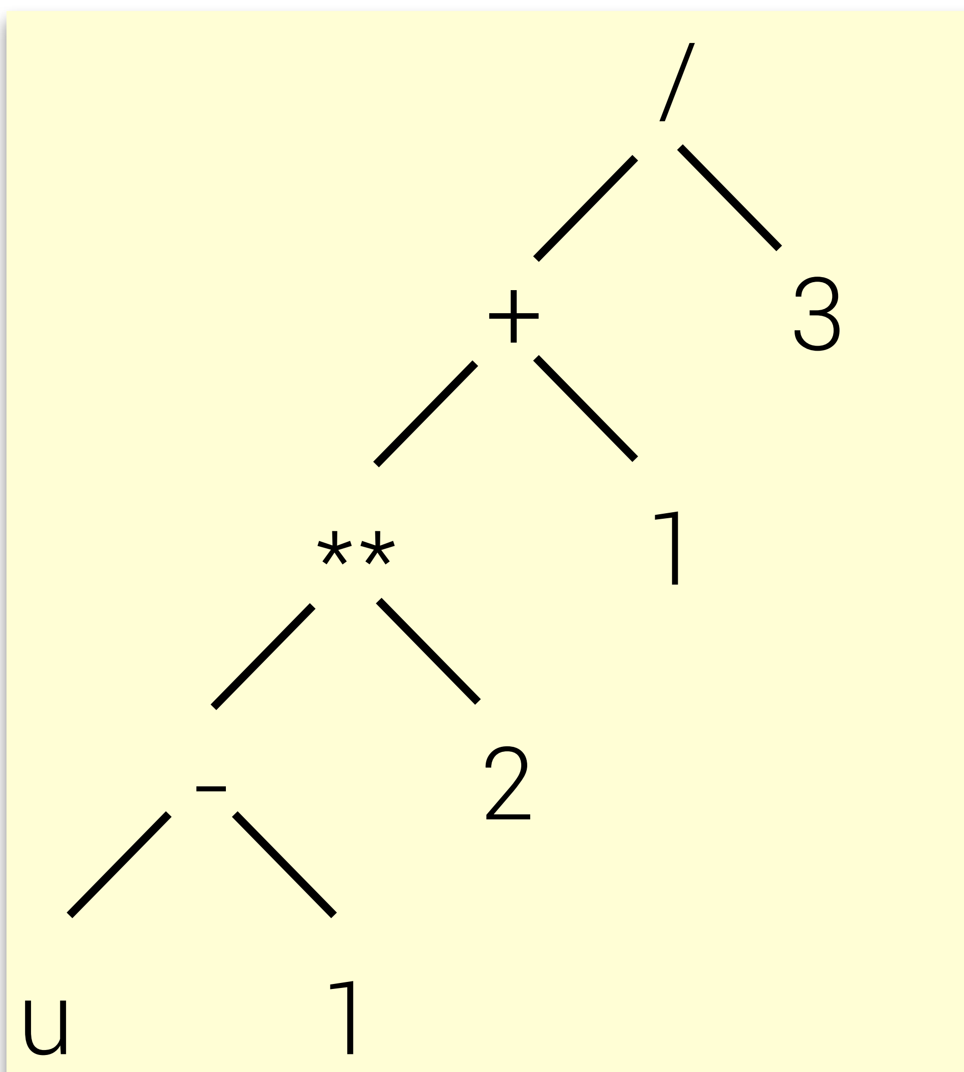
S		
C	A	A2
3	1	1
7	3	3
11	5	5
15	7	7

Building Complex RA Expressions

Building Complex RA Expressions

$v := u-1$
$x := v^2$
$y := x+1$
$z := y/3$

$z = ((u-1)^2+1)/3$



- By combining operators with parentheses and precedence rules, we can build arbitrarily complex expressions.
- Three notations, just as in arithmetic:
  - Sequences of assignment statements with (typically simple) expressions in the right-hand side.
  - Single (typically complex) expressions that are equivalent to entire sequences of assignments.
  - Operator (or expression) trees where nodes are operators or values (variables or literals) and edges denote operator-operand relationships.

As Sequences of Assignment Statements

$$R_1 \bowtie_C R_2$$

$$\equiv$$

$$S := R_1 \times R_2$$

$$R_3 := \sigma_C(S)$$

- Creates temporary relation names for later reference
- Renaming can be implied by giving left-hand side relations a list of attributes.
- For example, the derivation of θ-join from product and selection can be expressed as a sequence of assignments.

As Single Expressions

$$R_1 \bowtie_C R_2$$

$$\equiv$$

$$R_3 := \sigma_C(R_1 \times R_2)$$

- Observe precedence and use parentheses, as follows (from highest to lowest):

  $[\sigma, \pi, \rho]$

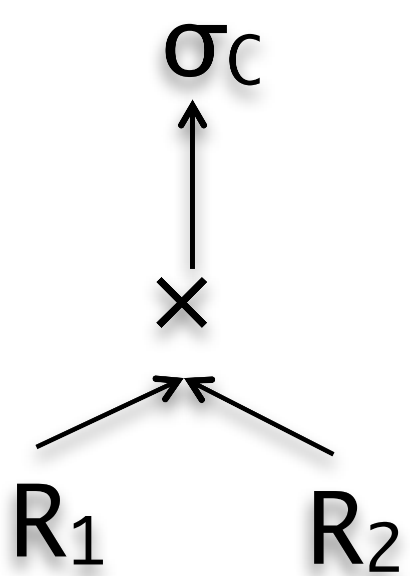
  $[\times, \bowtie]$

  $\cap$

  $[\cup, \setminus]$

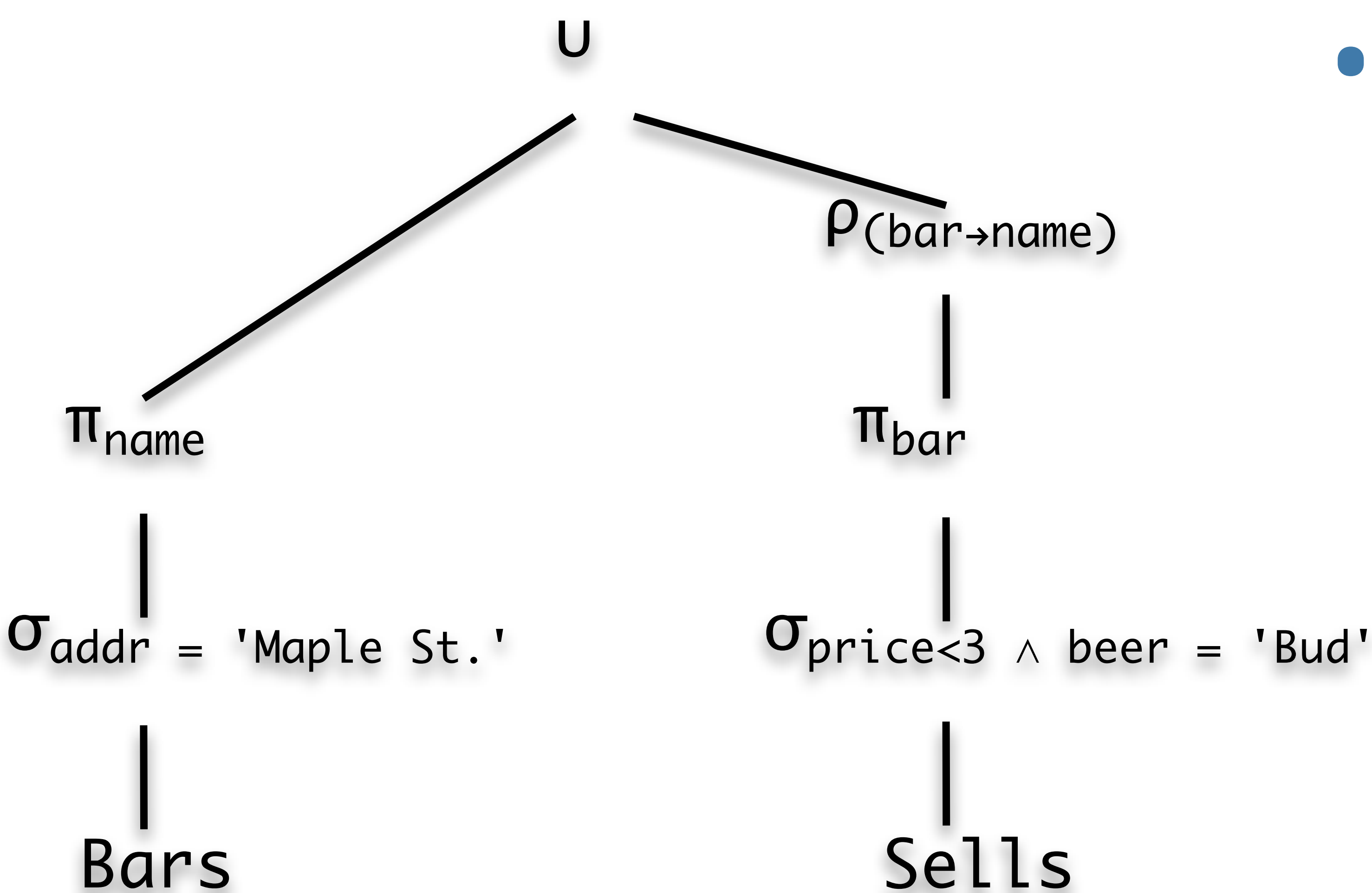
- Set difference is also denoted by $-$
- Renaming may need to be explicit

As Operator Trees



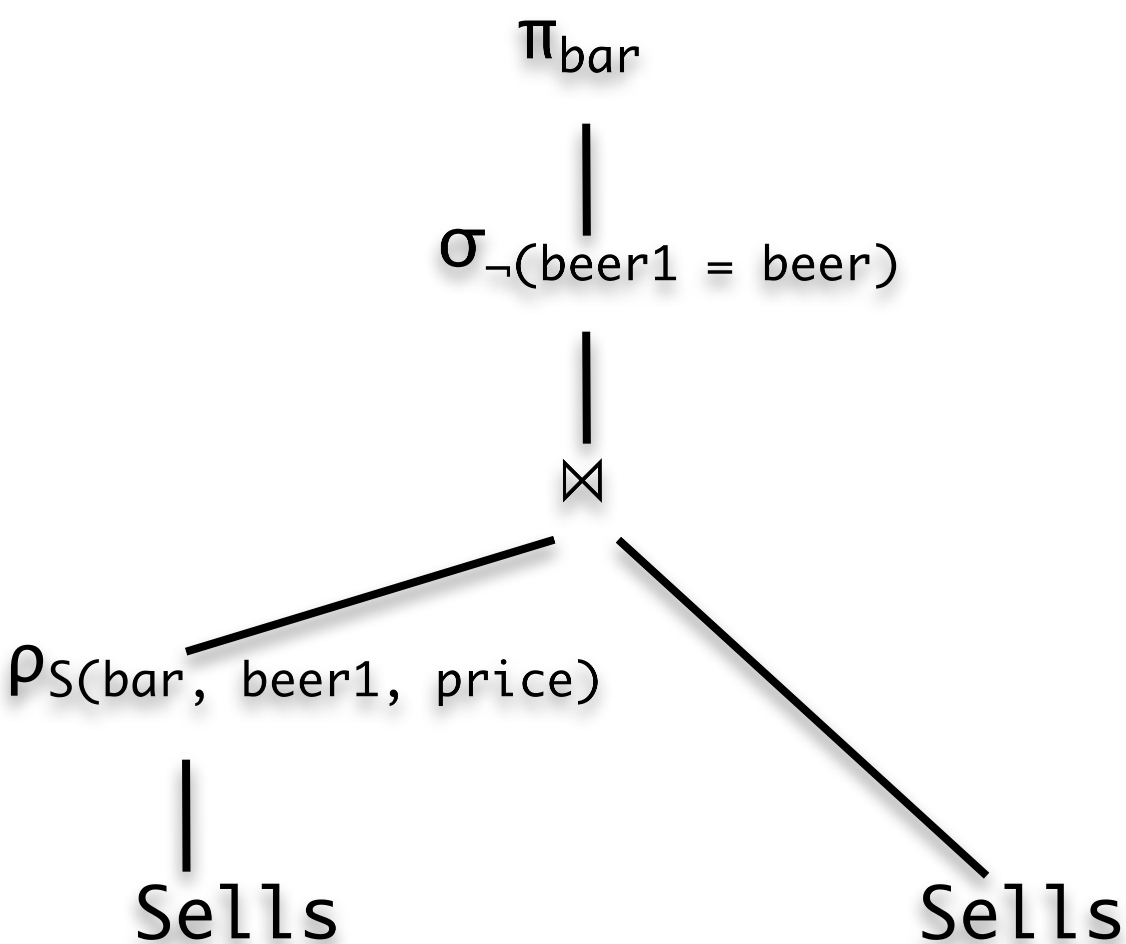
- Leaves are operands: either variables standing for relations or particular, constant relations.
- Interior nodes are operators, applied to their child or children.

More Example Operator Trees



- Using the relations **`Bars(name,addr)`** and **`Sells(bar,beer,price)`**, find the names of all the bars that are either on Maple St. or sell Bud for less than $3.

More Example Operator Trees



- Using **`Sells(bar,beer,price)`**, find the bars that sell two different beers at the same price.
- Here is one strategy:
  - By renaming, define a copy of **`Sells`**, called **`S(bar, beer1, price)`**.
  - Then, the natural join of **`Sells`** and **S** consists of quadruples **`(bar,beer,beer1,price)`** such that the bar sells these two beers at this price.
  - Finally, we pick only pairs of beers that are different.

RA Type Inference

Inferring the Schema of the Result Relation

- **Union**, **Intersection**, **Difference**
  - The schemas of the two operands must be the same, so it is also the schema of the result
- **Selection**
  - the schema of the result is the same as the schema of the single operand
- **Projection**
  - the list of attributes from the single operand tells us the (presumably different) schema of the result
- **Renaming**
  - The operator is designed to define the result schema.

Inferring the Schema of the Result Relation

- **Product**
  - The result schema is the list union of the attributes of the two relations.
  - Use *R.A*, etc., to distinguish two attributes named *A*.
- **Theta-join**
  - The same as for product
- **Natural join**
  - The result schema is the set union of the attributes of the two relations.

RA on Bags

Relational Algebra on Bags

$\{\{1,2,1,3\}\} \neq \{\{1,2,3\}\}$

$\{\{1,2,1,3\}\} = \{\{1,2,3,1\}\}$

$\{\{1,2,3\}\} = \{1,2,3\}$

- Classical relational algebra defines relations as sets.
- But, it is often implemented (e.g., as the basis for SQL) with relations as bags.
- A bag (or multiset) is like a set, but
  - may contain duplicates, i.e., any element may appear more than once.
  - order remains immaterial.
- Sometimes written with double curly brackets, e.g., {{1,2,1,3}}.

Why Bags?

- SQL, the most important query language for relational databases, is actually a bag-based language.
- Some operations, like projection, are more efficient to implement over bags than over sets.

Why Bags?

- This is because duplicate elimination is an expensive operation to compute.
- This, in turn, is because one needs to process all the input tuples before one can produce any result tuple.
- Operations for which this is true (such as projection on sets) are referred to as blocking operations.

Operations on Bags

- Selection applies to each tuple, so its effect on bags is like its effect on sets, but duplicates are kept.
- Projection also applies to each tuple, but as a bag operator, we do not need to eliminate duplicates.
- Products and joins are done on each pair of tuples, so duplicates in bags have no effect on how we operate.

Selection on Bags

R	
A	B
1	2
5	6
1	2

$\sigma_{A+B<5}(R)$

<result>	
A	B
1	2
1	2

Projection on Bags

R	
A	B
1	2
5	6
1	2

$\pi_A (R)$

<result>
A
1
5
1

Product on Bags

R	
A	B
1	2
5	6
1	2

S	
B	C
3	4
7	8

$R \times S$

<result>			
A	R.B	S.B	C
1	2	3	4
1	2	7	8
5	6	3	4
5	6	7	8
1	2	3	4
1	2	7	8

θ-Join on Bags

R	
A	B
1	2
5	6
1	2

S	
B	C
3	4
7	8

$R \bowtie_{R.B<S.B} S$

<result>			
A	R.B	S.B	C
1	2	3	4
1	2	7	8
5	6	7	8
1	2	3	4
1	2	7	8

Union, Intersection and Difference on Bags

- An element appears in the union of two bags the sum of the number of times it appears in each bag.

  {{1,2,1}} ∪ {{1,1,2,3,1}} = {{1,1,1,1,1,2,2,3}}

- An element appears in the intersection of two bags the minimum of the number of times it appears in either.

  {{1,2,1,1}} ∩ {{1,2,1,3}} = {1,1,2}

- An element appears in the difference of two bags A and B as many times as it appears in A, minus the number of times it appears in B.

  {{1,2,1,1}} – {{1,2,3}} = {1,1}

Some Laws that Hold for Sets *Do* Hold for Bags

- Some, but not all algebraic laws that hold for sets also hold for bags.

- The commutative law for union

  $$R \cup S = S \cup R$$

  does hold for bags.

- Since addition is commutative, adding the number of times that a tuple appears in *R* and *S* does not depend on the order of *R* and *S*.

Some Laws that Hold for Sets *Do Not* Hold for Bags

- Set union is idempotent, meaning that, for sets:

$$S \cup S = S$$

- However, for bags, if x appears n times in S, then it appears $2n$ times in the result of bag union

- Thus, in general, for bags:

$$S \cup S \neq S$$

- For example

$$\{\{1\}\} \cup \{\{1\}\} = \{\{1,1\}\} \neq \{\{1\}\}.$$

A Powerful Extended RA

Extended Relational Algebras

- The relational algebra admits of various extensions.
- The main kinds of extended operations are:
  - duplicate elimination
  - sorting
  - several versions of join
  - aggregation and grouping
  - division

Extended RA: Notation

- δ: duplicate elimination
- τ: sorting
- γ: aggregation/grouping
- ⟗, ⟕, ⟖: outer join, left outer join, right outer join
- ⋉, ⋊: left semijoin, right semijoin
- ▻: antijoin
- ÷: division
- Note that the notations for extended RA operators is much less agreed upon than for core RA operators.

Extended RA: Duplicate Elimination

$R_2 := \delta(R_1)$

- R_2 consists of one copy only of each tuple that appears in R_1.
- Note that this may be computationally expensive to implement.

R_1	
A	B
1	2
5	6
1	2

$R_2 := \delta(R_1)$

R_2	
A	B
1	2
5	6

Extended RA: Sorting

$R_2 := \tau_L(R_1)$

- where $L = (a_1,d_1), \ldots, (a_n,d_n)$ is a list of attribute/direction pairs such that each a_i is an attribute in R_1 and each d_i is either 'A' (for ascending order) or 'D' (for descending order).

- R_2 is the list of tuples of R_1 sorted first on the value of the first attribute in L, then on the second attribute in L, and so on, with ties broken arbitrarily

- τ is the only operator whose result is more constrained than a set.

R1	
X	Y
1	2
5	6
1	5

$S := \tau_{(X,A),(Y,D)}\ (R)$

R2	
X	Y
1	5
1	2
5	6

Extended RA: Aggregation

- Aggregation functions, strictly, are not operations in the relational algebra: their operands are not relations.
- Rather, they apply to (entire) columns (i.e., sets, bags, or lists) of a table and produce a single result.
- They map a bulk/collection value into a scalar value.
- The most commonly-used aggregation functions are:
  - `COUNT`
  - `SUM`
  - `AVG`
  - `MIN`
  - `MAX`

R	
A	B
1	3
3	4
3	2

```
COUNT(R.A) = 3
  SUM(R.A) = 7
  AVG(R.B) = 3
  MIN(R.A) = 1
  MAX(R.B) = 4
```

Extended RA: Grouping

$R_2 := \gamma_L(R_1)$

- where *L* is a list of elements that are:
  - either individual (so called, grouping) attributes
  - or the application of an aggregation function on an attribute *A* of R_1, with (re)naming, as in extended projection.

Extended RA: Grouping

- To obtain the result, we rearrange the input tuples according to all the grouping attributes on the list *L*.
- That is, we form one group for each distinct list of values in the input tuples, taking into account the grouping attributes in *L*.
- Over each such group, we compute the aggregation function for each such function in *L*.
- The result has one tuple for each group and has columns as follows:
  - One for each grouping attribute in *L*
  - One for each aggregation function in *L*

Extended RA: Grouping

R		
A	B	C
1	2	3
4	5	6
1	2	5

→

<tmp>		
A	B	C
1	2	3
1	2	5
4	5	6

↓

$S := \gamma_{A,B,AVG(C)\rightarrow X}\ (R)$

S		
A	B	X
1	2	4
4	5	6

Extended RA: Outer Join, Left and Right

- When we refer to join, without qualification, we mean, more precisely, an inner join.

- If we join $R \bowtie_C S$, a tuple of R that has no tuple of S with which it joins is said to be dangling and does not contribute to the result.

- Similarly for a tuple of S that has no tuple of R with which it joins.

Extended RA: Outer Join, Left and Right

- A join that emits such dangling tuples into the result relation by padding them with NULL values is called an outer join.

- If a join condition is given, we write $R ⟗_\theta S$; otherwise, we assume a natural outer join; both of these are referred to as full outer joins.

- An outer join in which only the tuples from the left (resp., right) operand are padded is called a left (resp., right) outer join, and we write $R ⟕_\theta S$ (only pad R tuples) and $R ⟖_\theta S$ (only pad S tuples), resp..

R	
A	B
1	2
4	5

S	
B	C
2	3
6	7

$$T := R ⟗ S$$

T		
A	B	C
1	2	3
4	5	NULL
NULL	6	7

Extended RA: Left/Right Semijoin

- A left semijoin is defined as:

  $$R \ltimes S \equiv \pi_L(R \bowtie S)$$

  where L is the list of attributes in R

- A right semijoin is defined as:

  $$R \rtimes S \equiv \pi_{L'}(R \bowtie S)$$

  where L' is the list of attributes in S

- So a left (resp., right) semijoin is a join that only retains in the result relation the columns of the left (resp., right) operand.

