Joins

Queries on Multiple Relations

Queries often need to access information from multiple relations.

Q :Retrieve the names of all instructors, along with their department names and department building name.

instructor(<u>id</u>, name, dept_name, salary)
department(<u>dept_name</u>, building, budget)

```
select name, instructor.dept_name, building
from instructor, department
where instructor.dept_name= department.dept_name;
```

We now consider the general case of SQL queries involving multiple relations. As we have seen earlier, an SQL query can contain three types of clauses, the select clause, the from clause, and the where clause. The role of each clause is as follows:

- The select clause is used to list the attributes desired in the result of a query.
- The from clause is a list of the relations to be accessed in the evaluation of the query.
- The where clause is a predicate involving attributes of the relation in the from clause.

A typical SQL query has the form

select $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
where P;

Each A_i represents an attribute, and each r_i a relation. P is a predicate. If the where clause is omitted, the predicate P is true.

Although the clauses must be written in the order select, from, where, the easiest way to understand the operations specified by the query is to consider the clauses in operational order: first from, then where, and then select.

The from clause by itself defines a Cartesian product of the relations listed in the clause. It is defined formally in terms of set theory, but is perhaps best understood as an iterative process that generates tuples for the result relation of the from clause.

```
for each tuple $t_1$ in relation $r_1$
    for each tuple $t_2$ in relation $r_2$
        . . .
        for each tuple $t_m$ in relation $r_m$
            Concatenate $t_1$, $t_2$, . . . , $t_m$ into a single tuple $t$
            Add $t$ into the result relation
```

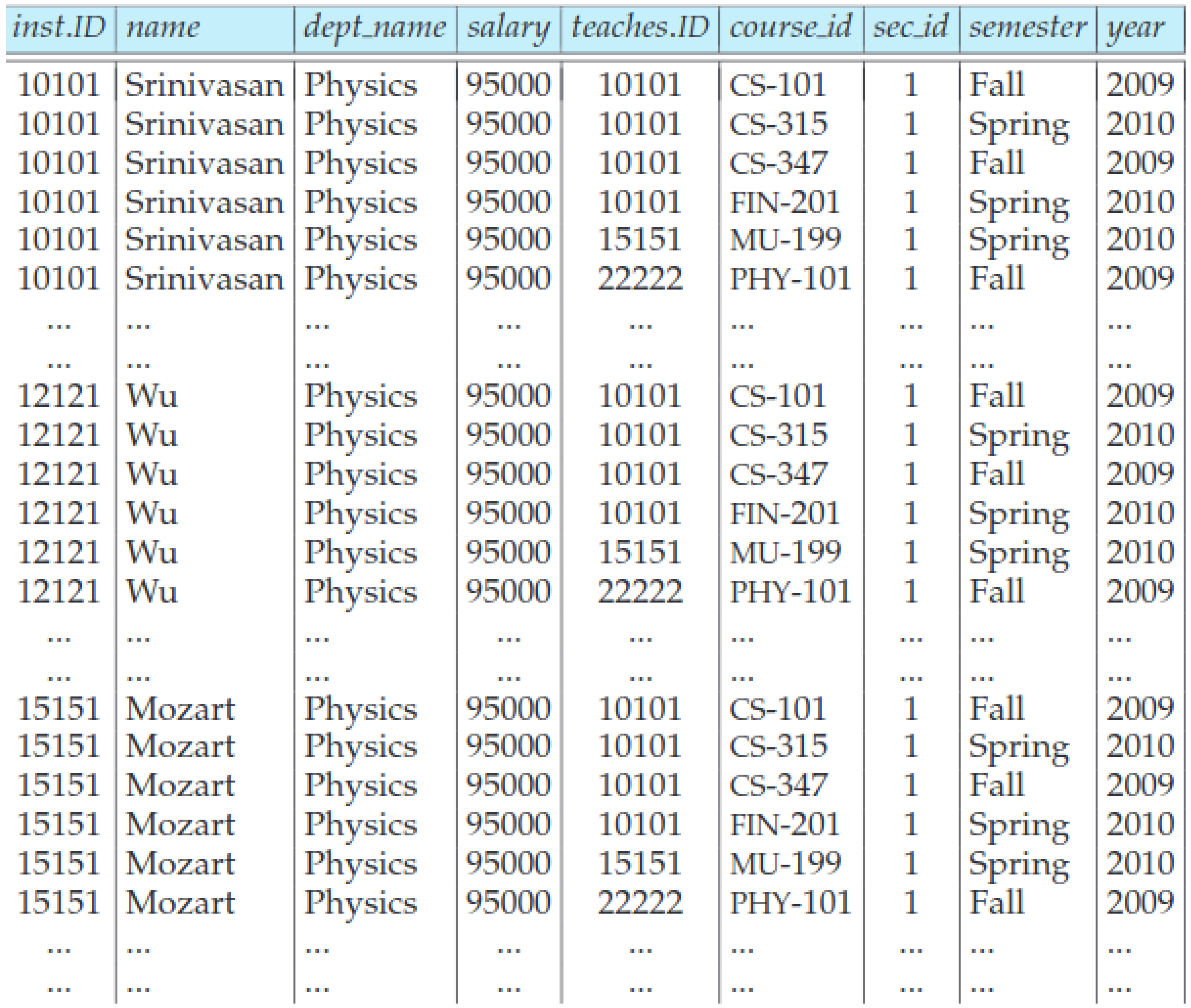
The result relation has all attributes from all the relations in the from clause. Since the same attribute name may appear in both r_i and r_j , as we saw earlier, we prefix the name of the relation from which the attribute originally came, before the attribute name.