R	
A	B
1	2
5	6
1	2

S	
B	C
2	4
6	8

$T := R \ltimes S$

$U := R \rtimes S$

T	
A	B
1	2
5	6

U	
B	C
2	4

Extended RA: Antijoin

- An antijoin is the complement of a left semijoin, i.e., it is defined as:

$$R \rhd S \equiv R \setminus (R \ltimes S)$$

R	
A	B
1	2
5	6
3	4

S	
B	C
2	4
6	8

$$T := R \rhd S$$

T	
A	B
3	4

Extended RA: Division

- Let *R* have two attributes *R.x* and *R.y*, and *S* one attribute *S.y* with the same domain as *R.y*.

- The division of *R* by *S* is the set of all *R.x* values (as unary tuples) such that for every *S.y* value there is a tuple *(R.x, R.y)* in *R*.

$$T := R \div S$$

$$\equiv$$

$$\pi_L(R) \setminus (\pi_L((\pi_L(R) \times S) \setminus R))$$

$$\equiv$$

$$\begin{aligned} X_1 &:= \pi_L(R) \times S \\ X_2 &:= X_1 \setminus R \\ X_3 &:= \pi_L(X_2) \\ T &:= \pi_L(R) \setminus X_3 \end{aligned}$$

where *L* is the list of attribute names unique to *R*

Extended RA: Division

R	
A	B
F	X
F	Y
F	C
E	X
E	C
S	X
S	Y

S
B
X
Y

$$T := R \div S$$

X_1	
A	B
F	X
F	Y
E	X
E	Y
S	X
S	Y

X_2	
A	B
E	Y

X_3
A
E

T
A
F
S

The University of Manchester

Translating SQL onto Relational Algebra

Direct Translation of Core SQL to RA

- There is is a useful way of computing in your mind what the result of a SQL query would be.
- It is a high-level procedure for obtaining a direct translation from SQL to RA.
- The basic SQL query blocks map to RA as follows:
  - Form the (cascaded binary) product of the relations in the **FROM** clause
  - To this result, apply a selection operator whose predicate is the one in the **WHERE** clause
  - To this result, apply a projection whose attribute list is that in the **SELECT** clause
  - If **DISTINCT** is requested, apply a duplicate removal operator

R

A	B
1	2
5	6
1	2

S

B	C
3	4
7	8

<result>

A	C
1	4
1	8
5	8

```
SELECT
DISTINCT R.A, S.C
FROM     R,S
WHERE    R.B < S.B
```

$\equiv$

$\delta(\pi_{R.A,S.C}(\sigma_{R.B<S.B}(R \times S)))$

SQL with Grouping and Aggregation

- With grouping and aggregation, we extend the previous semantics as follows:
  - Before applying the projection, tuples are grouped by *LG*, where *LG* is the list of grouping attributes in the **`GROUP BY`** clause
  - When the projection is applied, the aggregation functions are applied per group, so that one tuple is generated for each group

R	
A	B
1	2
5	6
1	2

S	
B	C
3	4
7	8

<result>	
A	SUM_C
1	12
5	8

```
SELECT    R.A, SUM(S.C) AS SUM_C
FROM      R,S
WHERE     R.B < S.B
GROUP BY  R.A
```

≡

$\gamma_{R.A, SUM(S.C) \rightarrow SUM\_C}(\sigma_{R.B<S.B}(R \times S))$

Importance of SQL to RA Translation

- Note that, while SQL has a declarative core, RA is wholly procedural.
- The order in which operations are applied is strictly specified by the structure of the relational-algebraic expression (i.e., parentheses and operator precedence).
- The ability to translate SQL to relational algebra is, therefore, a bridge from a declarative to a procedural language.
- Since RA is an algebra, one can use properties such as idempotence, commutativity, associativity as well as more specific equivalence laws to transform an expression E into another expression E'.
- The result of evaluating E' is the same as the result of evaluating E, but E' could be more efficient to compute.

Importance of SQL to RA Translation

- Consider calculating by hand

  1982736458+
  1982736458+
  1982736458+
  1982736458+
  1982736458+
  1982736458

  compared with

  6*1982736458

- One of the goals of an optimizer is to apply such laws to the direct translation of an SQL query to an RA expression and search (by rewriting) for a more efficient, equivalent RA expression.

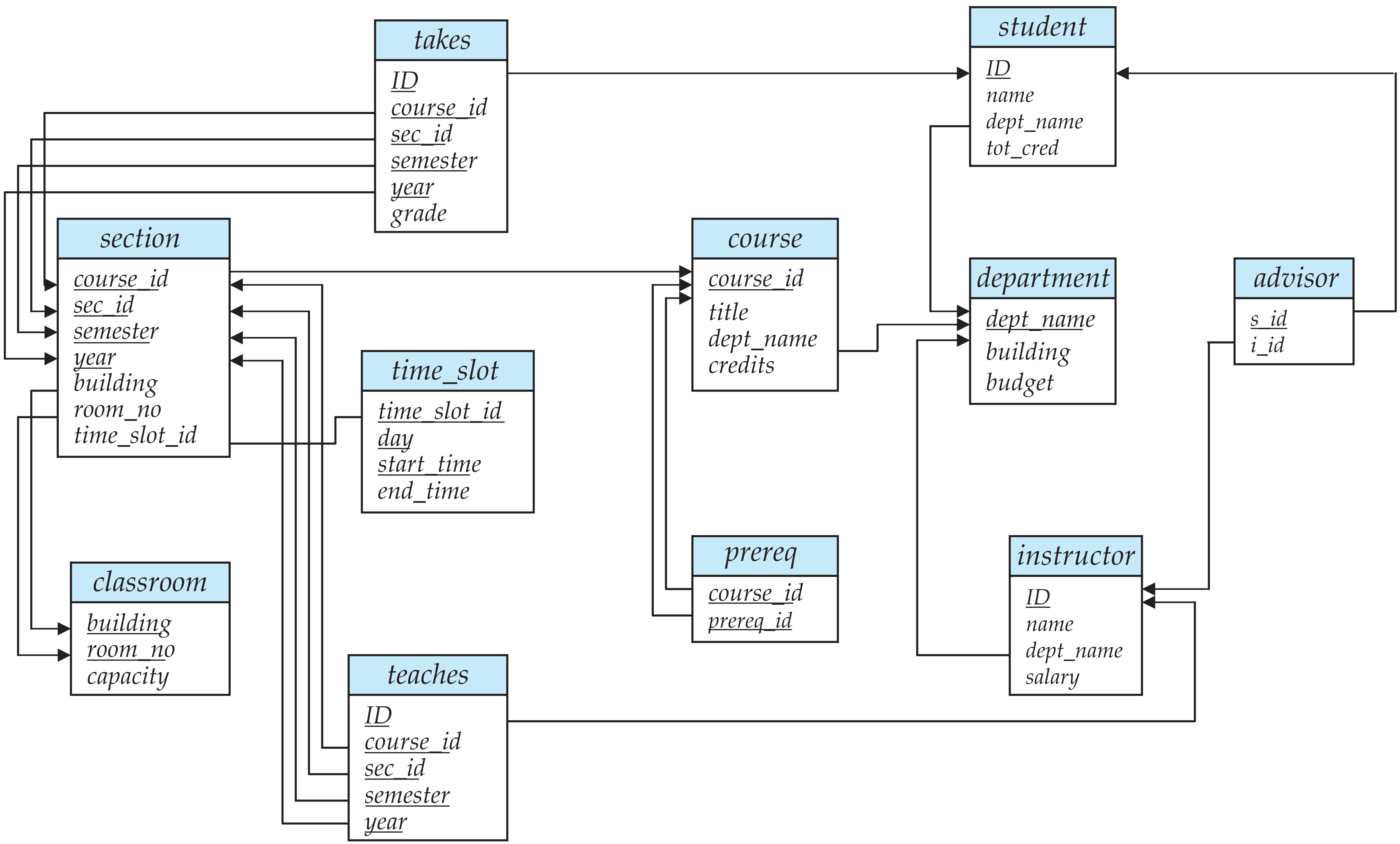
- This search is often guided by the insight that we should reduce the size of a relation (in bytes, i.e., the average width of a tuple times the average number of tuples) of intermediate results as early as possible.

SQL as a Query Language

The History of SQL

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory in the 1970s
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86, SQL-89, SQL-92
  - SQL:1999, SQL:2003, SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Some examples here may not work on some DBMSs.

An Example Database



Some (Not All) Relation Schemas


```
create table department (
       dept_name   varchar(20),
       building    varchar(15),
       budget      numeric(12,2)
                   check (budget > 0),
       primary key (dept_name)
);

create table course (
       course_id   varchar(8),
       title       varchar(50),
       dept_name   varchar(20),
       credits     numeric(2,0)
                   check (credits > 0),
       primary key (course_id),
       foreign key (dept_name)
              references department
              on delete set null
);

create table instructor (
       ID          varchar(5),
       name        varchar(20) not null,
       dept_name   varchar(20),
       salary      numeric(8,2)
                   check (salary > 29000),
       primary key (ID),
       foreign key (dept_name)
              references department
              on delete set null
);
```

```
create table section (
       course_id    varchar(8),
       sec_id       varchar(8),
       semester     varchar(6)
                    check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
       year         numeric(4,0)
                    check (year > 1701 and year < 2100),
       building     varchar(15),
       room_number  varchar(7),
       time_slot_id varchar(4),
       primary key (course_id, sec_id, semester, year),
       foreign key (course_id)
                   references course
                   on delete cascade,
       foreign key (building, room_number)
                   references classroom
                   on delete set null
);

create table takes (
       ID           varchar(5),
       course_id    varchar(8),
       sec_id       varchar(8),
       semester     varchar(6),
       year         numeric(4,0),
       grade        varchar(2),
       primary key (ID, course_id, sec_id, semester, year),
       foreign key (course_id,sec_id, semester, year)
                   references section
                   on delete cascade,
       foreign key (ID)
                   references student
                   on delete cascade
);
```

Some (Not All) Relation Instances

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

course

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

department

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

prereq

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

section

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

The University of Manchester

Core SQL

SQL: Basic Structure

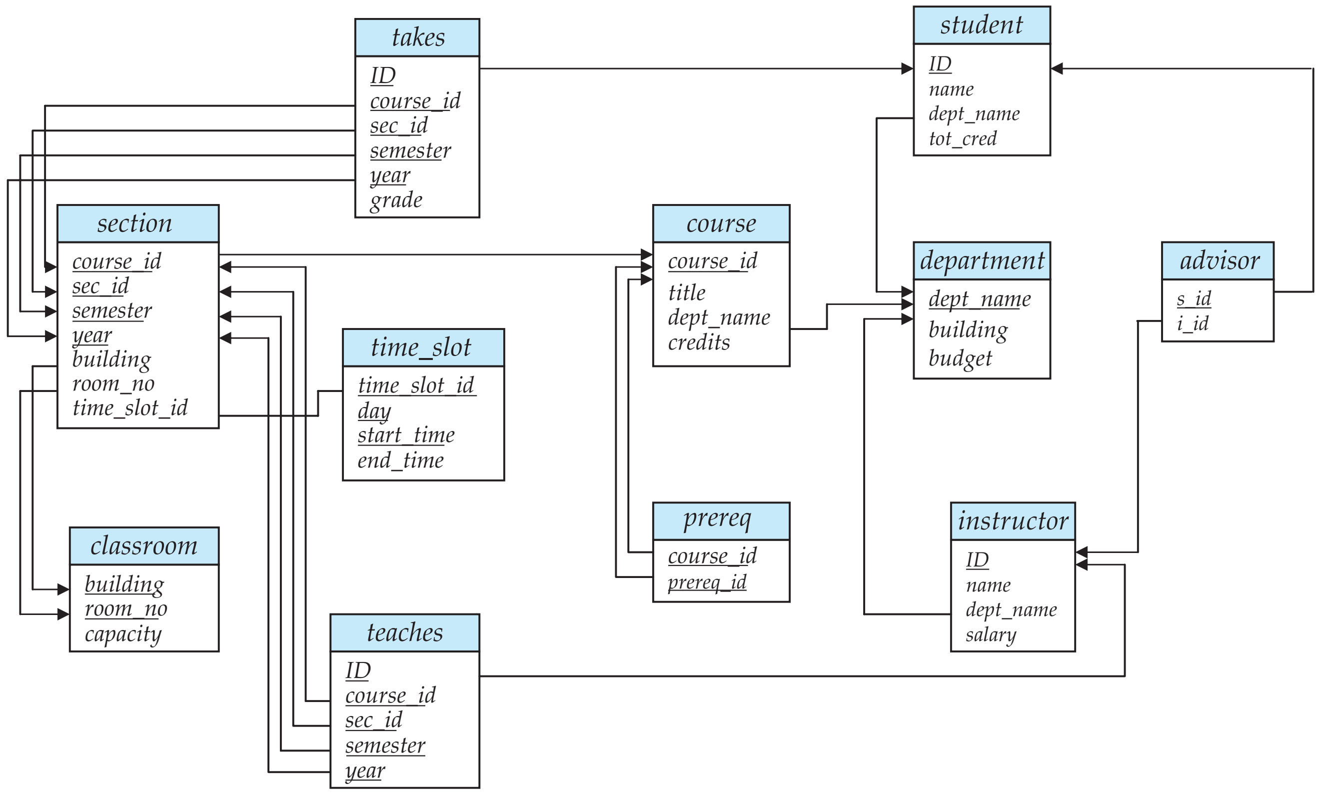
- A typical SQL query has the form:

```
SELECT A1, A2, ..., An
FROM   r1, r2, ..., rm
WHERE  P;
```

  - A_i denotes an attribute
  - R_i denotes a relation
  - P is a predicate.

- The result of an SQL query is a relation.

SQL: The SELECT Clause

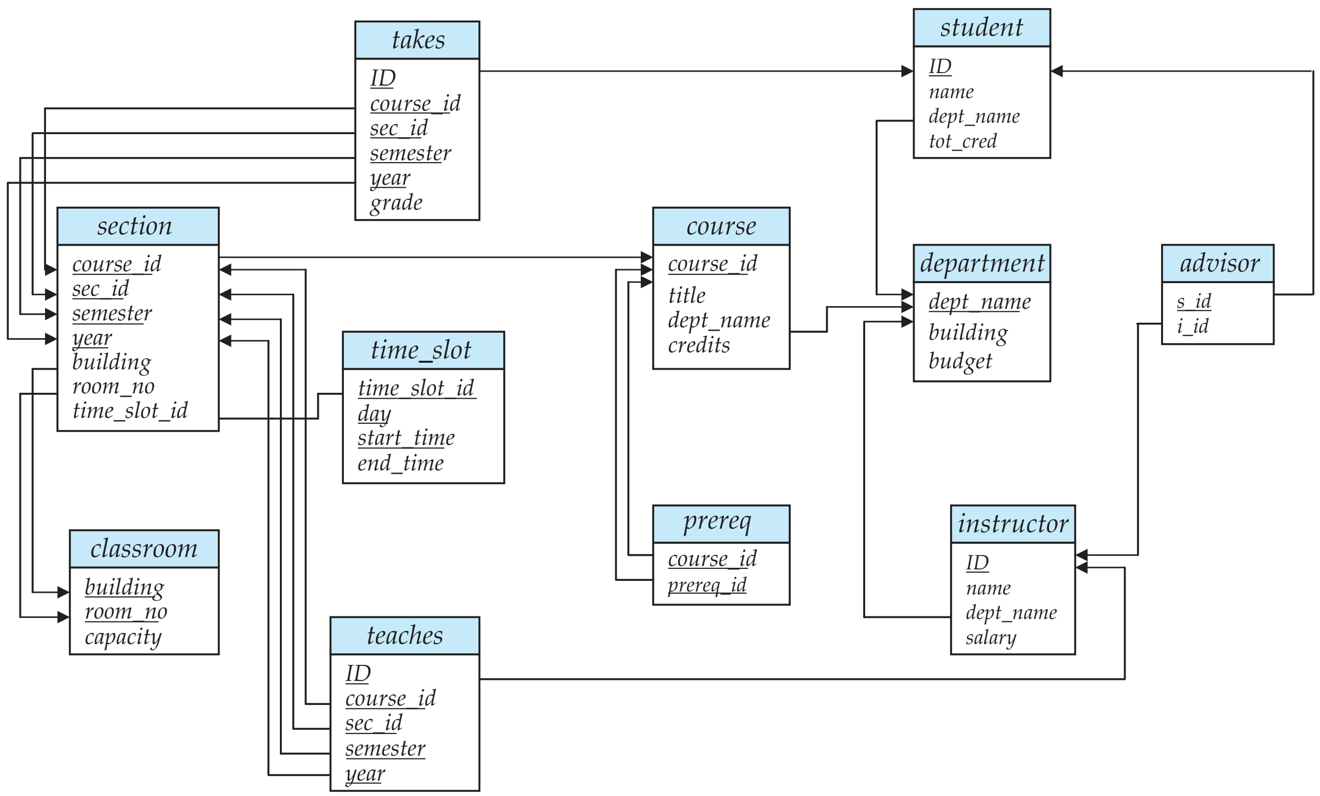


- The SELECT clause lists the attributes desired in the result of a query.
- It corresponds to the projection operation of the relational algebra.
- For example, to find the names of all instructors:

```
SELECT name
FROM   instructor;
```

- SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. Name ≡ NAME ≡ name
- Some people use bold font wherever we use upper case.

SQL: The SELECT Clause



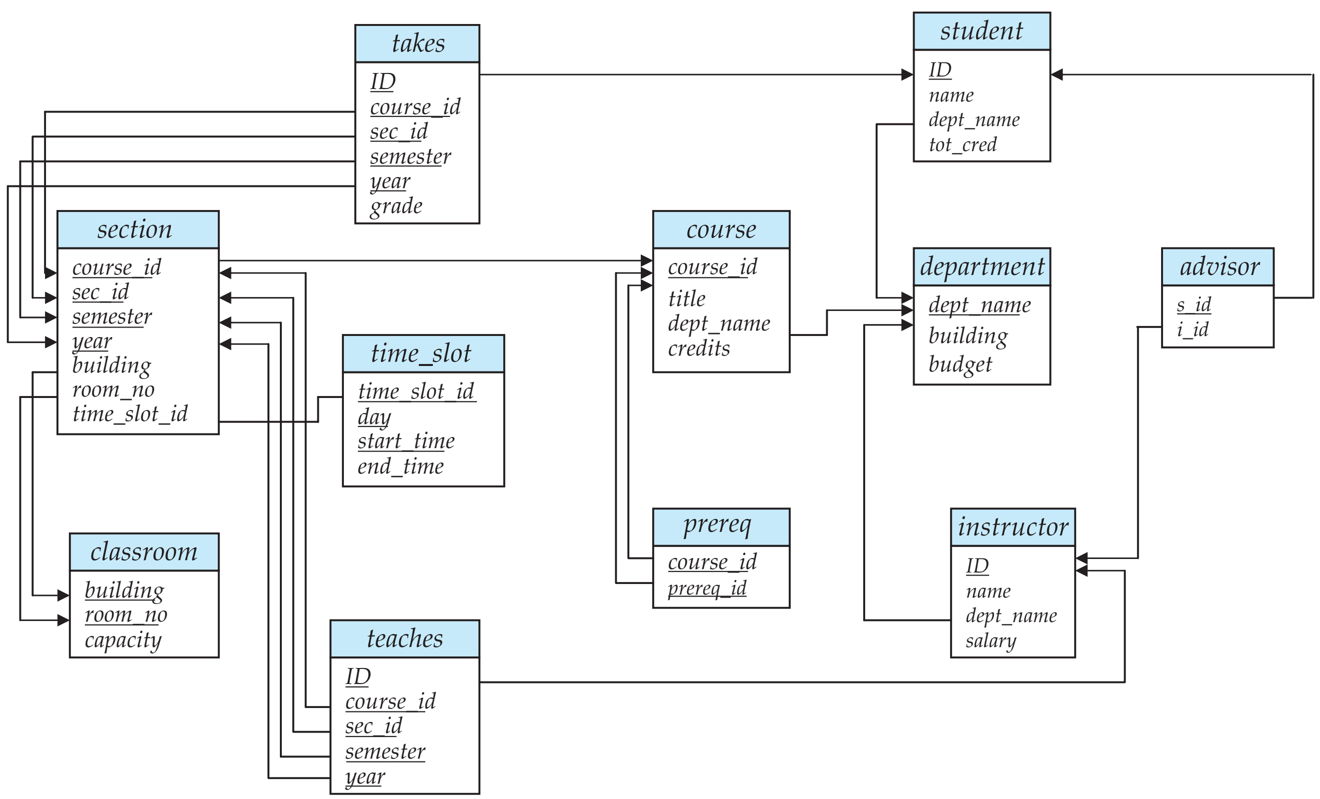
- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword DISTINCT after select.
- To find the names of all departments with instructor, and remove duplicates:

```
SELECT DISTINCT name
FROM   instructor;
```

- The keyword ALL specifies that duplicates must not be removed.

```
SELECT ALL name
FROM   instructor;
```

SQL: The SELECT Clause



- An asterisk in the select clause denotes "all attributes"

```
SELECT *
FROM   instructor;
```

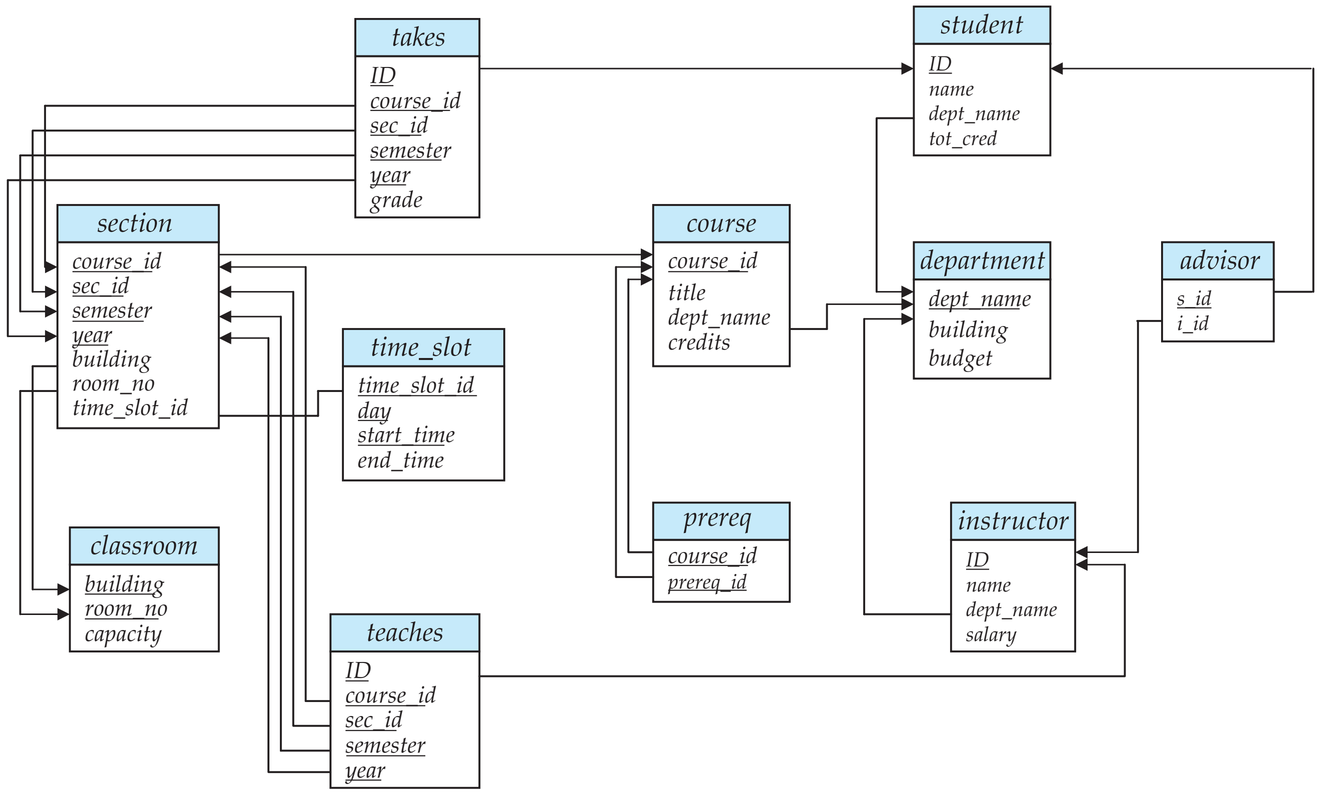
- The select clause can contain arithmetic expressions ranging over the operations +, –, *, and /, and in which constants or attributes of tuples may occur as operands.

- The query

```
SELECT ID, name, salary/12
FROM   instructor;
```

would return a relation that is the same as the instructor relation, except that the value of the attribute salary is divided by 12.

SQL: The WHERE Clause

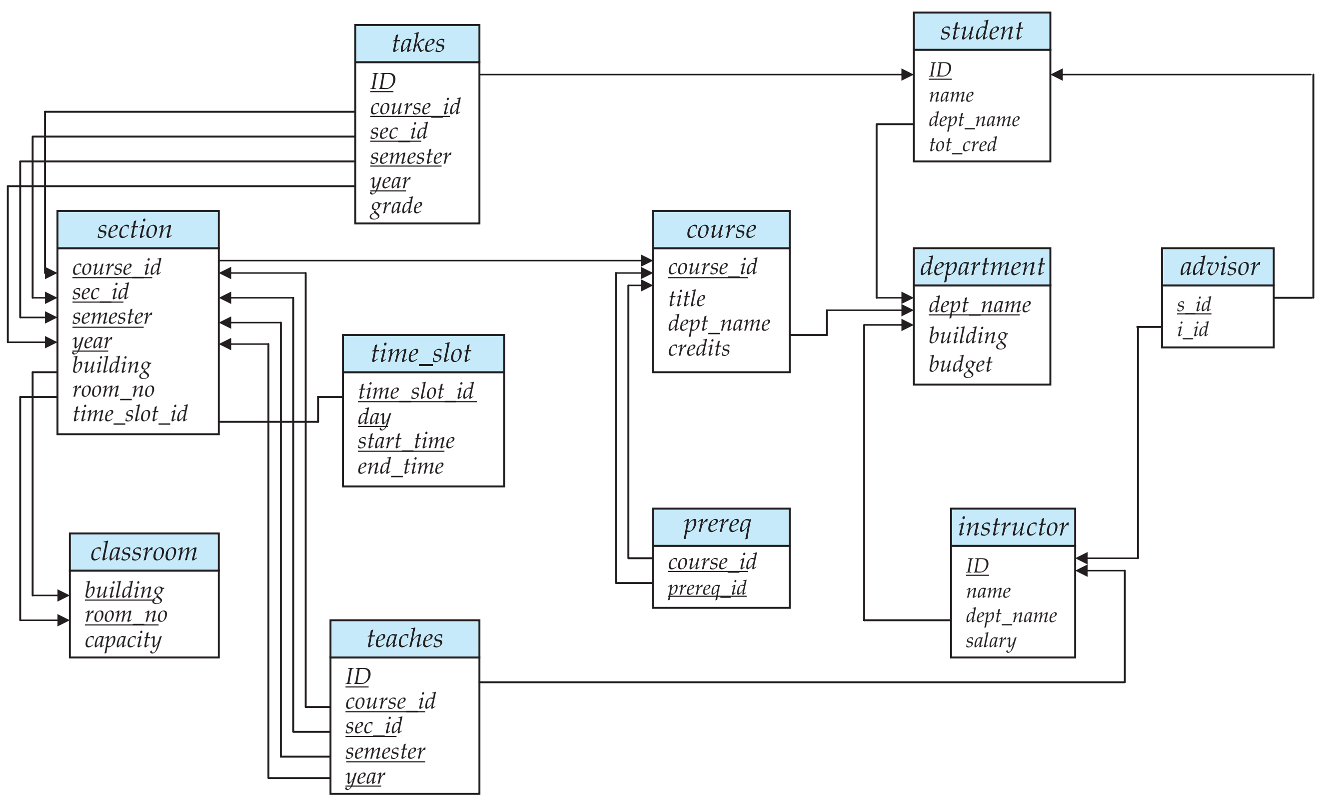


- The WHERE clause specifies conditions that every tuple in the result must satisfy.
- It corresponds to the selection operation of the relational algebra.
- To find all instructors in the Computer Science department with salary > 80000:

```
SELECT name
FROM   instructor
WHERE  dept_name = 'Comp. Sci'
AND    salary > 80000;
```

- Comparison results can be combined using the logical connectives AND, OR, and NOT.
- Comparisons can be applied to results of arithmetic expressions.

SQL: The FROM Clause



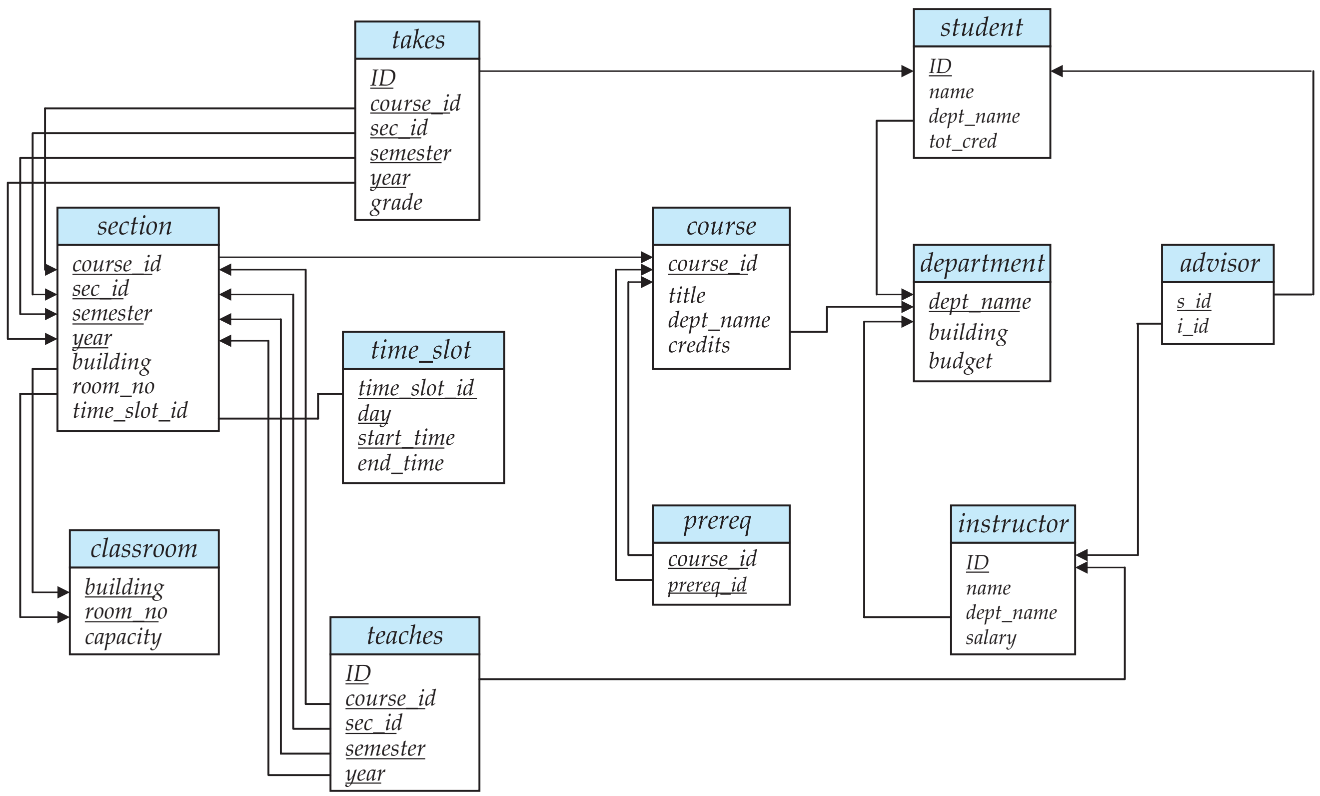
- The FROM clause lists the relations involved in the query.
- It corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product instructor X teaches:

```
SELECT *
FROM   instructor, teaches;
```

generates every possible (instructor, teaches) pair, with all attributes from both relations.

- Cartesian product not is very useful directly, but useful combined with the where-clause condition (selection operation in relational algebra).

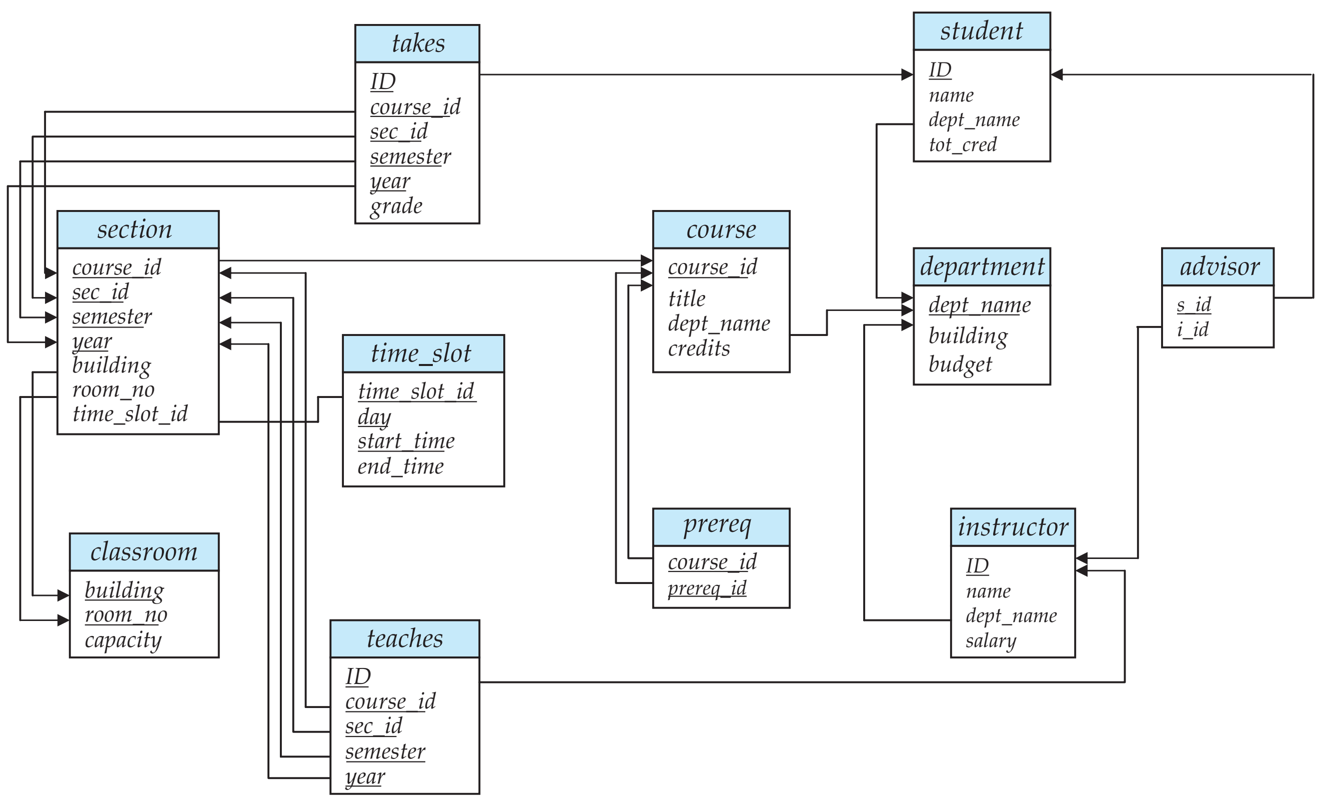
SQL: Set Operations: Union



- To find courses that ran in Fall 2009 or in Spring 2010:

```
(SELECT course_id
 FROM   section
 WHERE  sem = 'Fall' AND year = '2009')
UNION
(SELECT course_id
 FROM   section
 WHERE  sem = 'Spring' AND year = '2010')
;
```

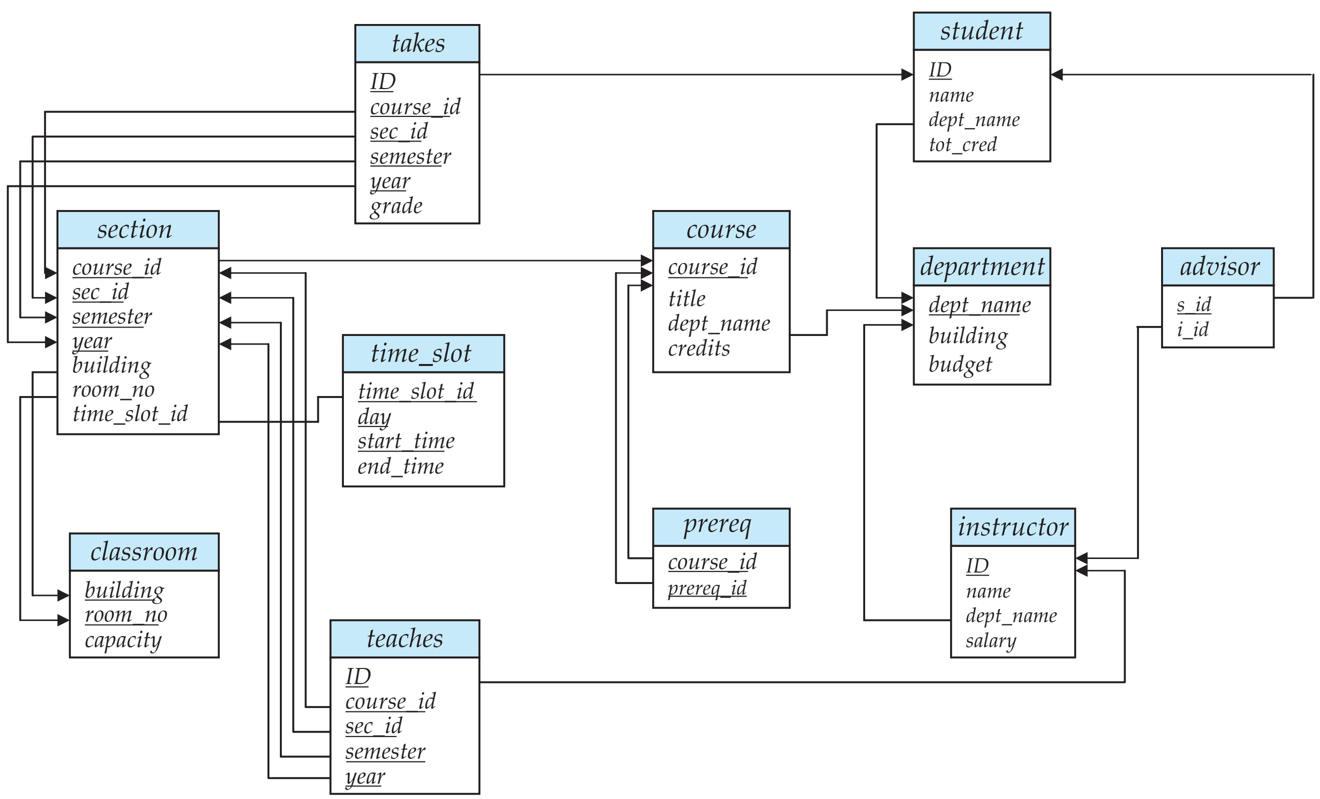
SQL: Set Operations: Intersection



- To find courses that ran in Fall 2009 and in Spring 2010:

```
(SELECT course_id
 FROM   section
 WHERE  sem = 'Fall' AND year = '2009')
INTERSECT
(SELECT course_id
 FROM   section
 WHERE  sem = 'Spring' AND year = '2010')
;
```

SQL: Set Operations: Difference



- To find courses that ran in Fall 2009 <u>but not</u> in Spring 2010:

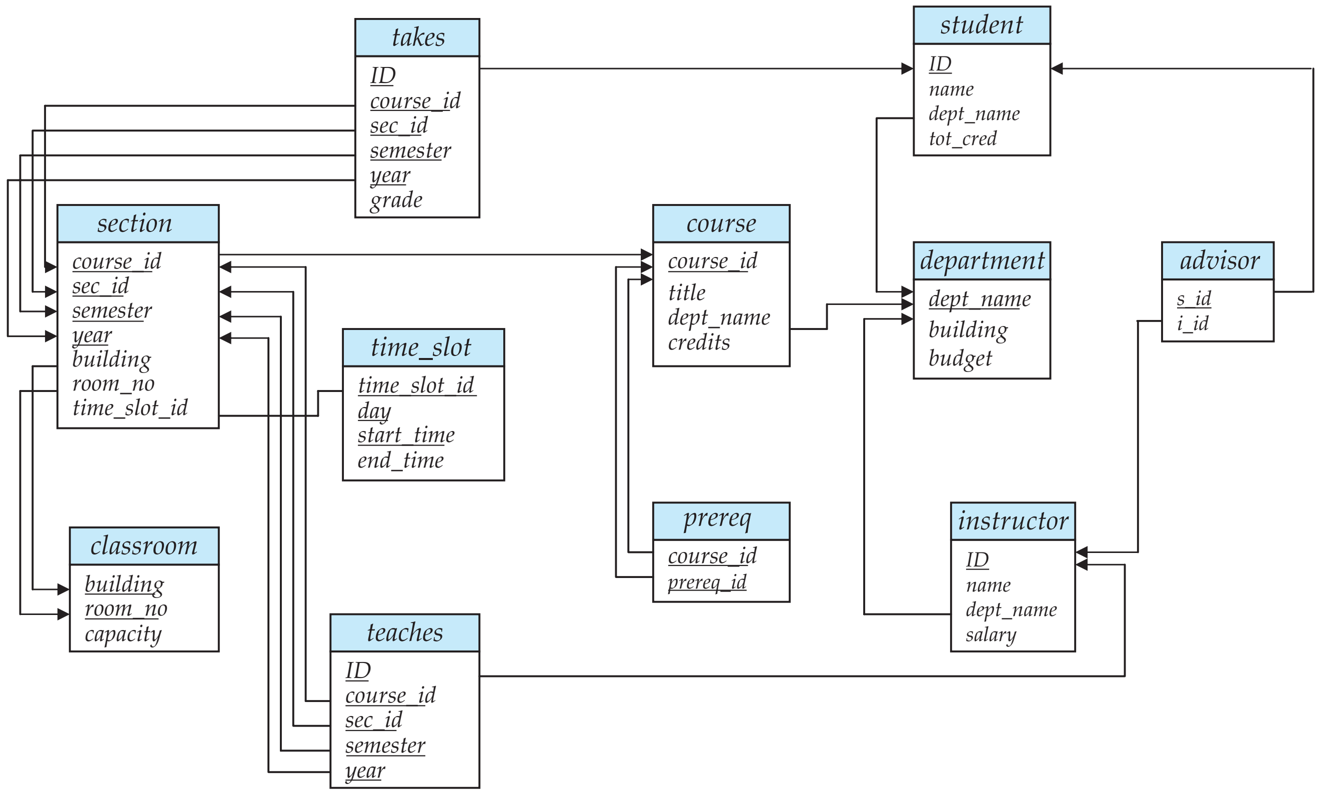
```
(SELECT course_id
 FROM   section
 WHERE  sem = 'Fall' AND year = '2009')
EXCEPT
(SELECT course_id
 FROM   section
 WHERE  sem = 'Spring' AND year = '2010')
;
```

- Oracle uses MINUS rather than EXCEPT.