For example, the relation schema for the Cartesian product of relations instructor and teaches is:
(instructor.ID, instructor.name, instructor.dept_name, instructor.salary
teaches.ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)

With this schema, we can distinguish instructor.ID from teaches.ID. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema as:

(instructor.ID, name, dept_name, salary
teaches.ID, course_id, sec_id, semester, year)

Their Cartesian product is shown in below figure, which includes only a portion of the tuples that make up the Cartesian product result.
The Cartesian product by itself combines tuples from instructor and teaches that are unrelated to each other. Each tuple in instructor is combined with every tuple in teaches, even those that refer to a different instructor. The result can be an extremely large relation, and it rarely makes sense to create such a Cartesian product.
Instead, the predicate in the where clause is used to restrict the combinations created by the Cartesian product to those that are meaningful for the desired answer. We would expect a query involving instructor and teaches to combine a particular tuple t in instructor with only those tuples in teaches that refer to the same instructor to which t refers. That is, we wish only to match teaches tuples with instructor tuples that have the same ID value.



inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Physics	95000	10101	CS-101	1	Fall	2009
12121	Wu	Physics	95000	10101	CS-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Physics	95000	10101	CS-101	1	Fall	2009
15151	Mozart	Physics	95000	10101	CS-315	1	Spring	2010
15151	Mozart	Physics	95000	10101	CS-347	1	Fall	2009
15151	Mozart	Physics	95000	10101	FIN-201	1	Spring	2010
15151	Mozart	Physics	95000	15151	MU-199	1	Spring	2010
15151	Mozart	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

The following SQL query ensures this condition, and outputs the instructor name and course identifiers from such matching tuples.

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;
```

Note that the above query outputs only instructors who have taught some course.
If we only wished to find instructor names and course identifiers for instructors in the Computer Science department, we could add an extra predicate to the where clause, as shown below.

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID and instructor.dept_name = ‘Comp. Sci.’;
```

In general, the meaning of an SQL query can be understood as follows:

1. Generate a Cartesian product of the relations listed in the from clause
2. Apply the predicates specified in the where clause on the result of Step 1.
3. For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the select clause.

The above sequence of steps helps make clear what the result of an SQL query should be, not how it should be executed. A real implementation of SQL would not execute the query in this fashion; it would instead optimize evaluation by generating (as far as possible) only elements of the Cartesian product that satisfy the where clause predicates.

When writing queries, you should be careful to include appropriate where clause conditions. If you omit the where clause condition in the preceding SQL query, it would output the Cartesian product, which could be a huge relation. For the instructor relation and the teaches relation, their Cartesian product has 12 ∗ 13 = 156 tuples. To make matters worse, suppose we have a more realistic number of instructors, say 200 instructors. Let's assume each instructor teaches 3 courses, so we have 600 tuples in the teaches relation. Then the above iterative process generates 200 ∗ 600 = 120,000 tuples in the result.

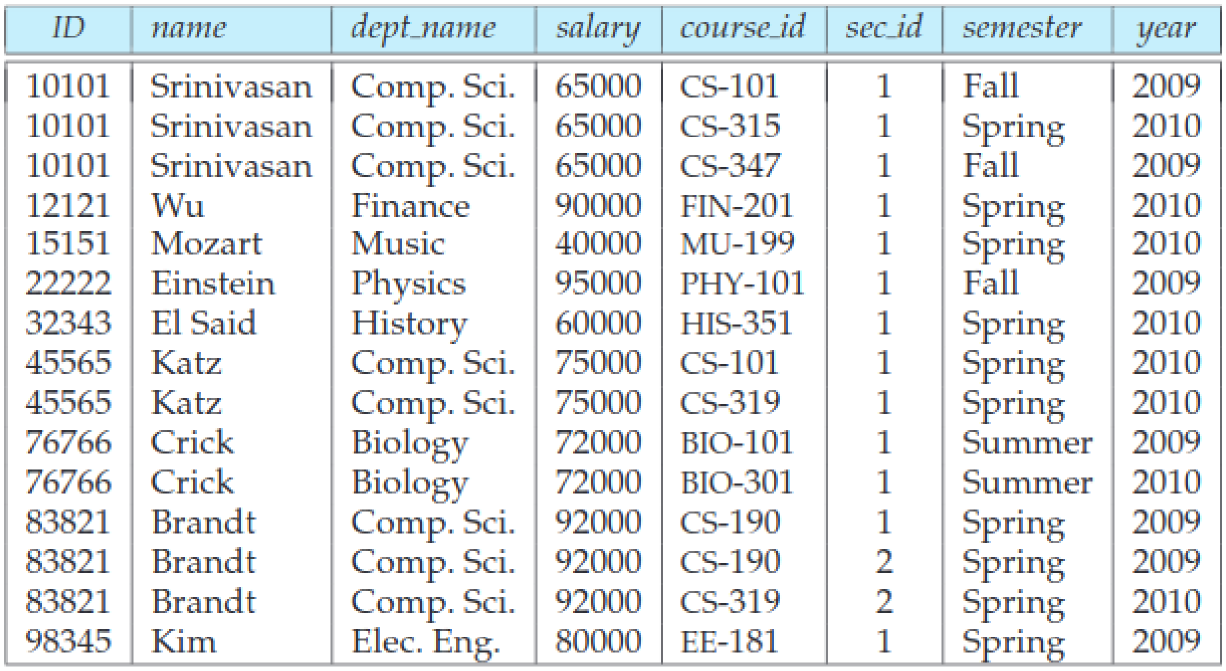
The Natural Join

In our example query that combined information from the instructor and teaches table, the matching condition required instructor.ID to be equal to teaches.ID. These are the only attributes in the two relations that have the same name. In fact this is a common case; that is, the matching condition in the from clause most often requires all attributes with matching names to be equated.

(instructor.ID, name, dept_name, salary
teaches.ID, course_id, sec_id, semester, year)

To make the life of an SQL programmer easier for this common case, SQL supports an operation called the **natural join**. In fact SQL supports several other ways in which information from two or more relations can be joined together. We have already seen how a Cartesian product along with a where clause predicate can be used to join information from multiple relations.
The **natural join** operation operates on two relations and produces a relation as the result. Unlike the Cartesian product of two relations, which concatenates each tuple of the first relation with every tuple of the second, **natural join** considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both

relations. So, going back to the example of the relations instructor and teaches, computing instructor **natural join** teaches considers only those pairs of tuples where both the tuple from instructor and the tuple from teaches have the same value on the common attribute, ID.
The result relation, shown in below figure, has only 13 tuples, the ones that give information about an instructor and a course that that instructor actually teaches. Notice that we do not repeat those attributes that appear in the schemas of both relations; rather they appear only once. Notice also the order in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.



ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Consider the query ‘For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught’, which we wrote earlier as:

```
select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;
```

This query can be written more concisely using the natural-join operation in SQL as:

```
select name, course_id
from instructor natural join teaches;
```

Both of the above queries generate the same result.

instructor(id, name, dept_name, salary)
teaches(id, course_id, sec_id, semester, year)
course(course_id, title, dept_name, credits)

Q: List the names of instructors along with the titles of courses that they teach.
The query can be written in SQL as follows:

```
select name, title
from instructor natural join teaches, course
where teaches.course_id= course.course_id;
```

The natural join of instructor and teaches is first computed, as we saw earlier, and a Cartesian product of this result with course is computed, from which the where clause extracts only those tuples where the course identifier from the join result matches the course identifier from the course relation. Note that teaches.course_id in the where clause refers to the course_id field of the natural join result, since this field in turn came from the teaches relation.
In contrast the following SQL query does not compute the same result:

```
select name, title
from instructor natural join teaches natural join course;
```

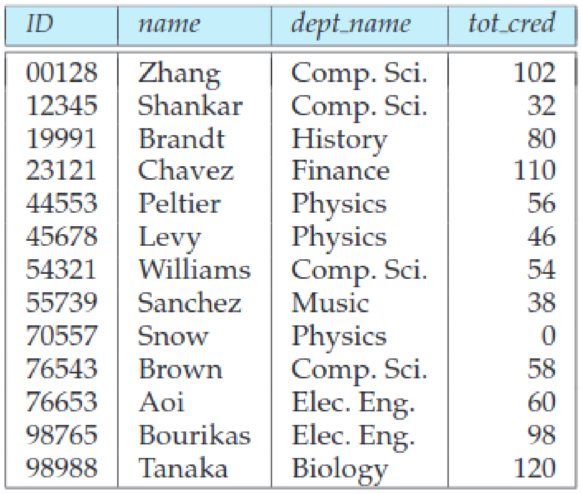
To see why, note that the natural join of instructor and teaches contains the attributes (ID, name, dept_name, salary, course_id, sec_id), while the course relation contains the attributes (course_id, title, dept_name, credits). As a result, the natural join of these two would require that the dept_name attribute values from the two inputs be the same, in addition to requiring that the course_id values be the same. This query would then omit all (instructor name, course title) pairs where the instructor teaches a course in a department other than the instructor's own department. The previous query, on the other hand, correctly outputs such pairs.
To provide the benefit of natural join while avoiding the danger of equating attributes erroneously, SQL provides a form of the natural join construct that allows you to specify exactly which columns should be equated.
This feature is illustrated by the following query:

```
select name, title
from (instructor natural join teaches) join course using (course_id);
```

The operation join . . . using requires a list of attribute names to be specified. Both inputs must have attributes with the specified names.

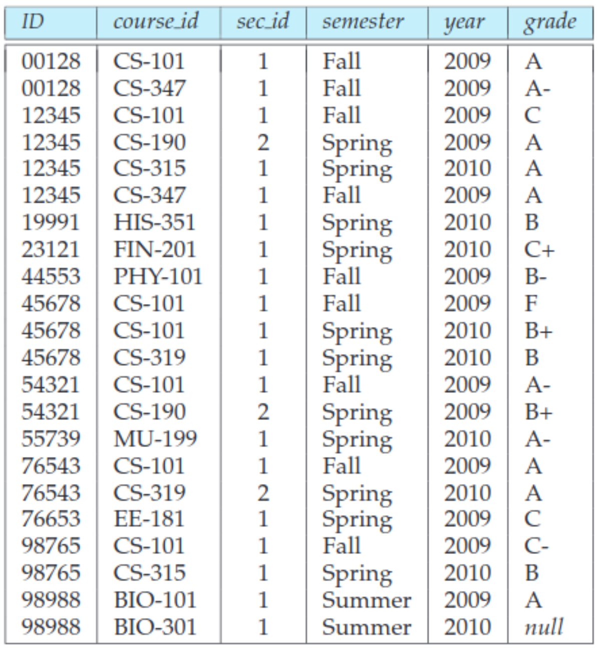
Consider the operation r_1 join r_2 using(A_1, A_2). The operation is similar to r_1 natural join r_2 , except that a pair of tuples t_1 from r_1 and t_2 from r_2 match if $t_1.A_1 = t_2.A_1$ and $t_1.A_2 = t_2.A_2$; even if r_1 and r_2 both have an attribute named A_3 , it is not required that $t_1.A_3 = t_2.A_3$. Thus, in the preceding SQL query, the join construct permits teaches.dept_name and course.dept_name to differ, and the SQL query gives the correct answer.