SQL: Retaining Duplicates in Set Operations

- UNION, INTERSECT, and EXCEPT automatically eliminate duplicates
- To retain duplicates, we use the corresponding multiset/bag versions UNION ALL, INTERSECT ALL and EXCEPT ALL.
- Suppose a tuple occurs *m* times in *R* and *n* times in *S*, then, it occurs:
  - $m+n$ times in R UNION ALL S
  - $min(m,n)$ times in R INTERSECT ALL S
  - $max(0, m-n)$ times in R EXCEPT ALL S

SQL: Joins



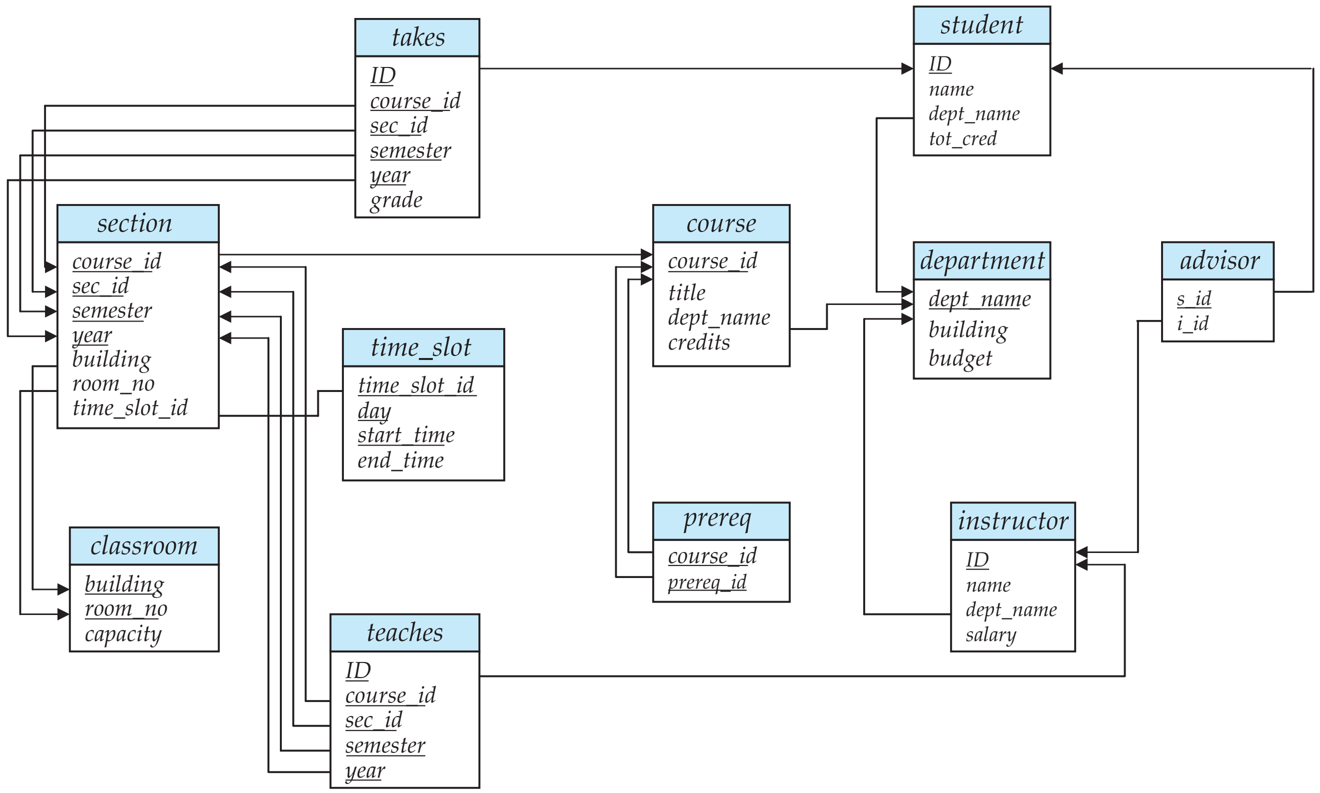
- For all instructors who teach some course, find their names and the course ID of the courses they teach:

```
SELECT instructor.name, teaches.course_id
FROM   instructor, teaches
WHERE  instructor.ID = teaches.ID;
```

- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department:

```
SELECT s.course_id, s.semester, s.year, c.title
FROM   section s, course c
WHERE  s.course_id = c.course_id
AND    c.dept_name = 'Comp. Sci.'
```

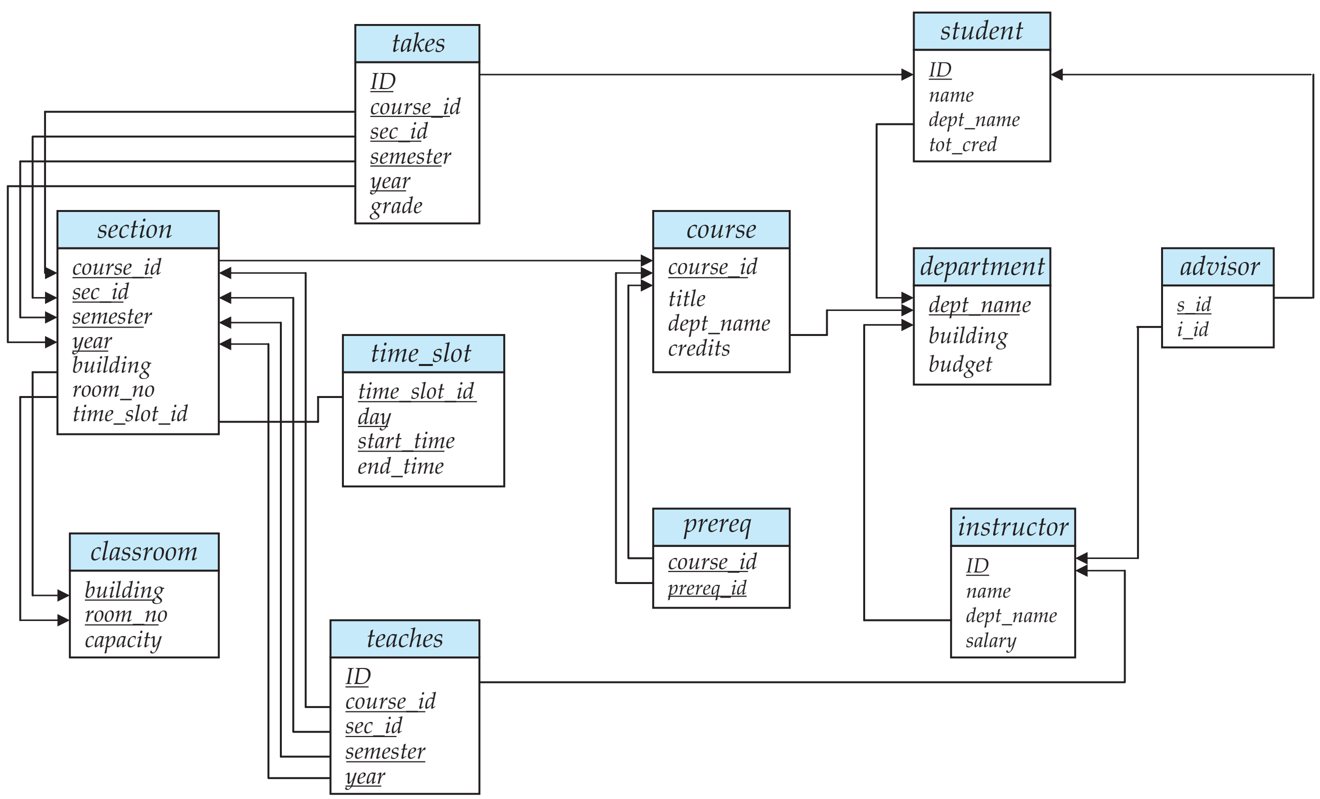
SQL: Joins



- We can also use the JOIN/ USING keywords explicitly.
- For all instructors who teach some course, find their names and the course ID of the courses they teach (same as before):

```
SELECT name, course_id
FROM   instructor
JOIN   teaches
USING  (ID);
```

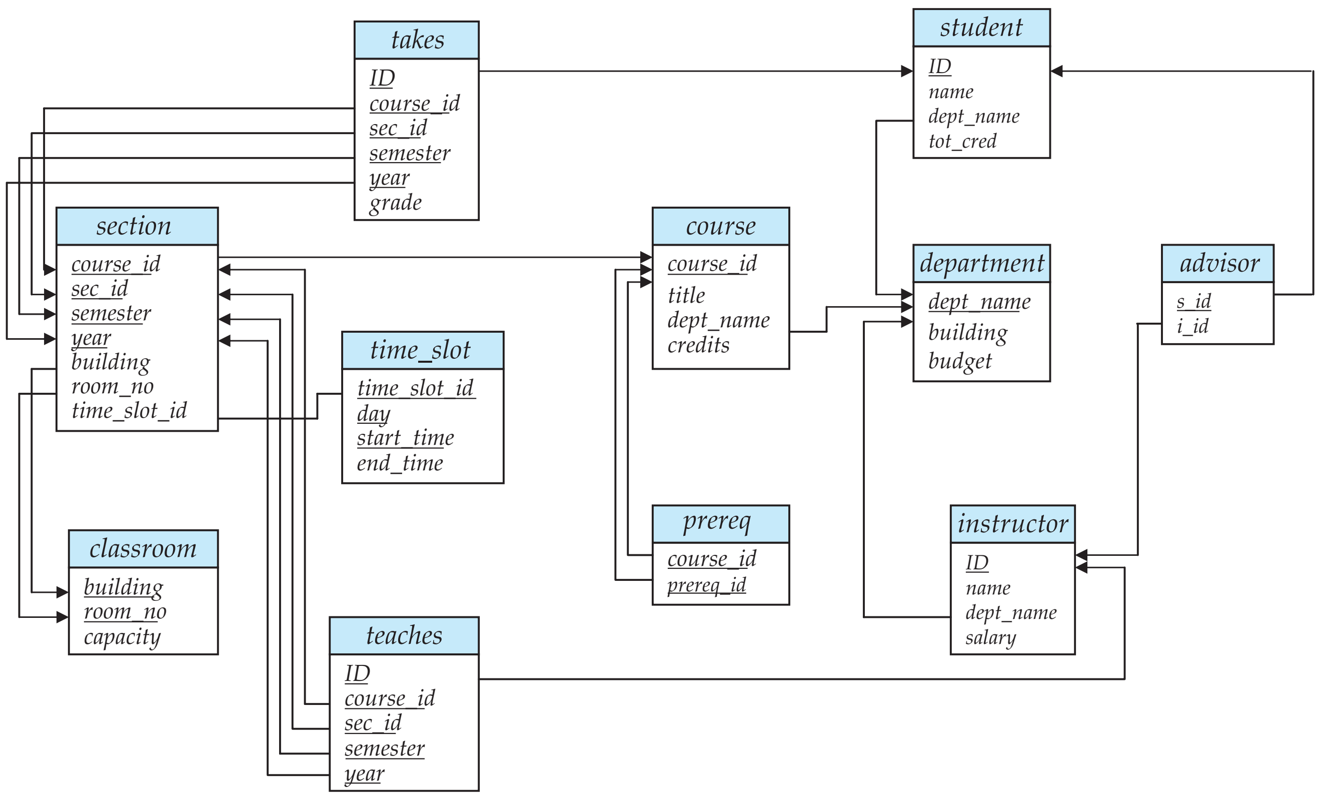
SQL: Natural Join



- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column:

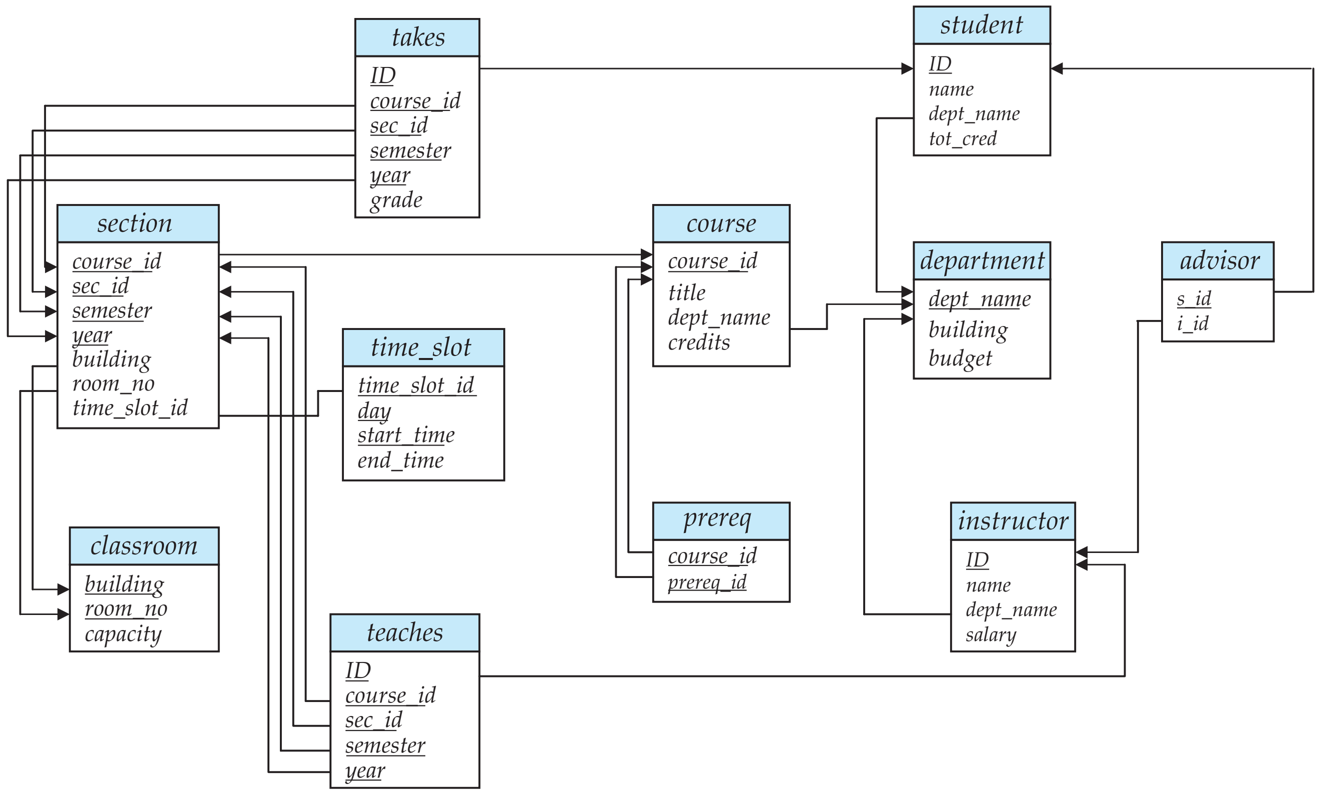
```
     SELECT *
       FROM instructor
NATURAL JOIN teaches;
```

SQL: Natural Join



- There is danger in natural join!
- Beware of unrelated attributes with same name that get equated incorrectly!

SQL: Natural Join



- List the names of instructors along with the titles of courses that they teach:
  - Incorrect version (erroneously makes course.dept_name = instructor.dept_name)

```
SELECT name, title
FROM   instructor NATURAL JOIN
       teaches    NATURAL JOIN
       course;
```

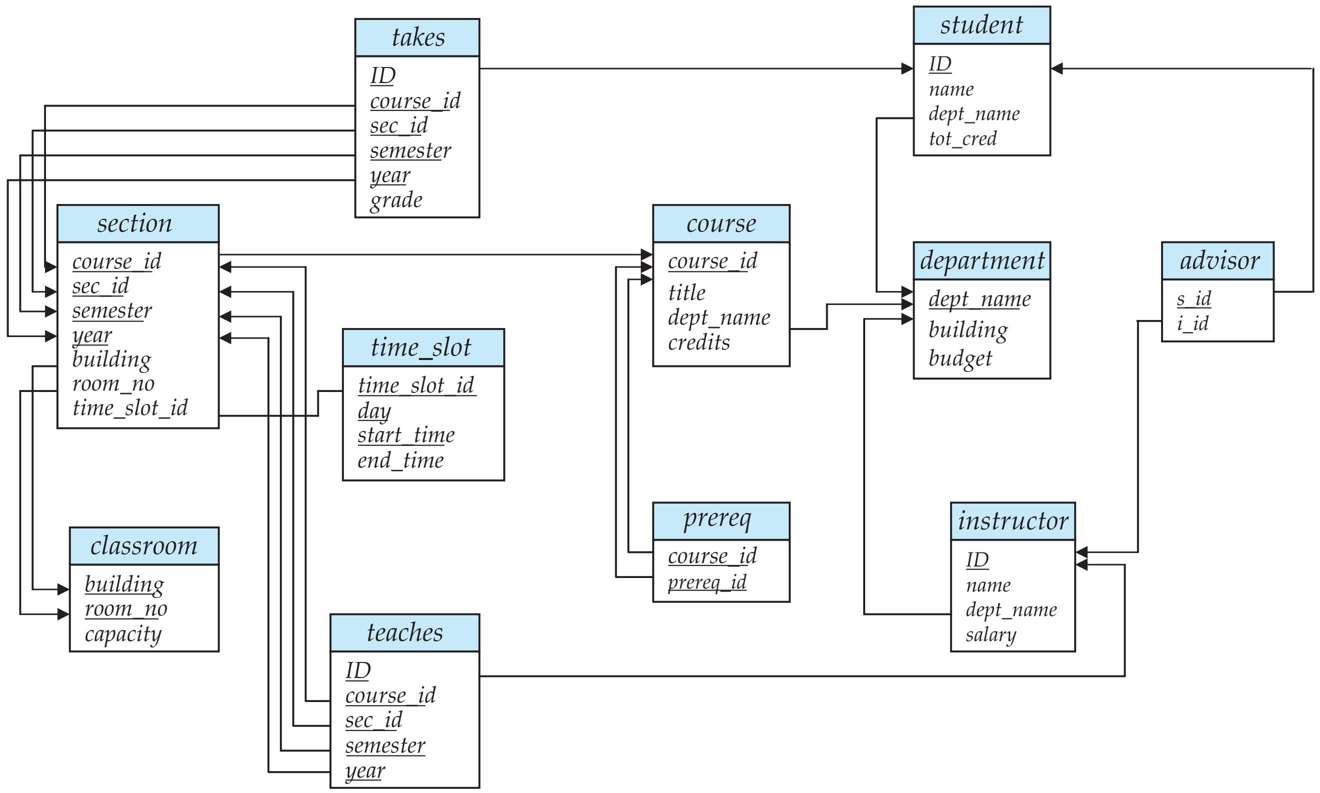
  - Correct version (an additional, specific join condition)

```
SELECT name, title
FROM   instructor NATURAL JOIN teaches, course
WHERE  teaches.course_id = course.course_id;
```

  - Another correct version (different joins)

```
SELECT name, title
FROM   (instructor NATURAL JOIN teaches)
JOIN   course USING (course_id);
```

SQL: Renaming



- SQL allows renaming relations and attributes using the AS clause of the form:

```
<old-name> AS <new-name>
```

- E.g.

```
SELECT ID, name, salary/12 AS monthly_salary
FROM   instructor;
```

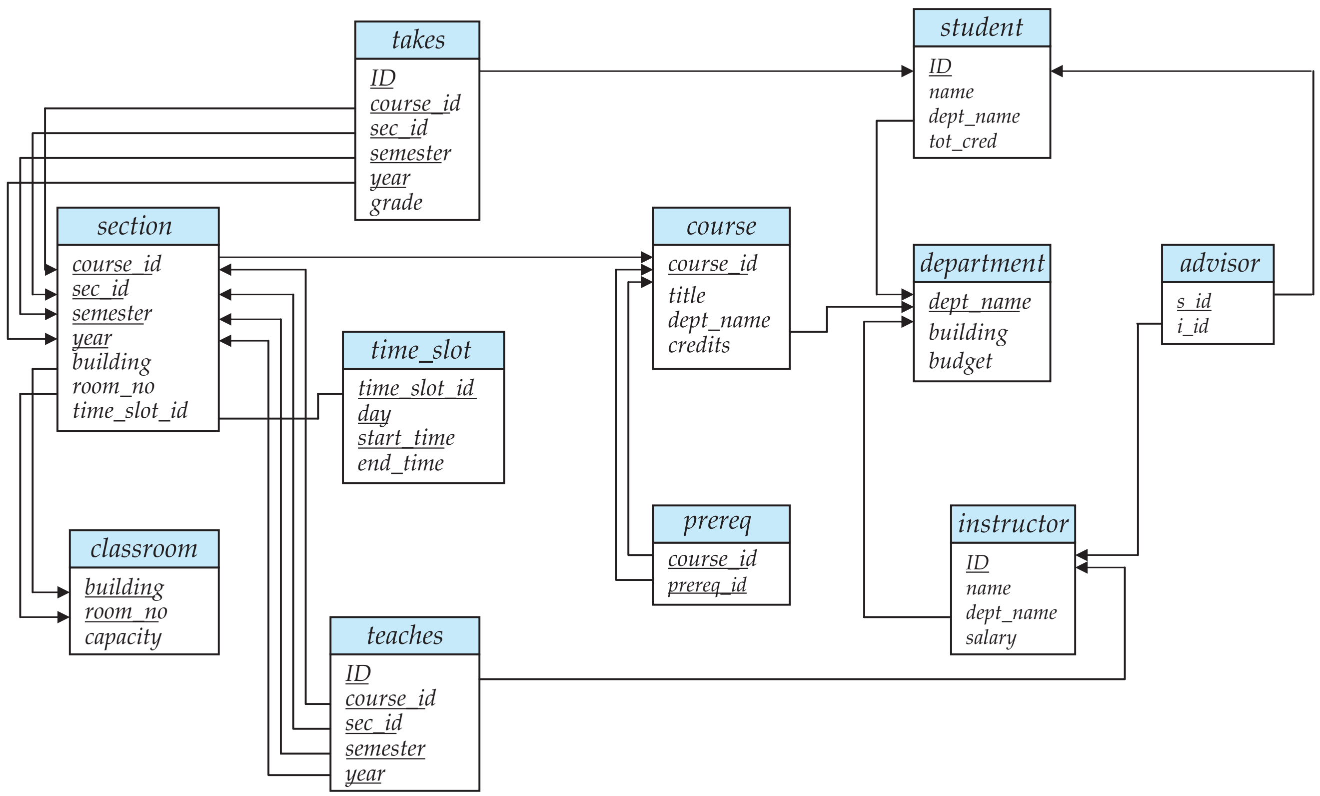
- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci':

```
SELECT DISTINCT T.name
FROM   instructor AS T, instructor AS S
WHERE  T.salary > S.salary
AND    S.dept_name = 'Comp. Sci.';
```