SQL also provides other forms of the join operation, including the ability to specify an explicit join predicate, and the ability to include in the result tuples that are excluded by natural join. We shall discuss these forms of join.
The examples in this section involve the two relations student and takes, shown in below figures respectively. Observe that the attribute grade has a value null for the student with ID 98988, for the course BIO-301, section 1, taken in Summer 2010. The null value indicates that the grade has not been awarded yet.



ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure 4.1 student relation



ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Figure 4.2 takes relation

Join Conditions

We saw how to express natural joins, and we saw the join . . .using clause, which is a form of natural join that only requires values to match on specified attributes. SQL supports another form of join, in which an arbitrary join condition can be specified.

The **on** condition allows a general predicate over the relations being joined. This predicate is written like a **where** clause predicate except for the use of the keyword **on** rather than **where**. Like the **using** condition, the **on** condition appears at the end of the join expression. Consider the following query, which has a join expression containing the **on** condition.

```
select *
from student join takes on student.ID= takes.ID;
```

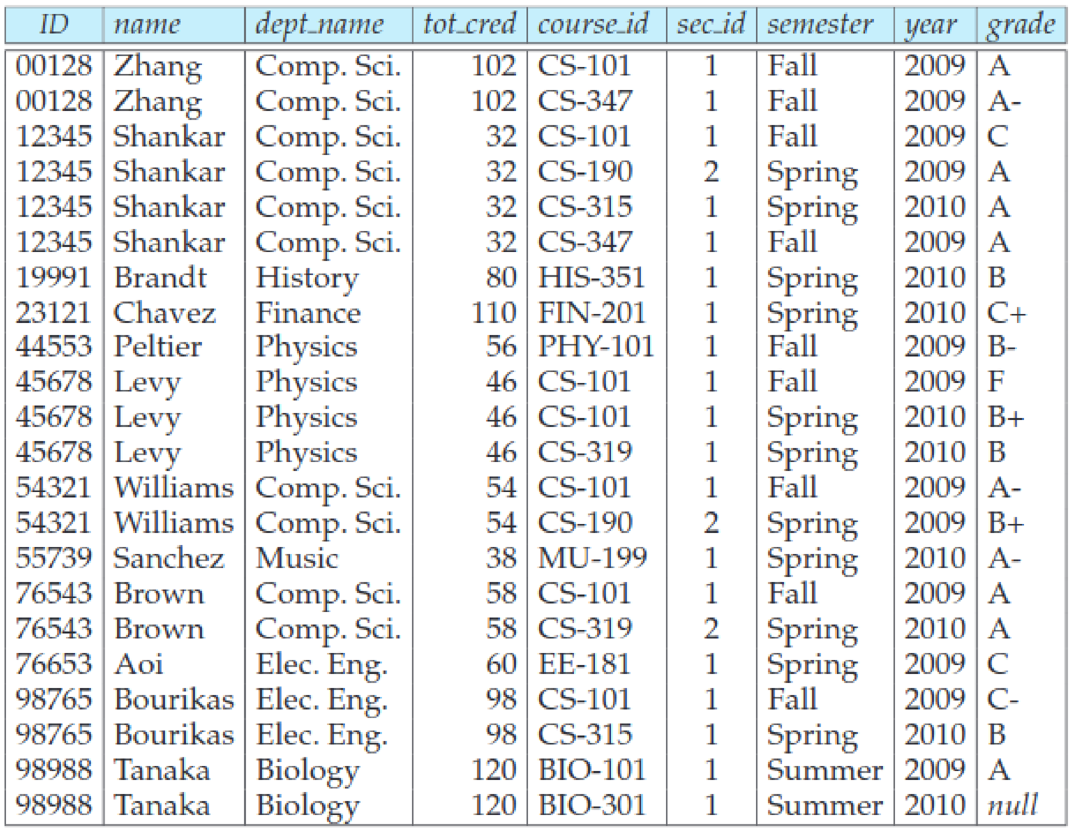
The **on** condition above specifies that a tuple from student matches a tuple from takes if their ID values are equal. The join expression in this case is almost the same as the join expression student **natural join** takes, since the natural join operation also requires that for a student tuple and a takes tuple to match. The one difference is that the result has the ID attribute listed twice, in the join result, once for student and once for takes, even though their ID values must be the same. In fact, the above query is equivalent to the following query (in other words, they generate exactly the same results):

```
select *
from student, takes
where student.ID= takes.ID;
```