- Keyword AS is optional and may be omitted
- Keyword AS **<u>must</u>** be omitted in Oracle SQL

SQL: Strings

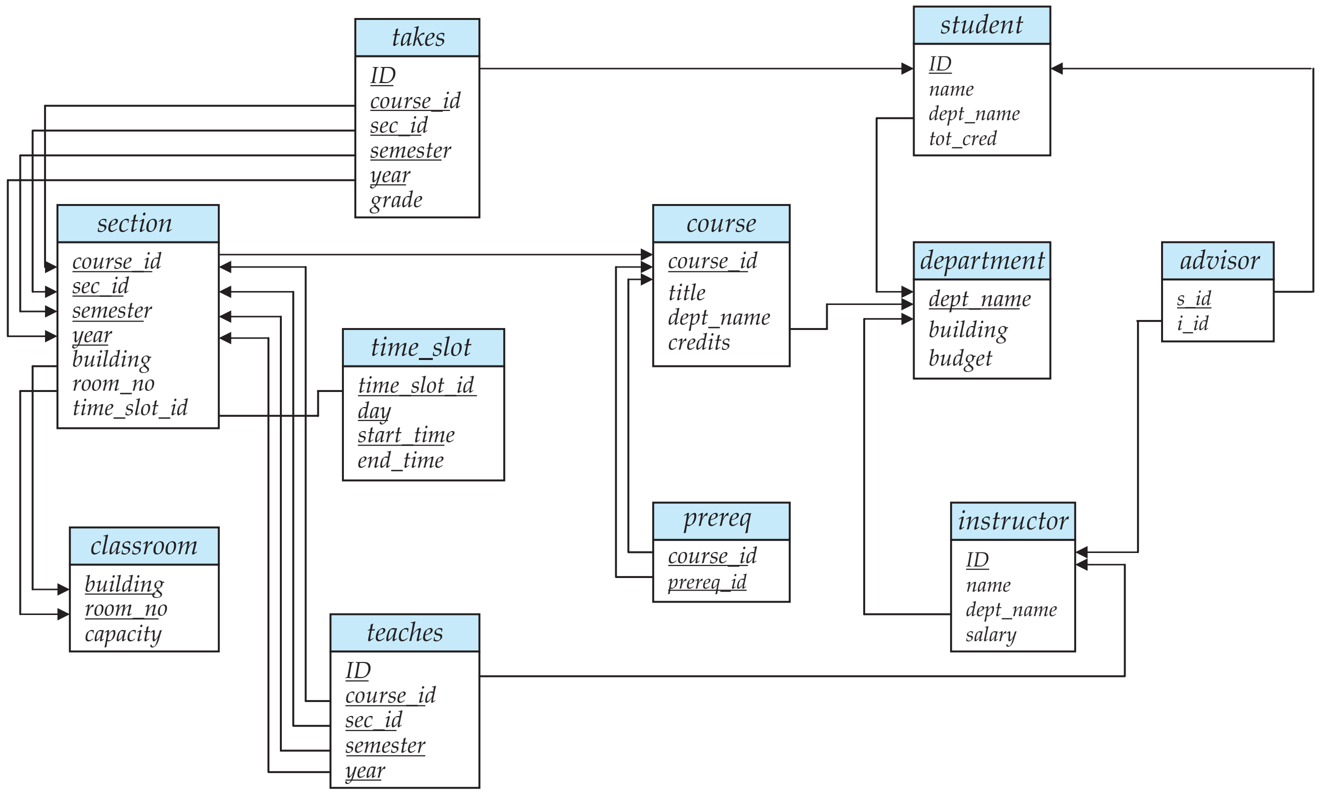
SQL: String Operations



- SQL includes a string-matching operator for comparisons on character strings.
- The operator LIKE uses patterns that are specified using three special characters:
  - percent (%), which matches any substring
  - underscore (_), which matches any character
  - use backslash (\) as the escape character if you want to use % or _ as actual characters.
- Find the names of all instructors whose name includes the substring "dar":

```
SELECT name
FROM   instructor
WHERE  name LIKE '%dar%';
```

SQL: String Operations



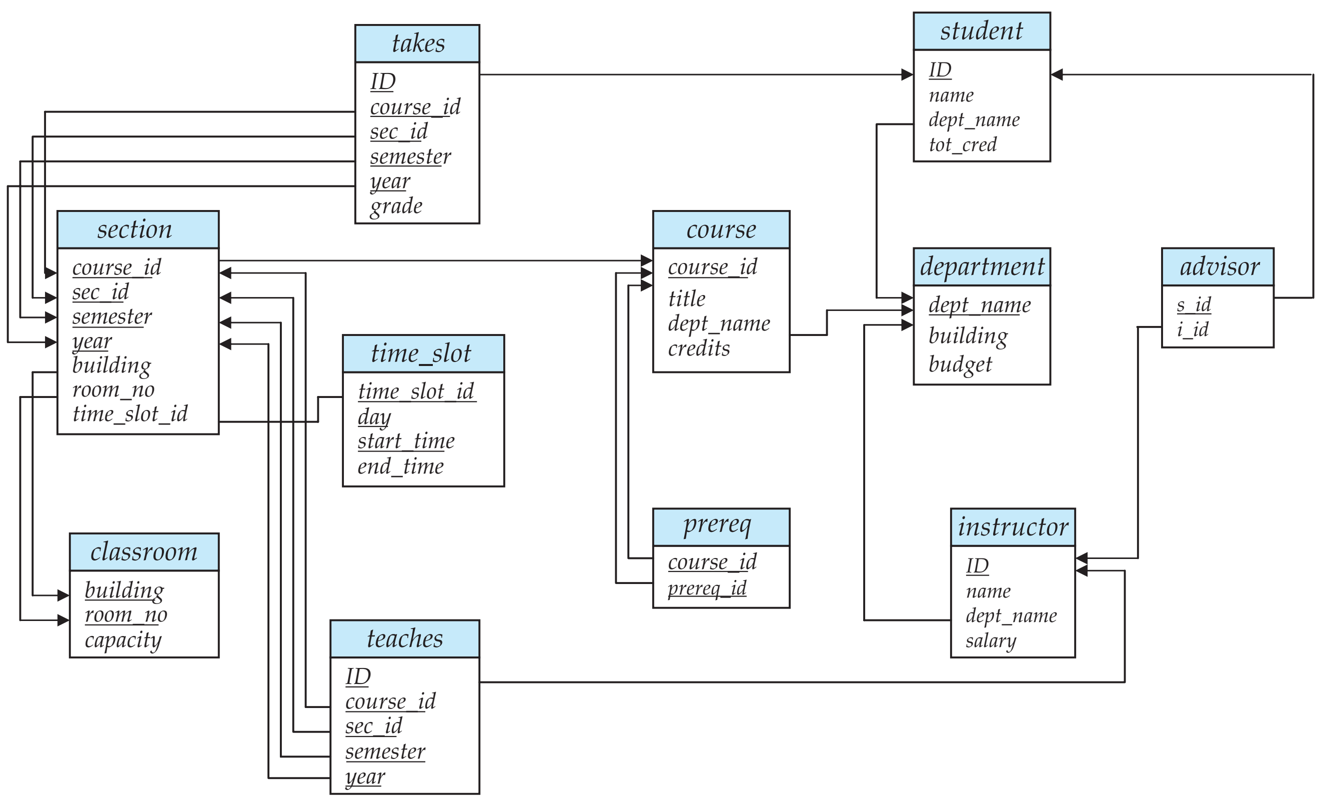
- Patterns are case-sensitive (as are literals).
- Some examples:
  - '100\%' matches the string '100%'
  - 'Intro%' matches any string beginning with "Intro".
  - '%Comp%' matches any string containing "Comp" as a substring.
  - '_ _ _' matches any string of exactly three characters.
  - '_ _ _ %' matches any string of at least three characters.

SQL: String Operations

- SQL supports a variety of string operations such as
  - concatenation (using '||')
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.
- Consult the DBMS manual for the details.

SQL: Extended Features in the WHERE Clause

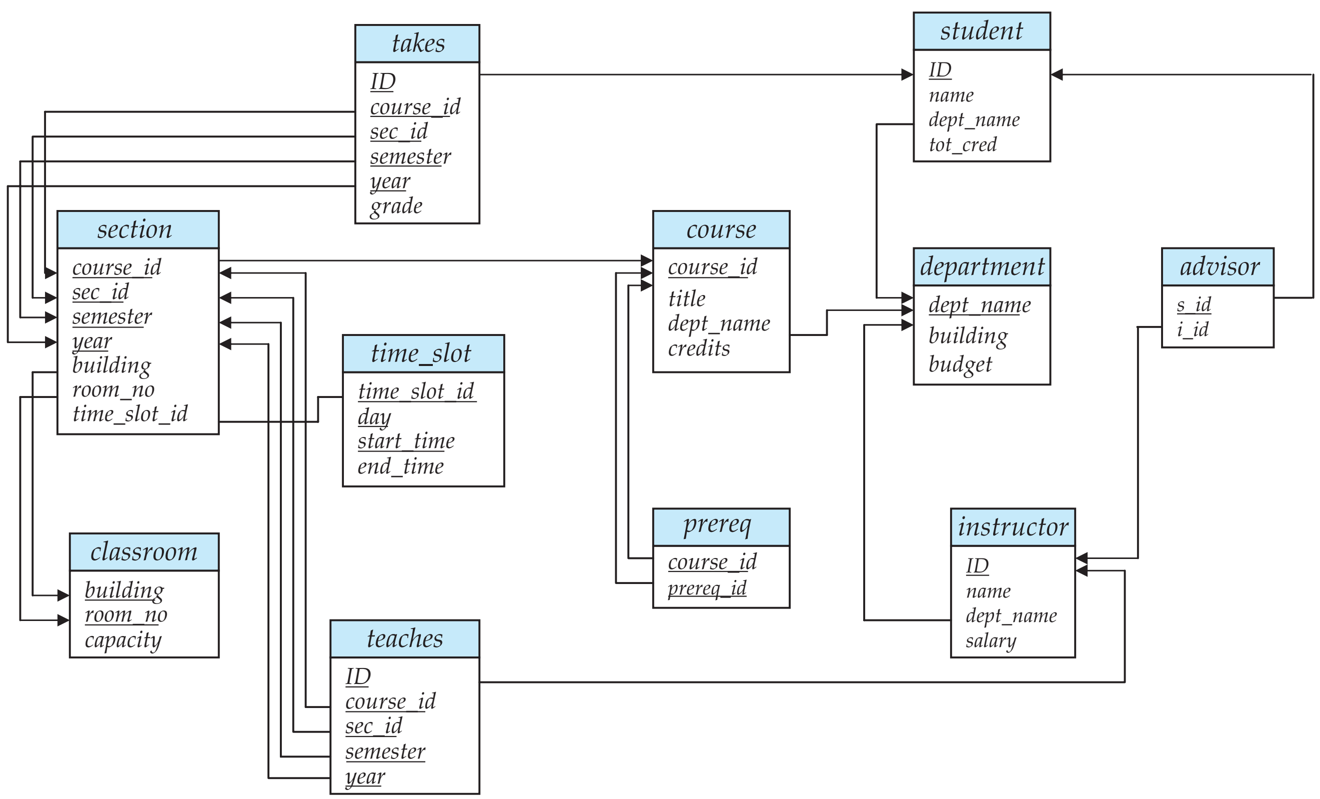
SQL: WHERE Clause Extended Features: Interval Membership



- SQL includes a BETWEEN comparison operator that can appear in a WHERE clause
- For example, find the names of all instructors such that \$90,000 ≤ salary ≤ \$100,000):

```
SELECT name
FROM   instructor
WHERE  salary BETWEEN 90000 AND 100000;
```

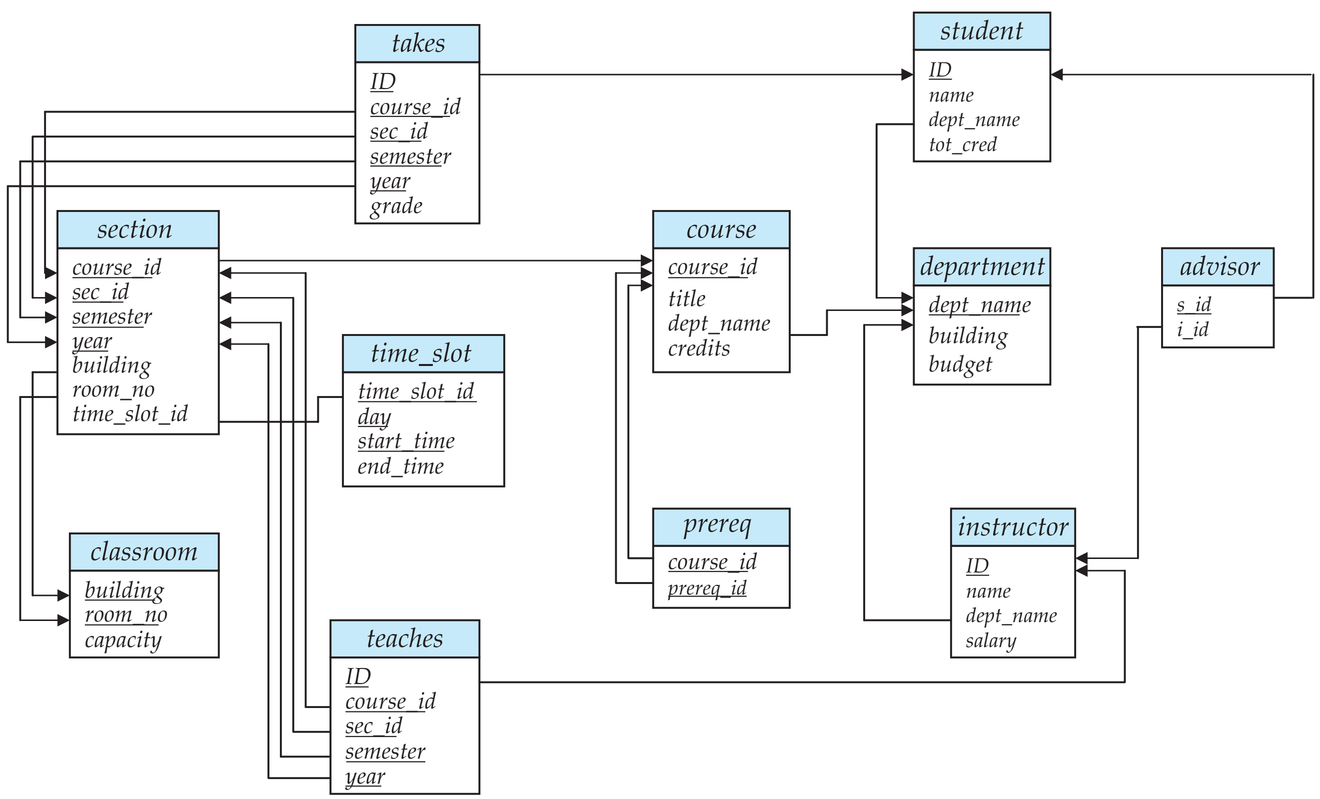
SQL: WHERE Clause Extended Features: Tuple Comparison



- SQL can do pairwise comparison of a list of elements:

```
SELECT name, course_id
FROM   instructor i, teaches t
WHERE  (i.ID,t.dept_name) = (t.ID,'Biology');
```

SQL: WHERE Clause Extended Features: NULL values



- It is possible for tuples to have a null value, denoted by NULL, for some of their attributes
- NULL denotes an value that may exist but hasn't been inserted into the tuple or a value that does not exist.
- The result of any arithmetic expression involving null is null (e.g., 5 + NULL returns NULL)
- The predicate IS NULL can be used to check for null values.
- For example, find all instructors whose salary is null:

```
SELECT name
FROM   instructor
WHERE  salary IS NULL;
```

SQL: WHERE Clause Extended Features: Tests and Comparisons on NULL and UNKNOWN

- Any comparison with NULL returns UNKNOWN, e.g., 5 < NULL or NULL <> NULL or NULL = NULL
- The three-valued logic using the truth value UNKNOWN is:
  - (UNKNOWN OR TRUE) = TRUE
    (UNKNOWN OR FALSE) = UNKNOWN
    (UNKNOWN OR UNKNOWN) = UNKNOWN
  - (TRUE AND UNKNOWN) = UNKNOWN
    (FALSE AND UNKNOWN) = FALSE
    (UNKNOWN and UNKNOWN) = UNKNOWN
  - (NOT UNKNOWN) = UNKNOWN
- "P IS UNKNOWN" evaluates to TRUE if predicate P evaluates to UNKNOWN
- The result of a WHERE clause predicate is treated as FALSE if it evaluates to UNKNOWN

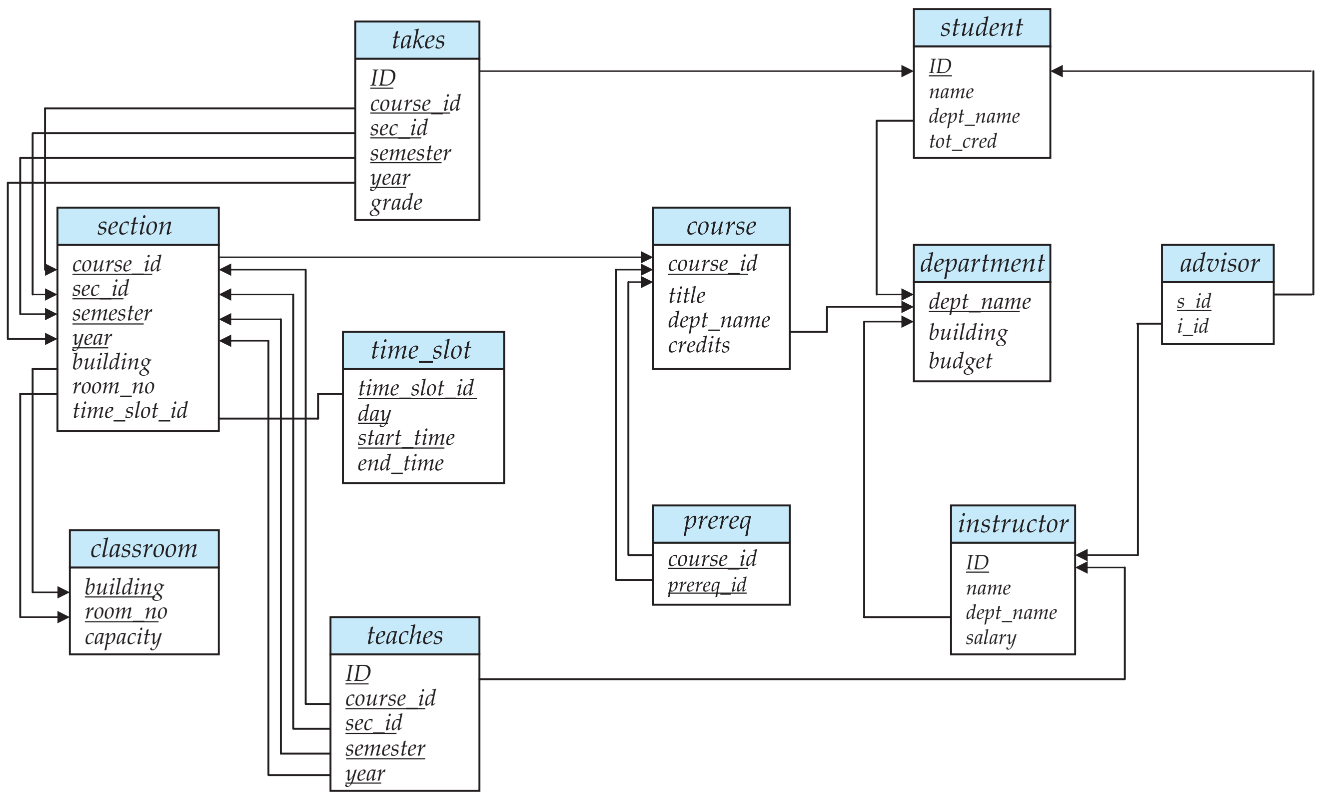
SQL: Aggregation

SQL: Aggregation

- These functions operate on the multiset of values of a column of a relation, and return a value

    AVG: average over
    MIN: minimum in
    MAX: maximum in
    SUM: sum of
    COUNT: number of

SQL: Aggregation



- Find the average salary of instructors in the Computer Science department:

```
SELECT AVG(salary)
FROM   instructor
WHERE  dept_name = 'Comp. Sci.';
```