As we have seen earlier, the relation name is used to disambiguate the attribute name ID, and thus the two occurrences can be referred to as student.ID and takes.ID respectively. A version of this query that displays the ID value only once is as follows:

```
select student.ID as ID, name, dept_name, tot_cred, course_id, sec_id, semester, year,
grade
from student join takes on student.ID= takes.ID;
```

The result of the above query is shown in the following Figure 4.3.



ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

Figure 4.3

The on condition can express any SQL predicate, and thus a join expression using the on condition can express a richer class of join conditions than natural join.

However, as illustrated by our preceding example, a query using a join expression with an on condition can be replaced by an equivalent expression without the on condition, with the predicate in the on clause moved to the where clause. Thus, it may appear that the on condition is a redundant feature of SQL.

However, there are two good reasons for introducing the on condition.

First, we shall see shortly that for a kind of join called an outer join, on conditions do behave in a manner different from where conditions.

Second, an SQL query is often more readable by humans if the join condition is specified in the on clause and the rest of the conditions appear in the where clause.

Outer Joins

Suppose we wish to display a list of all students, displaying their ID, and name, dept_name, and tot_cred, along with the courses that they have taken. The following SQL query may appear to retrieve the required information:

```
select *
from student natural join takes;
```

Unfortunately, the above query does not work quite as intended. Suppose that there is some student who takes no courses. Then the tuple in the student relation for that particular student would not satisfy the condition of a natural join with any tuple in the takes relation, and that student's data would not appear in the result. We would thus not see any information about students who have not taken any courses. For example, in the student and takes relations of Figures 4.1 and 4.2, note that student Snow, with ID 70557, has not taken any courses. Snow appears in student, but Snow's ID number does not appear in the ID column of takes. Thus, Snow does not appear in the result of the natural join.

More generally, some tuples in either or both of the relations being joined may be "lost" in this way. The outer join operation works in a manner similar to the join operations we have already studied, but preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.
For example, to ensure that the student named Snow from our earlier example appears in the result, a tuple could be added to the join result with all attributes from the student relation set to the corresponding values for the student Snow, and all the remaining attributes which come from the takes relation, namely course_id, sec_id, semester, and year, set to null. Thus the tuple for the student Snow is preserved in the result of the outer join.

There are in fact three forms of outer join:

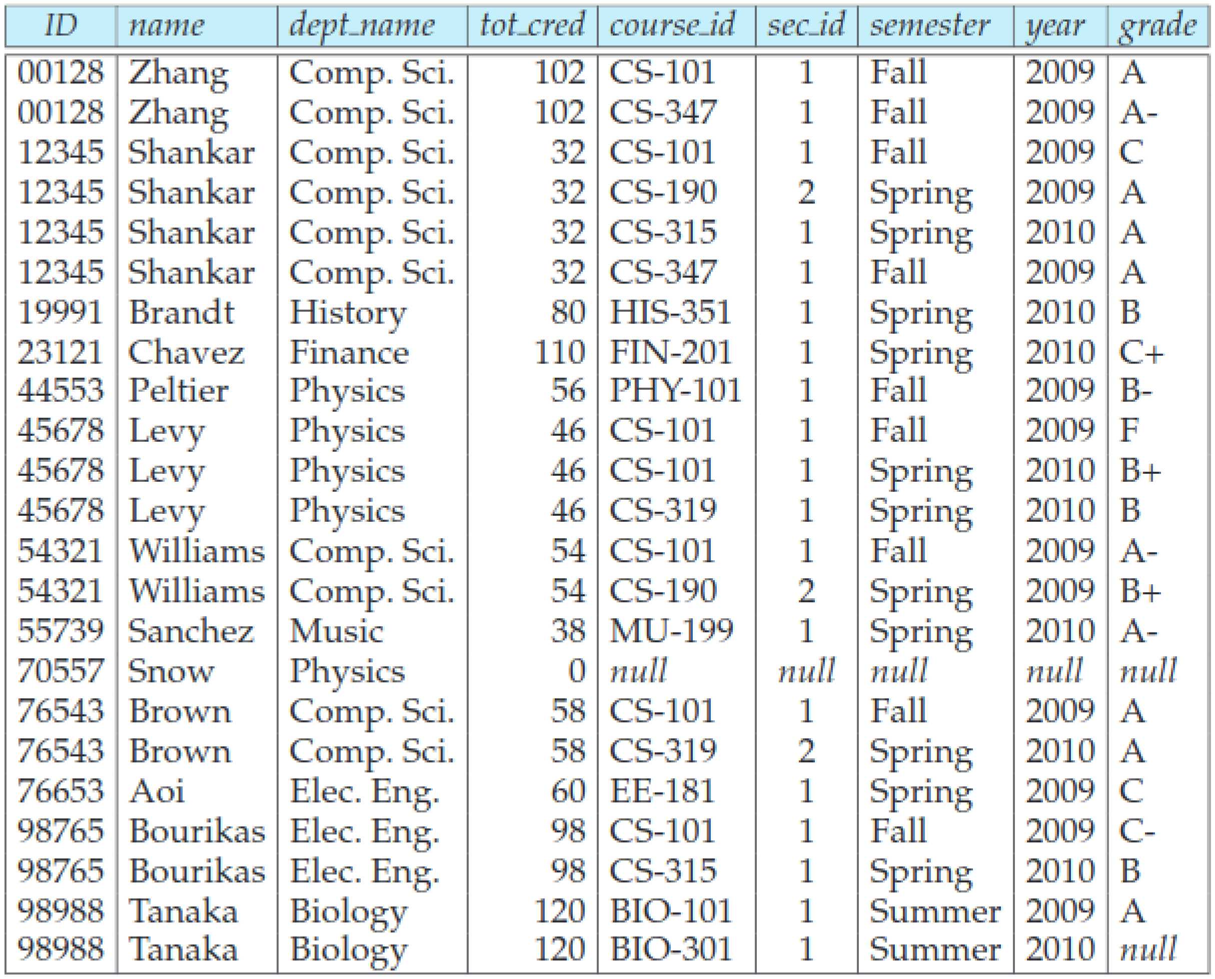
- The **left outer join** preserves tuples only in the relation named before (to the left of) the left outer join operation.
- The **right outer join** preserves tuples only in the relation named after (to the right of) the right outer join operation.
- The **full outer join** preserves tuples in both relations.

In contrast, the join operations we studied earlier that do not preserve non-matched tuples are called **inner join** operations, to distinguish them from the **outer-join** operations.
We now explain exactly how each form of outer join operates. We can compute the left outer-join operation as follows.
First, compute the result of the inner join as before. Then, for every tuple t in the left-hand-side relation that does not match any tuple in the right-hand-side relation in the inner join, add a tuple r to the result of the join constructed as follows:

- The attributes of tuple r that are derived from the left-hand-side relation are filled in with the values from tuple t.

- The remaining attributes of r are filled with null values.

Figure 4.4 shows the result of:



ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

Figure 4.4 Result of *student* **natural left outer join** *takes*.

```
select *
from student natural left outer join takes;
```

That result includes student Snow (ID 70557), unlike the result of an inner join, but the tuple for Snow includes nulls for the attributes that appear only in the schema of the takes relation.