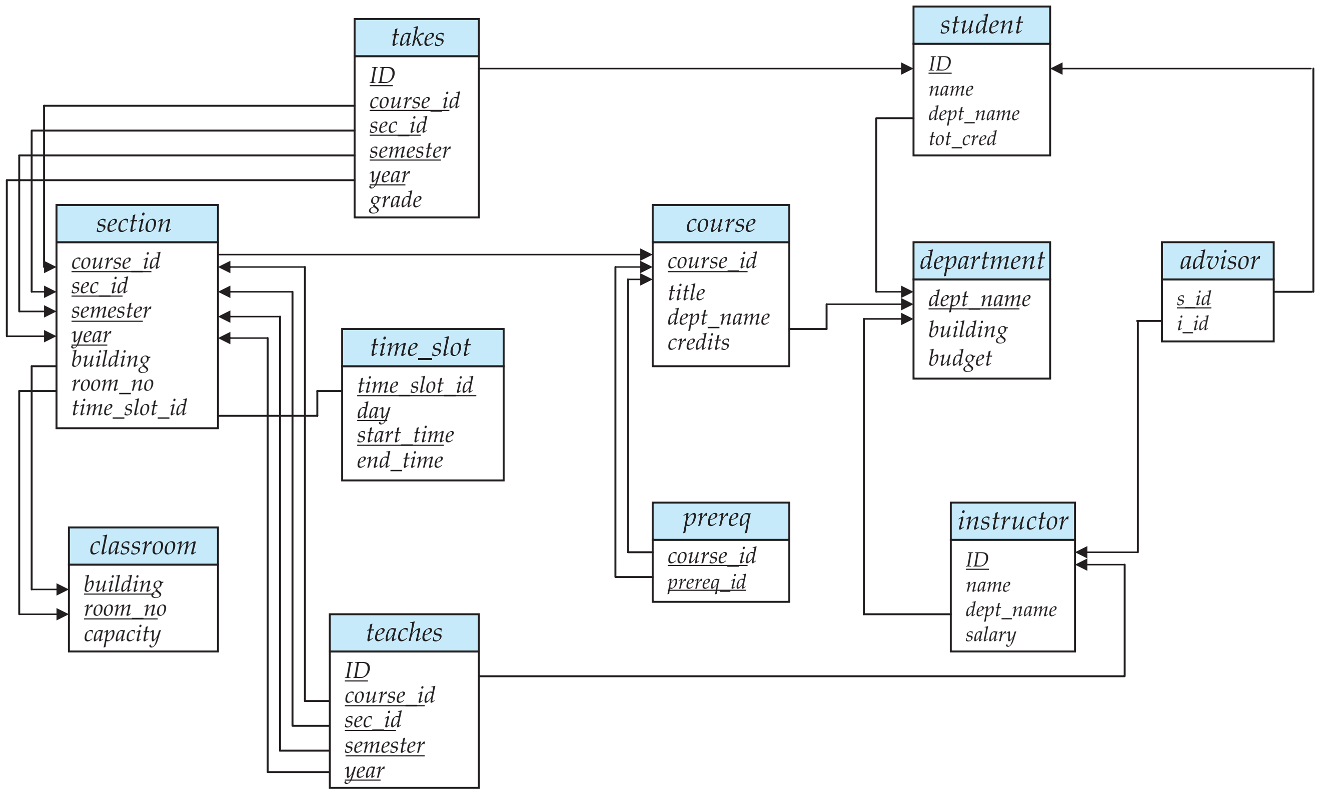
- Find the total number of instructors who teach a course in the Spring 2010 semester:

```
SELECT COUNT (DISTINCT ID)
FROM   teaches
WHERE  semester = 'Spring'
AND    year = 2010;
```

- Find the number of tuples in the course relation:

```
SELECT COUNT(*)
FROM   course;
```

SQL: Aggregation: Treatment of NULL Values

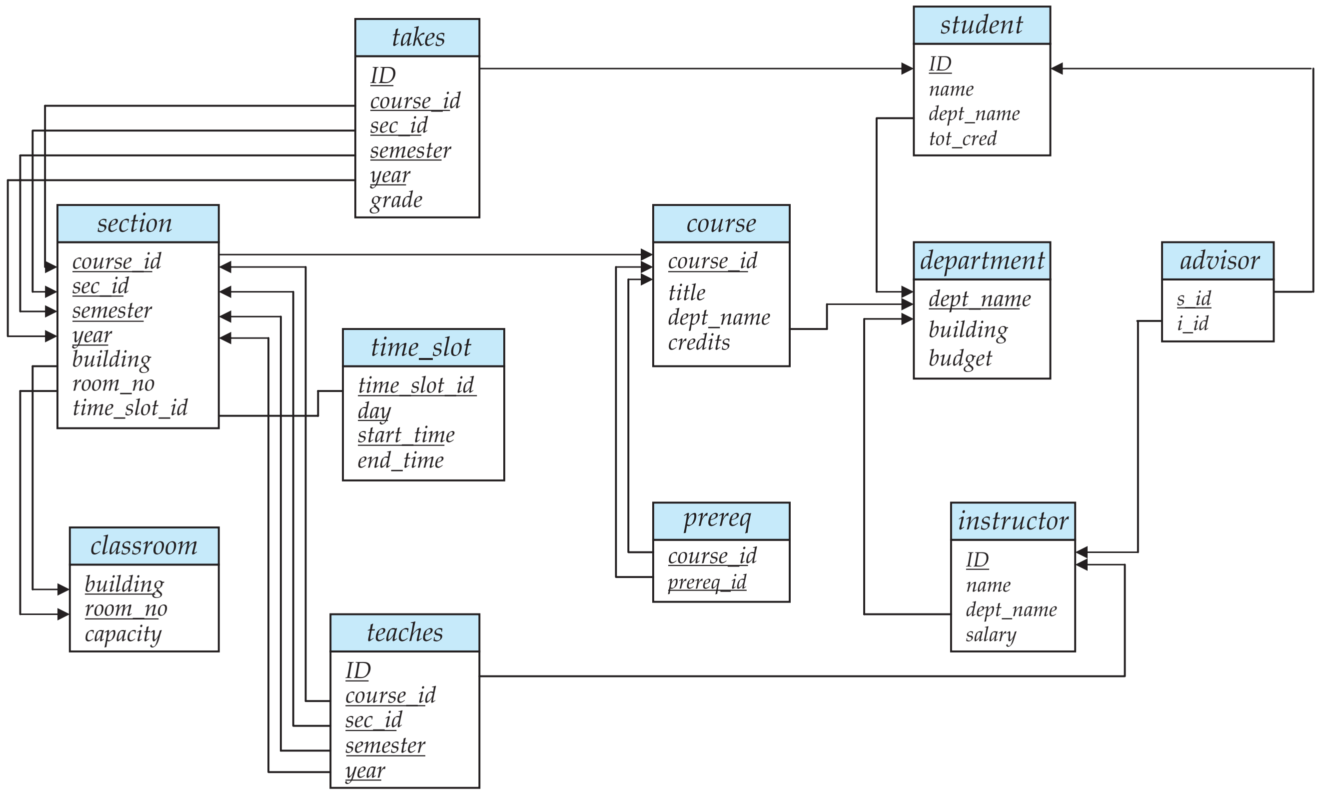


- Find the sum of all instructor salaries:

```
SELECT SUM(salary)
FROM   instructor;
```

- The evaluation ignores any NULL amount.
- The result is NULL if there is no non-NULL amount.
- All aggregate operations except COUNT(*) ignore tuples with NULL values on the aggregated attributes.
- What if the collection only has NULL values?
  - COUNT returns 0
  - all other aggregates return NULL

SQL: GROUP BY Aggregation

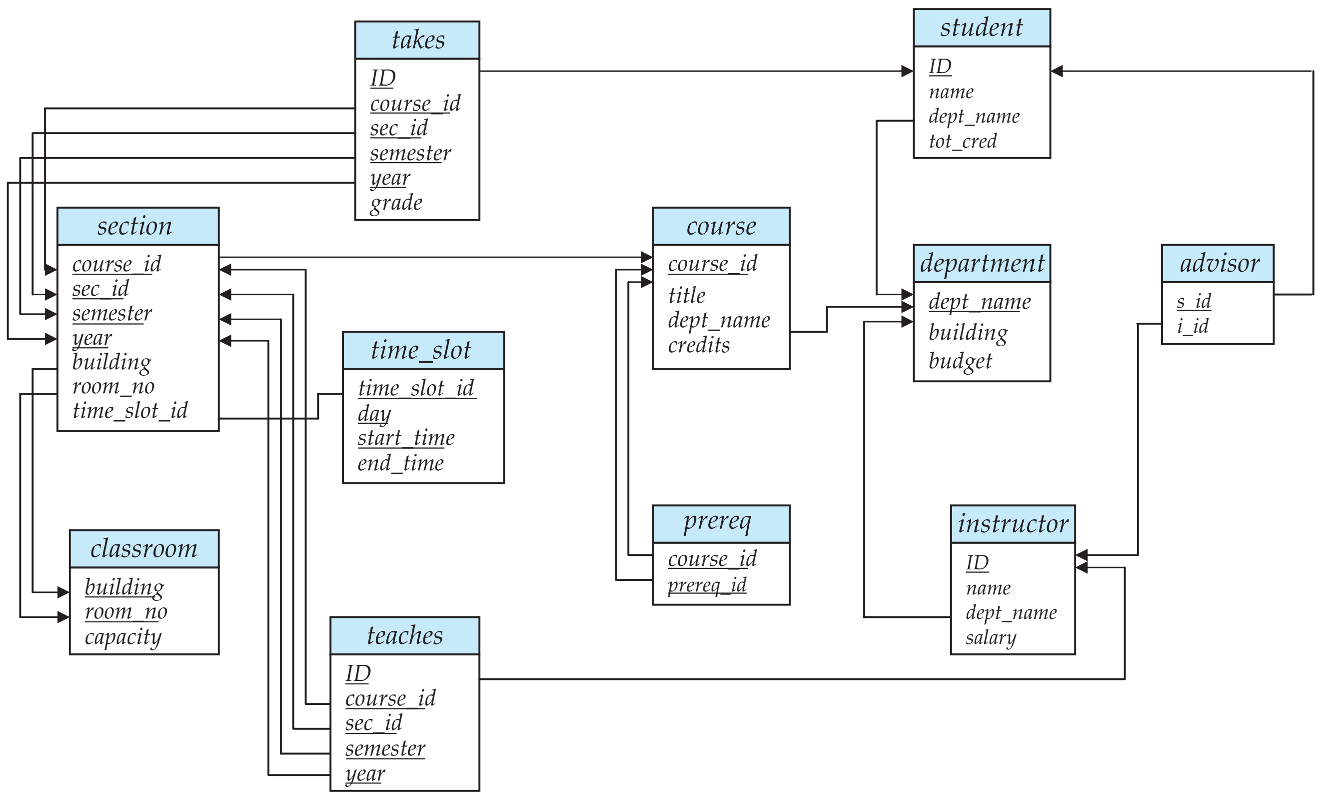


- Find the average salary of instructors in each department:

```
SELECT dept_name, AVG(salary)
FROM   instructor
GROUP BY dept_name;
```

- Note that departments with no instructors do not appear in result.

SQL: GROUP BY Aggregation



- Attributes in the SELECT clause that appear outside (i.e., are not arguments) of aggregate functions <u>must</u> appear in the GROUP BY list.

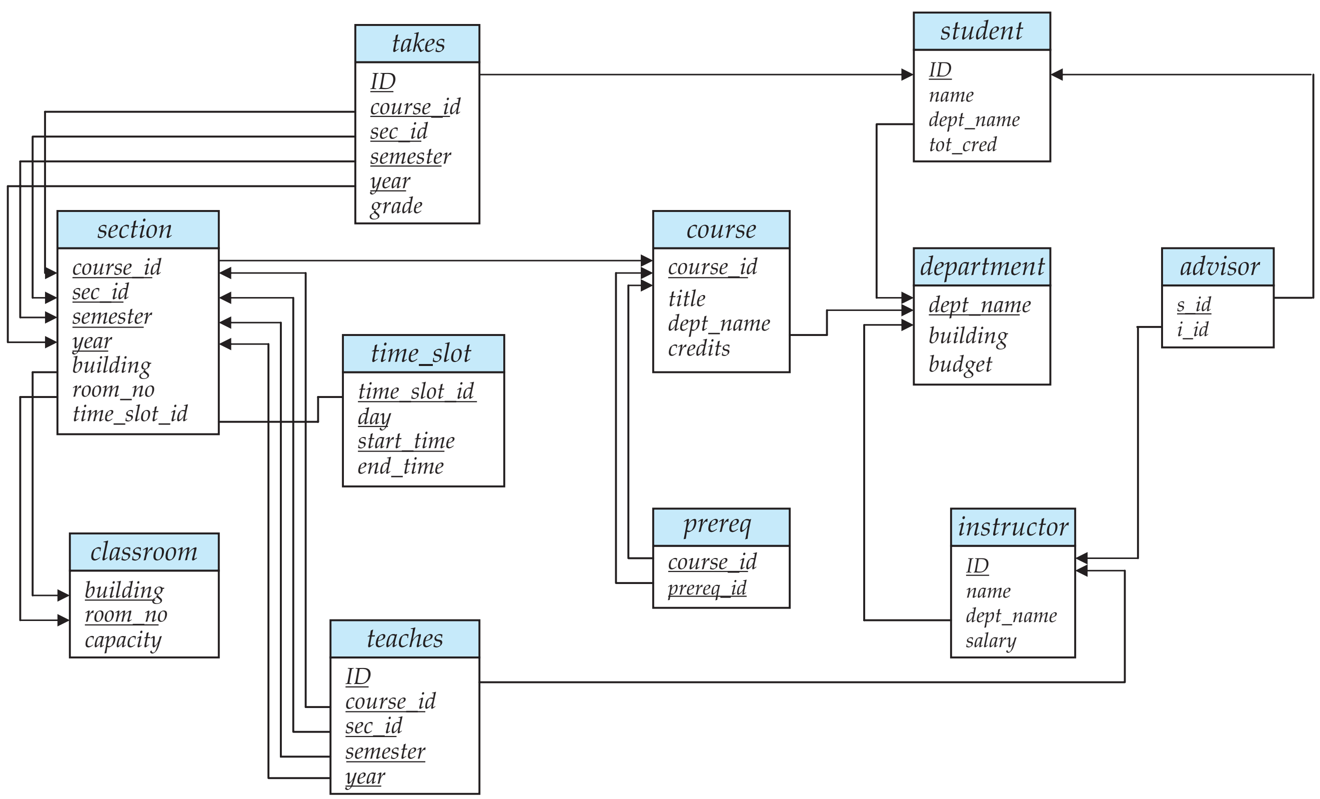
- An **<u>erroneous</u>** query:

```
SELECT dept_name, name, AVG(salary)
FROM   instructor
GROUP BY dept_name;
```

- A **<u>correct</u>** query (but probably not what you want!):

```
SELECT dept_name, name, AVG(salary)
FROM   instructor
GROUP BY dept_name, name;
```

SQL: GROUP BY Aggregation: The HAVING Clause

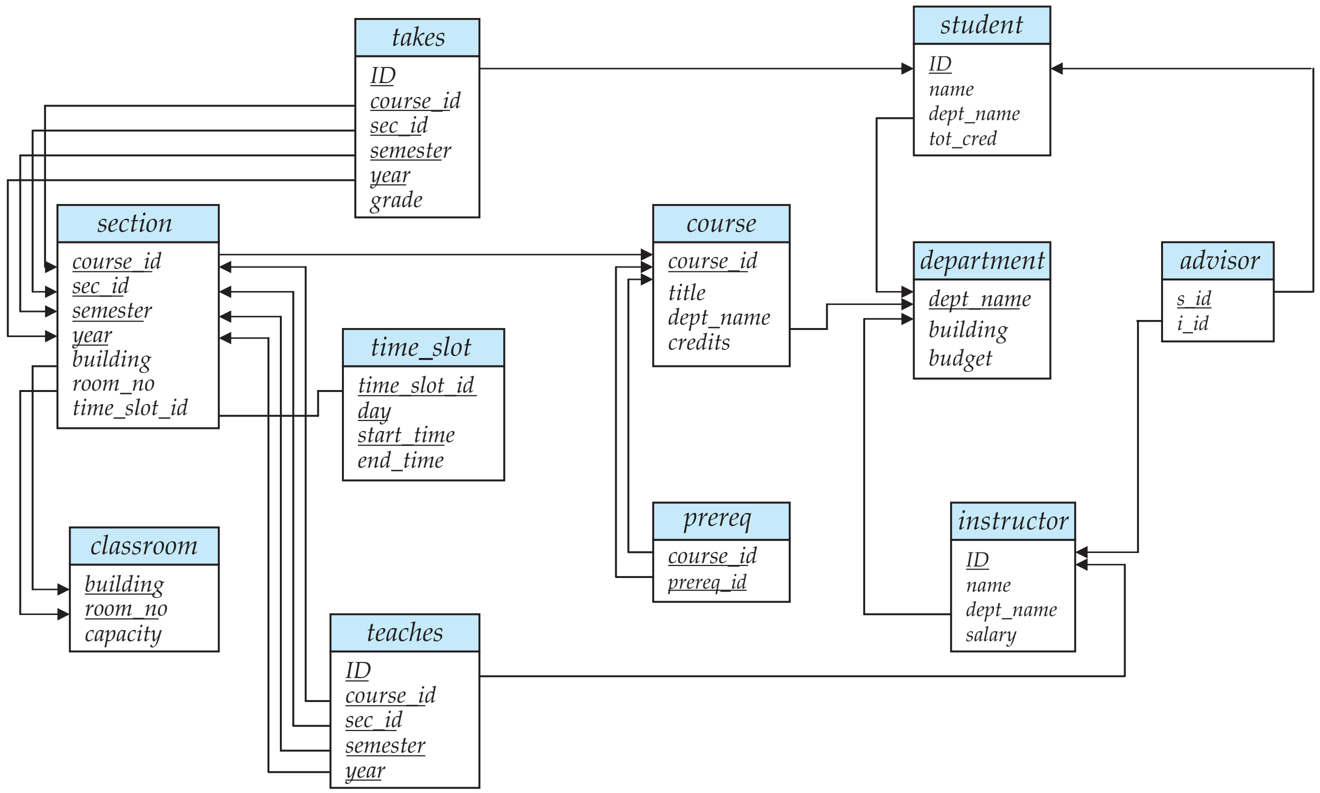


- Find the names and average salaries of all departments whose average salary is greater than 42000:

```
SELECT dept_name, AVG(salary)
FROM   instructor
GROUP BY dept_name
HAVING AVG(salary) > 42000;
```

- Whereas predicates in the WHERE clause are applied before forming groups, predicates in the HAVING clause are applied after the formation of groups.

SQL: Displaying Tuples in a Given Order



- List in alphabetic order the names of all instructors:

```
SELECT DISTINCT name
FROM   instructor
ORDER BY name;
```

- We may specify more than one attribute to order by

- For each attribute, we may specify DESC for descending order or ASC (which is the default) for ascending order.

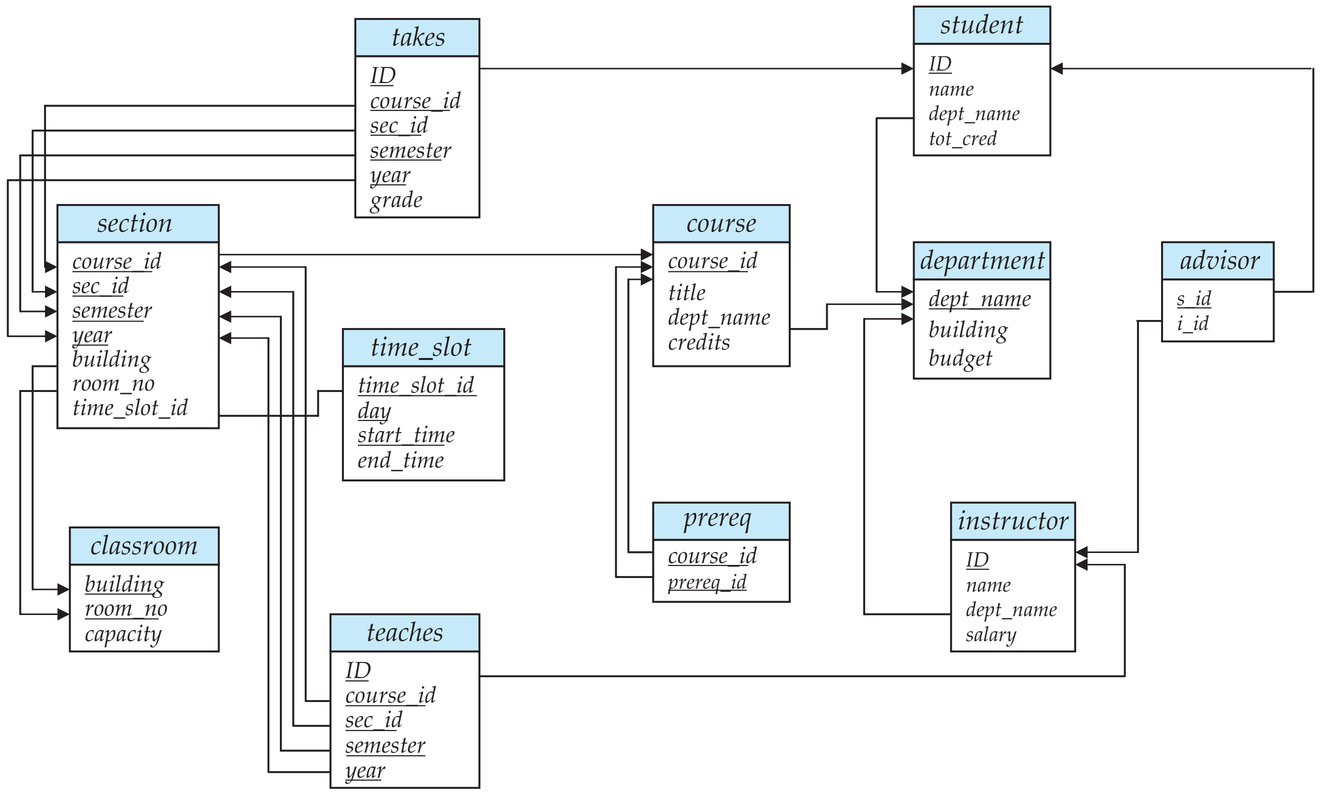
The University of Manchester

SQL: Nested Subqueries

SQL: Nested Subqueries

- SQL queries are compositional, i.e., they allow for the nesting of subqueries.
- A subquery is a SELECT-FROM-WHERE expression that is nested within another query.
- Common uses of subqueries are to perform:
  - set membership tests
  - set comparisons
  - empty set tests

SQL: Nested Subqueries: The IN Clause for Testing Set Membership



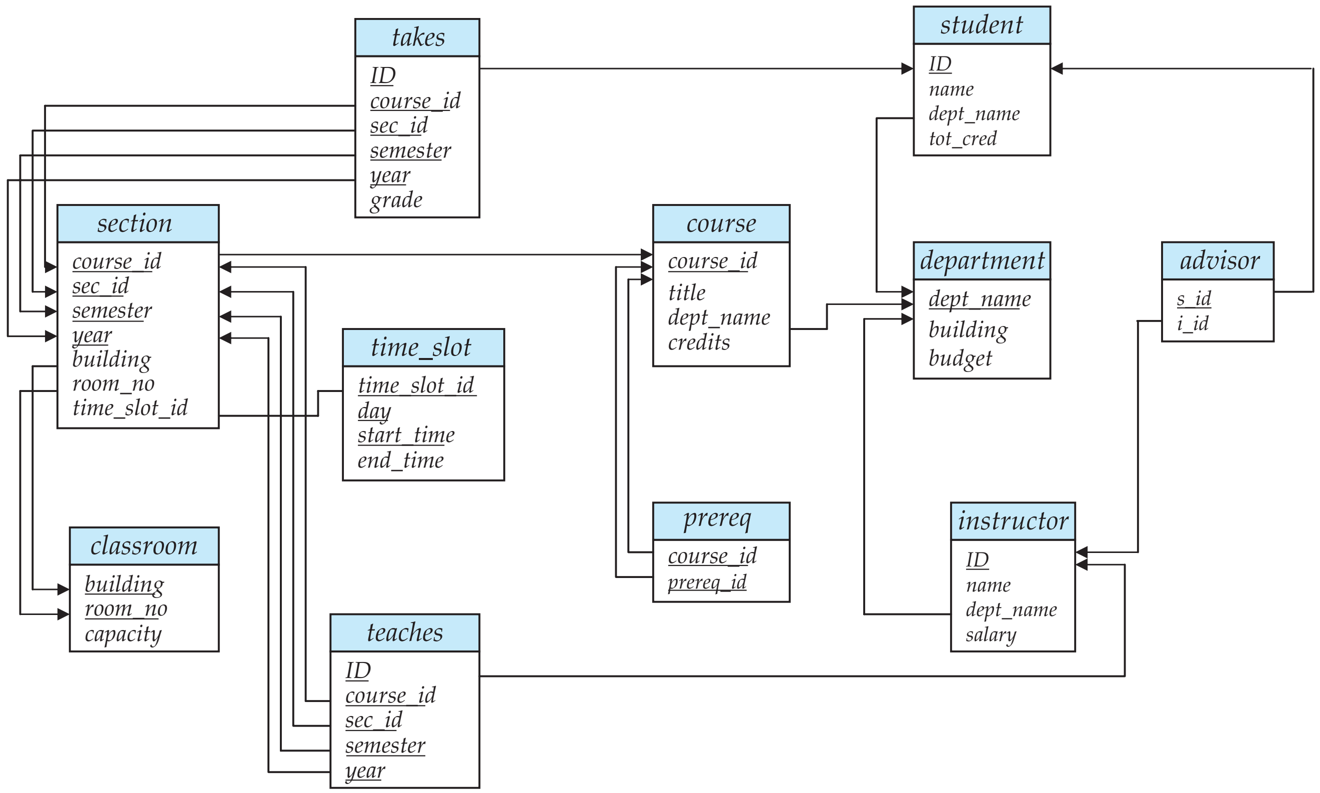
- Find courses offered in Fall 2009 and in Spring 2010:

```
SELECT DISTINCT course_id
FROM  section
WHERE semester = 'Fall'
AND   year = 2009
AND   course_id IN (SELECT course_id
                    FROM  section
                    WHERE semester = 'Spring'
                    AND year = 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010:

```
SELECT DISTINCT course_id
FROM  section
WHERE semester = 'Fall'
AND   year = 2009
AND   course_id NOT IN (SELECT course_id
                        FROM  section
                        WHERE semester = 'Spring'
                        AND year = 2010);
```

SQL: Nested Subqueries: The IN Clause for Testing Set Membership

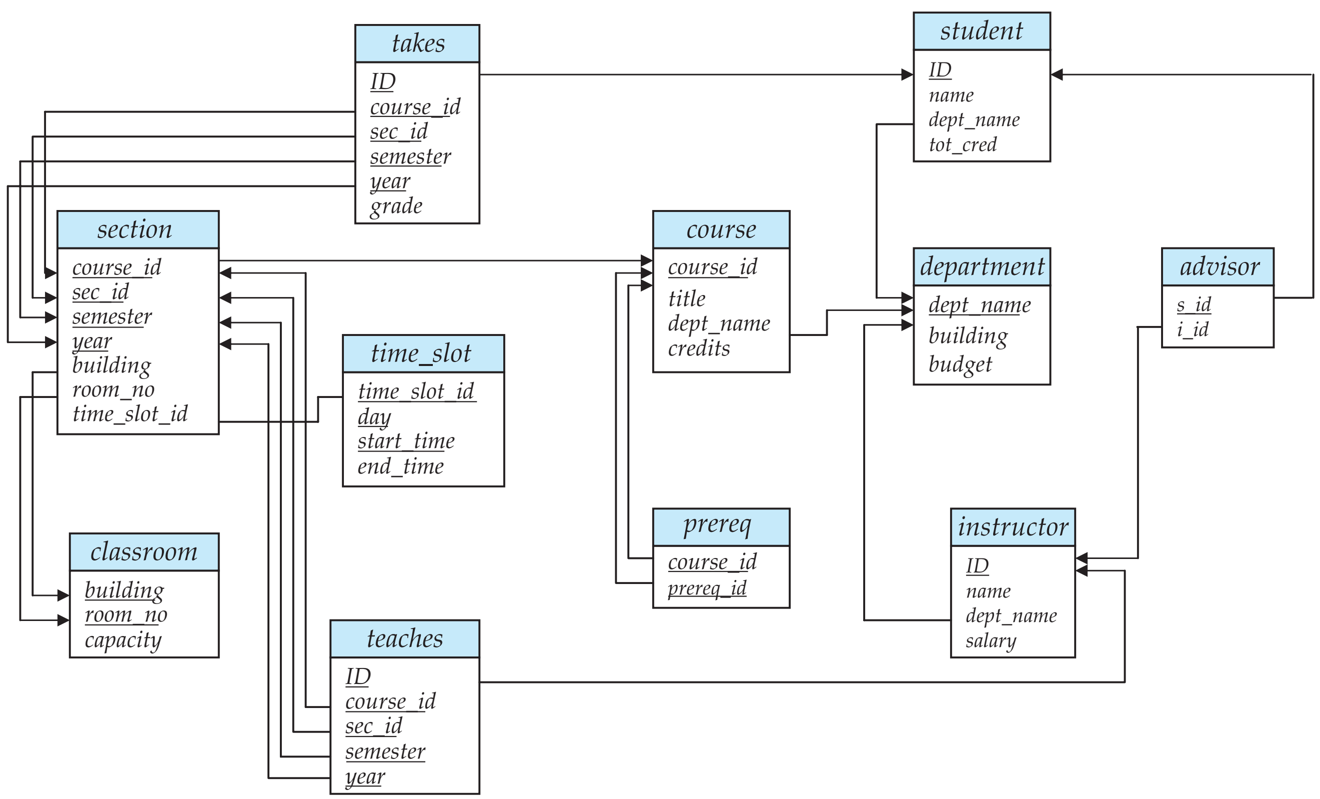


- The example query below can be written in a much simpler manner; this formulation is simply to illustrate SQL features.

- Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101:

```
SELECT COUNT (DISTINCT ID)
FROM  takes
WHERE     (course_id, sec_id, semester, year)
IN (SELECT course_id, sec_id, semester, year
    FROM  teaches
    WHERE teaches.ID = 10101);
```

SQL: Nested Subqueries: Comparisons over Sets Using SOME



- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department:

```
SELECT DISTINCT t1.name
FROM   instructor t1, instructor t2
WHERE  t1.salary > t2.salary
AND    t2.dept_name = 'Biology';
```

- Same query using a SOME clause:

```
SELECT t1.name
FROM   instructor t1
WHERE  t1.salary > SOME (SELECT DISTINCT t2.salary
                         FROM   instructor t2
                         WHERE  t2.dept_name = 'Biology');
```

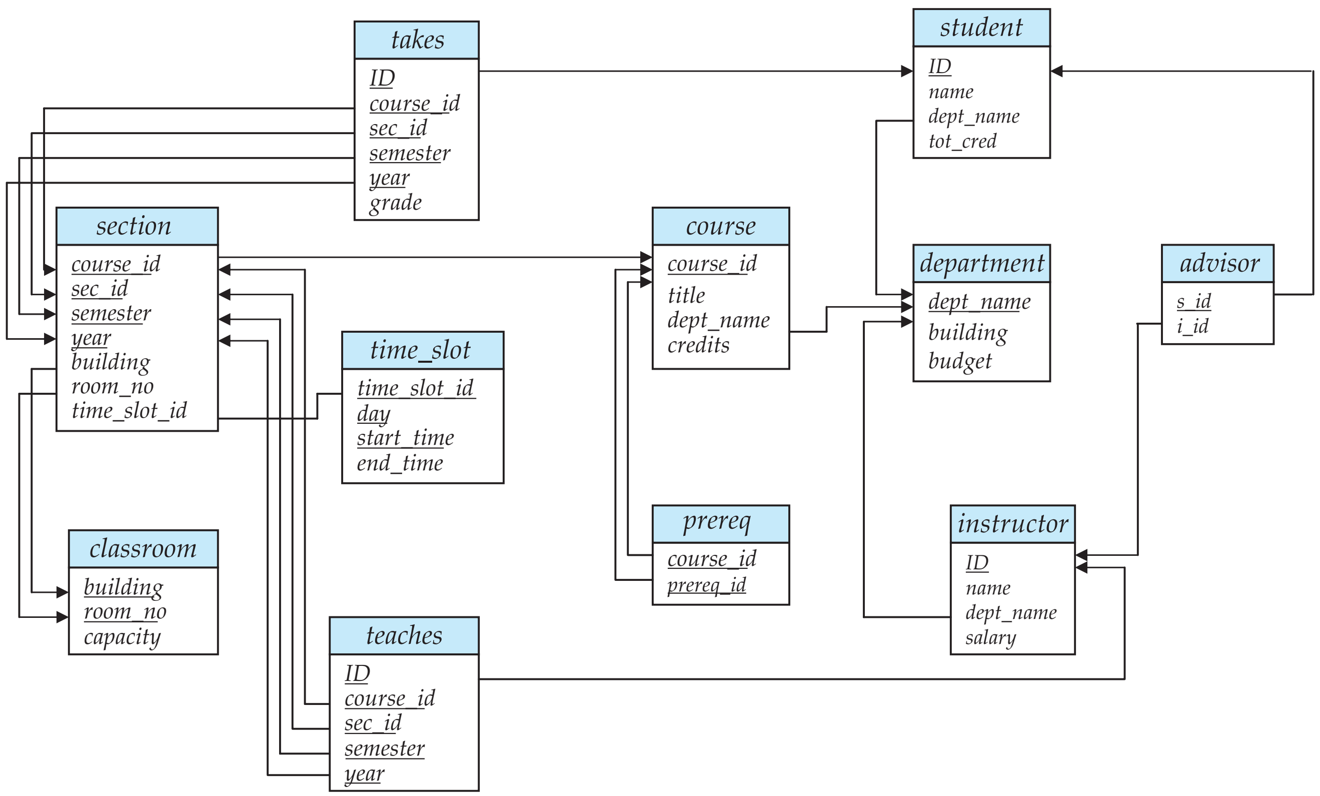
SQL: Nested Subqueries: Semantics of the SOME Clause

$e \, \phi$ SOME $r \equiv \exists \, t \in r$ s.t. $(e \, \phi \, t)$

where e is an expression that yields a value in the domain of r, t is a tuple in r and $\phi \in \{<, \leq, >, \geq, =, \neq\}$.

- So:
  - if e = 5 and r = (0, 5, 6), then
    'e < SOME r' evaluates to true
  - if e = 5 and r = (0, 5), then
    'e < SOME r' evaluates to false
  - if e = 5 and r = (0, 5), then
    'e = SOME r' evaluates to true
  - if e = 5 SOME r = (0, 5), then
    'e ≠ SOME r' evaluates to true
- Note that '= SOME' ≡ 'IN'
  but '≠ SOME' ≢ 'NOT IN'

SQL: Nested Subqueries: Comparisons over Sets Using ALL



- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department:

```
SELECT t1.name
FROM   instructor t1
WHERE  t1.salary > ALL (SELECT DISTINCT t2.salary
                        FROM   instructor t2
                        WHERE  t2.dept_name = 'Biology');
```

SQL: Nested Subqueries: Semantics of the ALL Clause

$e \;\phi$ ALL $r \equiv \exists\, t \in r$ s.t. $(e \;\phi\; t)$

where e is an expression that yields a value in the domain of r, t is a tuple in r and $\phi \in \{<, \leq, >, \geq, =, \neq\}$.

- So:
  - if e = 5 and r = (0, 5, 6), then 'e < ALL r' evaluates to false
  - if e = 5 and r = (6, 10), then 'e < ALL r' evaluates to true
  - if e = 5 and r = (4, 5), then 'e = ALL r' evaluates to false
  - if e = 5 ALL r = (4, 6), then 'e ≠ ALL r' evaluates to true
- Note that '= ALL' $\not\equiv$ 'IN'
  but '≠ ALL' $\equiv$ 'NOT IN'

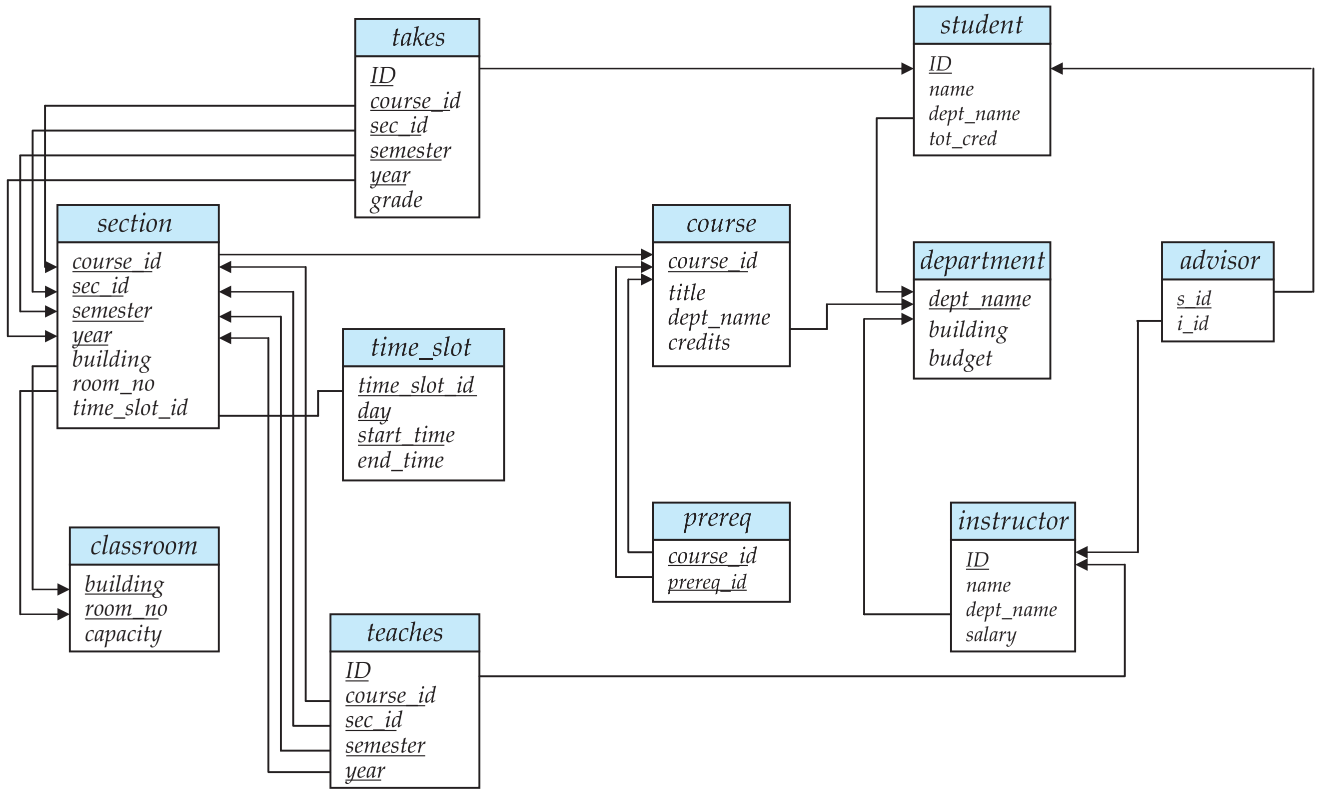
SQL: Nested Subqueries: Testing for the Empty Set Using the EXISTS Clause

- The EXISTS clause is used to return true if the argument subquery is nonempty and false otherwise:

  EXISTS $r \equiv r \neq \emptyset$

  NOT EXISTS $r \equiv r = \emptyset$

SQL: Nested Subqueries: Testing for the Empty Set Using the EXISTS Clause

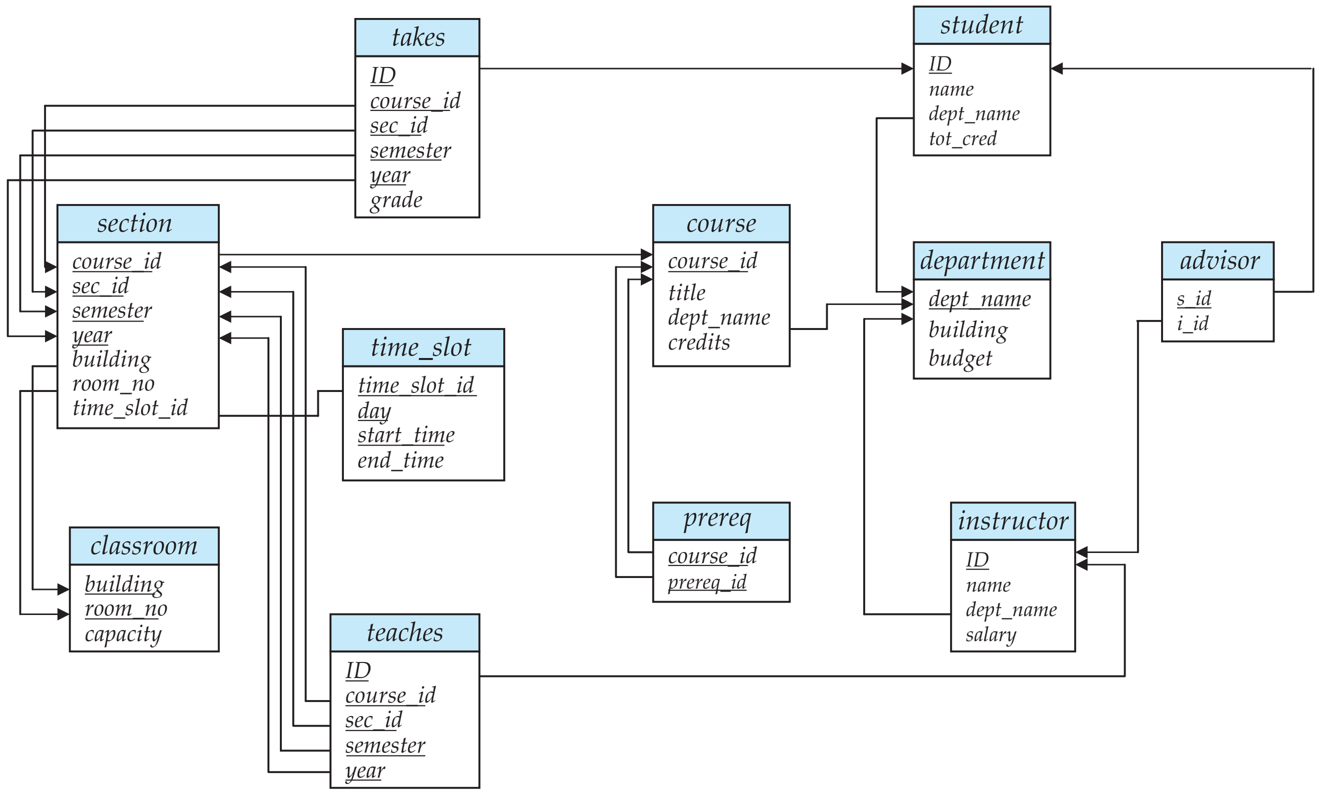


- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester":

```
SELECT s1.course_id
FROM   section s1
WHERE  s1.semester = 'Fall' AND s1.year = 2009
AND    EXISTS (SELECT *
               FROM  section s2
               WHERE s2.semester = 'Spring' AND year = 2010
               AND   s1.course_id = s2.course.id);
```

- This is an example of a **correlated subquery**, i.e., a subquery in which a range variable from the outer query (i.e., s1) is used in the inner query (it is compared with s2).

SQL: Nested Subqueries: Testing for the Empty Set Using the EXISTS Clause

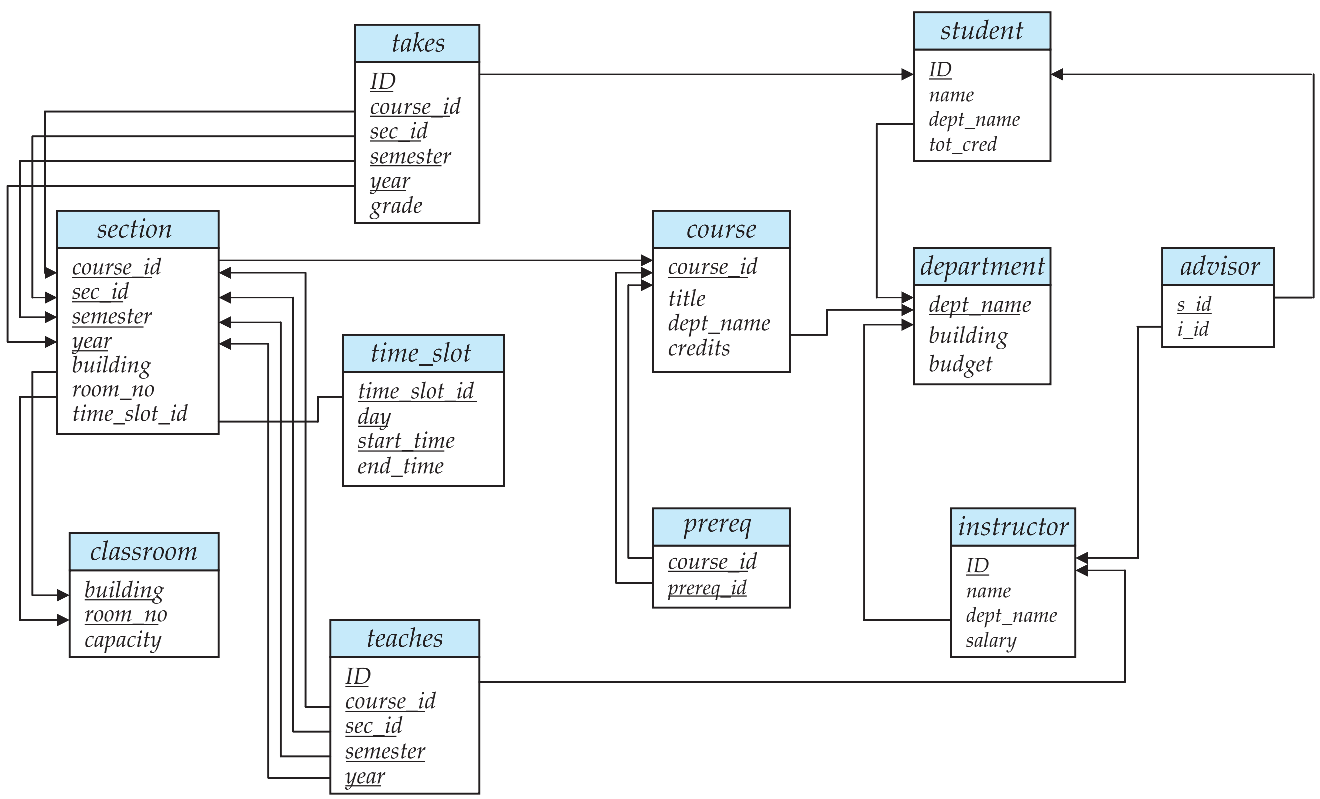


- Find all students who have taken all courses offered in the Biology department.

```
SELECT DISTINCT s.ID, s.name
FROM   student s
WHERE
NOT EXISTS ((SELECT course_id
             FROM  course
             WHERE dept_name = 'Biology')
    EXCEPT  (SELECT t.course_id
             FROM  takes t
             WHERE s.ID = t.ID));
```

- Note that $X \setminus Y = \emptyset \equiv X \subseteq Y$

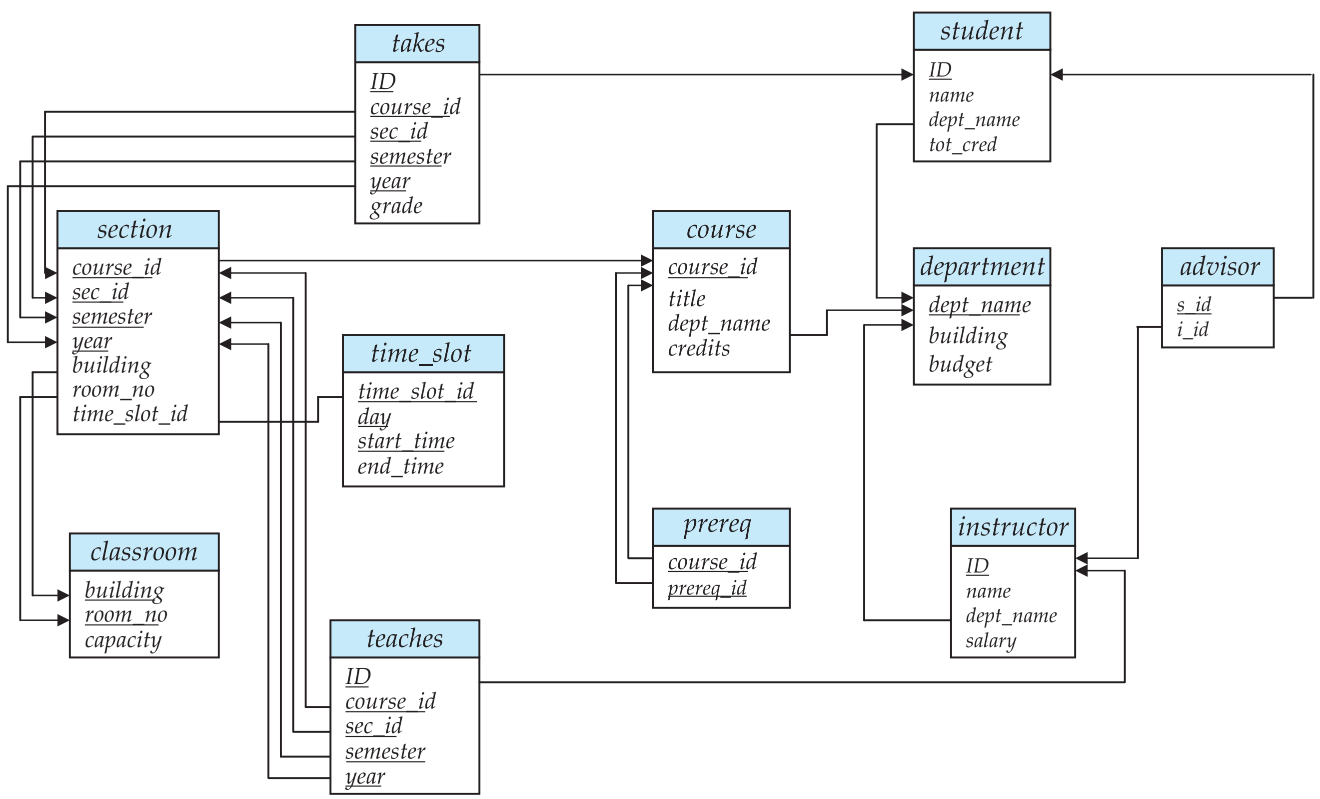
SQL: Nested Subqueries:
Testing for the Absence of Duplicates Using the UNIQUE Clause



- The UNIQUE clause is used to return true if the argument subquery does not return duplicate tuples and false otherwise.
- It returns true on the empty set.
- Find all courses that were offered at most once in 2009:

```
SELECT c.course_id
FROM   course c
WHERE  UNIQUE (SELECT s.course_id
               FROM  section s
               WHERE c.ID = s.ID AND s.year = 2009);
```

SQL: Nested Subqueries: Subqueries in the FROM Clause

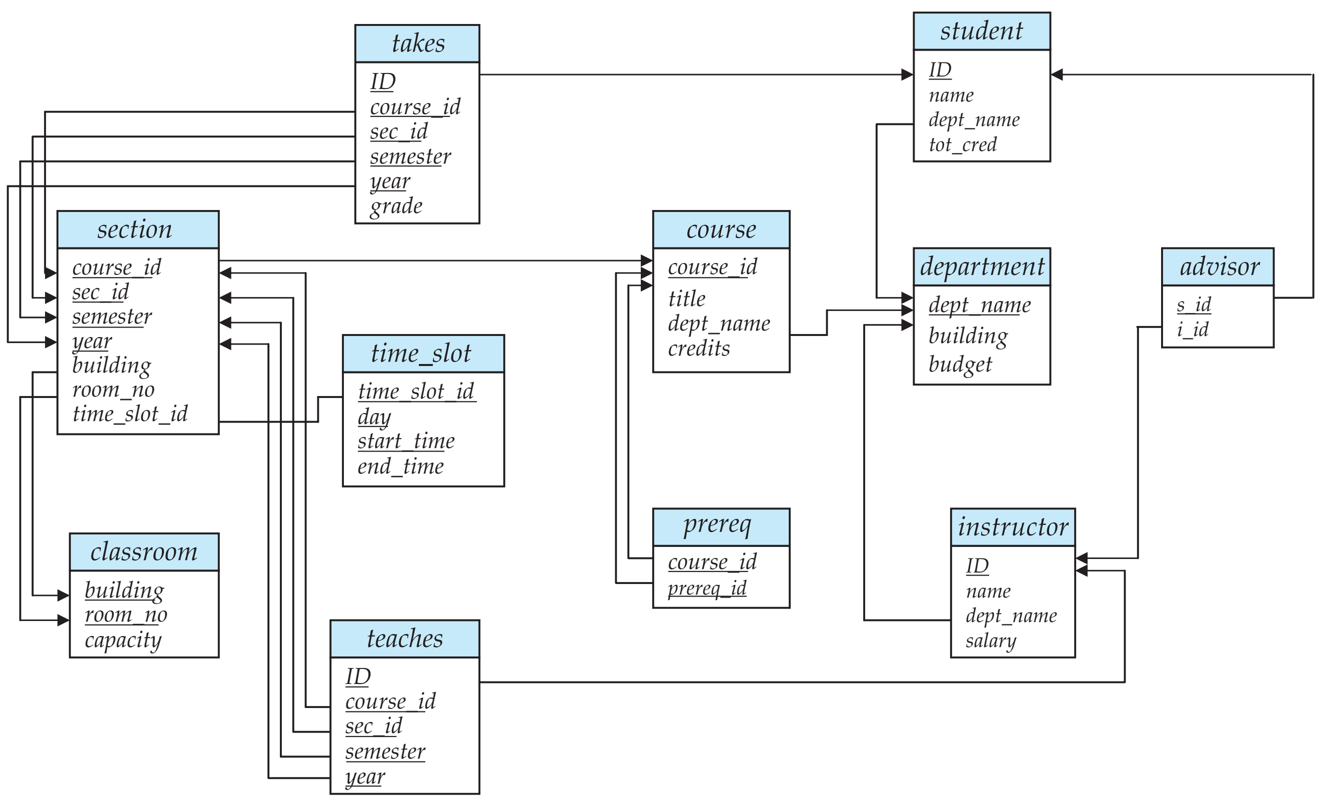


- SQL allows a subquery expression to be used in the FROM clause.
- Find the average instructor salary in those departments where the average salary is greater than $42,000:

```
SELECT dept_name, avg_salary
FROM   (SELECT dept_name, AVG(salary) AS avg_salary
        FROM   instructor
        GROUP BY dept_name)
WHERE  avg_salary > 42000;
```

- Note that this acts as a HAVING clause.
- Note also that avg_salary in the result is the renaming of the attribute resulting from computing AVG(salary) in the inner query.

SQL: Nested Subqueries: Subqueries in the FROM Clause

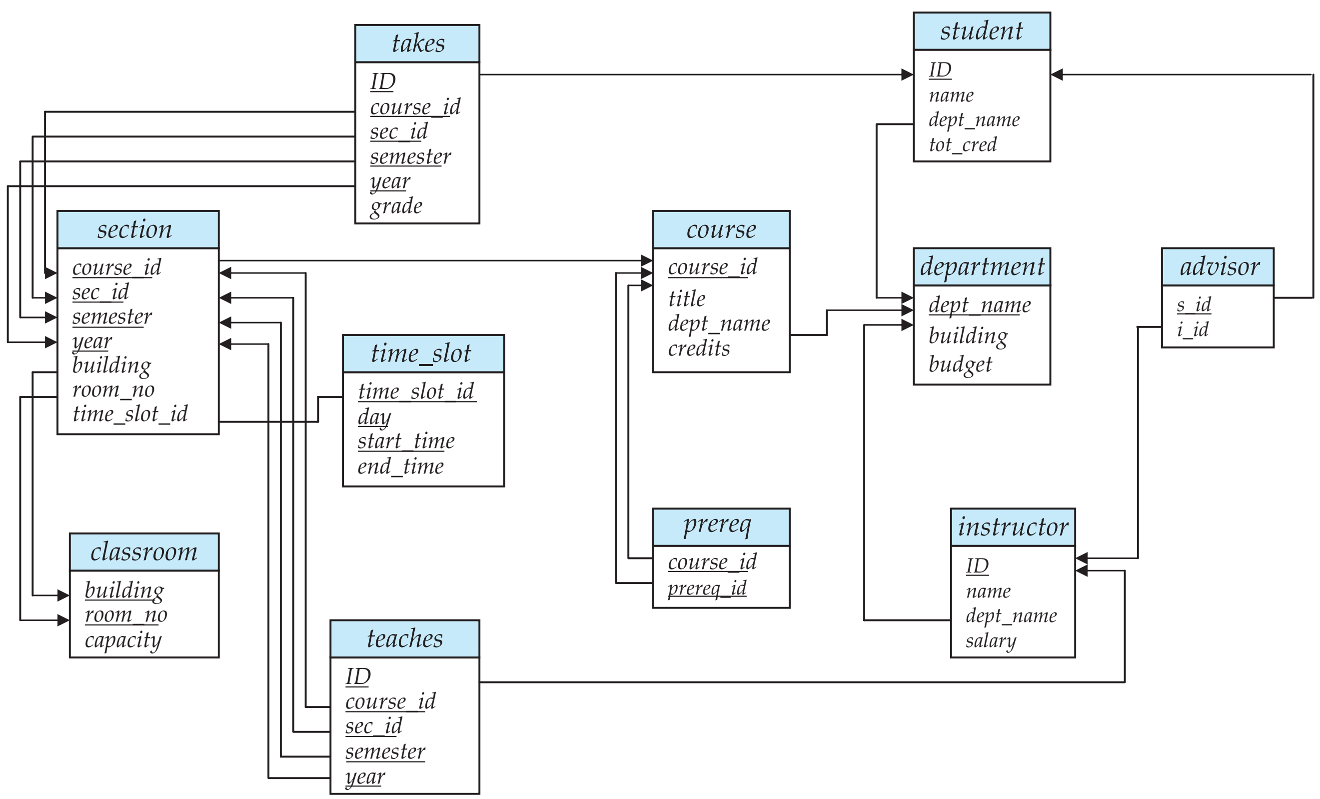


- Another way to write the previous query:

```
SELECT dept_name, avg_salary
FROM   (SELECT dept_name, AVG(salary)
        FROM   instructor
        GROUP BY dept_name)
        AS (dept_name, avg_salary)
WHERE  avg_salary > 42000;
```

- In the second case, the renaming of the attributes in the nested subquery is done using a tuple of attribute names.

SQL: Nested Subqueries: Temporary Views Using the WITH Clause



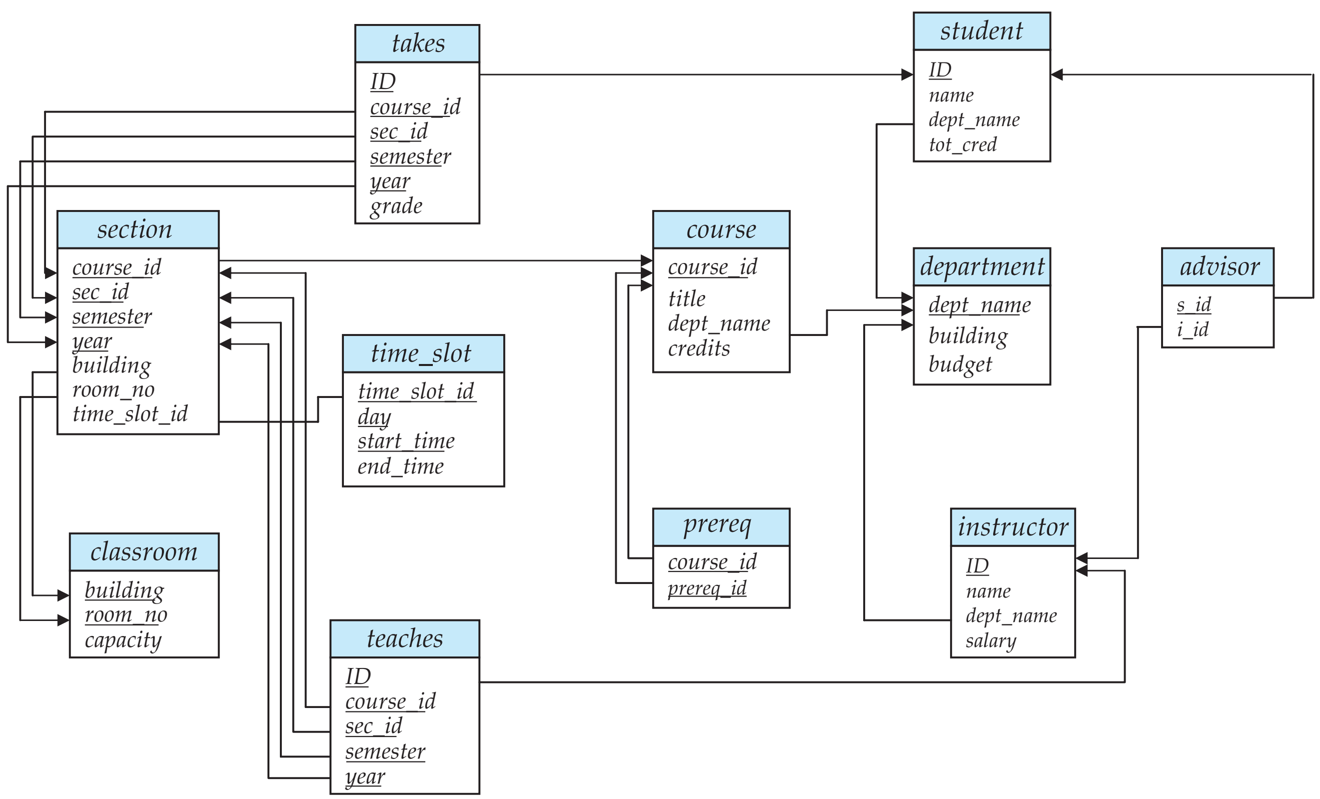
- The WITH clause provides a way of defining a temporary view whose definition is available only to the query in which the WITH clause occurs.

- Find all departments with the maximum budget:

```
WITH max_budget(value)
AS  (SELECT max(budget)
     FROM   department)
SELECT budget
FROM   department, max_budget
WHERE  department.budget = max_budget.value;
```

- Note that the WITH clause specifies a schema for the temporary relation, so that its name (i.e., max_budget) can appear in the FROM clause of the main query and its attribute (i.e., value) can be referred to in the WHERE clause of the main query.

SQL: Nested Subqueries: Temporary Views Using the WITH Clause



- The WITH clause is very useful for writing complex queries
- Supported by most database systems, with minor syntactic variations
- We can use two temporary views to find all departments where the total salary is greater than the average of the total salary at all departments:

```
WITH
  dept_total(dept_name, value)
AS  (SELECT dept_name, SUM(salary)
     FROM   instructor
     GROUP BY dept_name),
  dept_total_avg(value)
AS  (SELECT AVG(value)
     FROM   dept_total)
SELECT dept_name
FROM   dept_total, dept_total_avg
WHERE  dept_total.value >= dept_total_avg.value;
```

SQL: Nested Subqueries: Scalar Subqueries

- A **scalar subquery** is one that returns a single value and therefore can be used wherever a single value is expected.

- One example, in the SELECT clause:

```
SELECT dept_name,
    (SELECT COUNT(*)
     FROM   instructor
     WHERE  department.dept_name = instructor.dept_name) AS num_instructors
FROM   department;
```

- Another example, in the WHERE clause:

```
SELECT name
FROM   instructor
WHERE  salary*10 > (SELECT budget
                    FROM   department
                    WHERE  department.dept_name = instructor.dept_name);
```

- A runtime error is raised if the subquery returns more than one result tuple.

Summary

The Relational Languages

- An algebra is a mathematical structure made of a set of values/ variables and a set of operations on those values.

- A relational algebra (RA) has relations as values and operations on relations.

- The basic operations of RA are selection, projection, product, renaming, as well as union, intersection and difference.

- The most useful derived operations are various forms of join (e.g., inner, outer, semi- and antijoins).

- The most useful operations in an extended RA are extended projection, duplicate removal, sorting, grouping and aggregation.

- SQL has a direct translation to (extended) RA, which allows optimizers to manipulate RA expressions using equivalence laws.

- SQL is a powerful, practical language for interacting with relational DBMSs.

In The Next Lecture

- We'll begin our exploration of conceptual modelling.
- We'll see that conceptual modelling is aimed at providing an intuitive approach to identifying information needs.
- We'll observe that a good conceptual model facilitates the generation of well-formed logical models.
- In particular, we'll learn about the formalism for conceptual modelling known as Entity-Relationship (ER) Modelling.