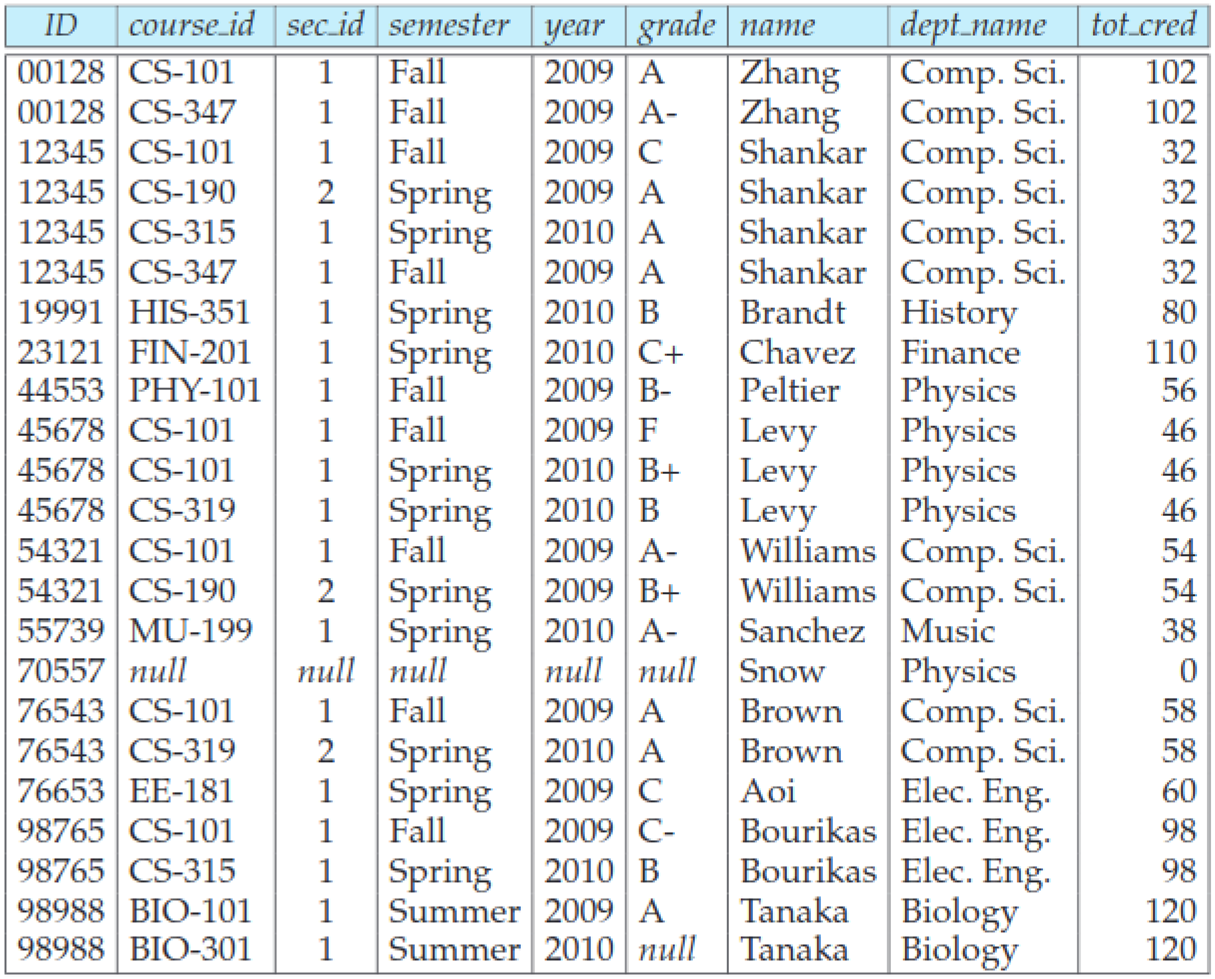
As another example of the use of the outer-join operation, we can write the query 'Find all students who have not taken a course' as:

```
select ID
from student natural left outer join takes
where course_id is null;
```

The right outer join is symmetric to the left outer join. Tuples from the right-hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join. Thus, if we rewrite our above query using a right outer join and swapping the order in which we list the relations as follows:

select *
from takes natural right outer join student;

we get the same result except for the order in which the attributes appear in the result (see Figure 4.5).



ID	*course_id*	*sec_id*	*semester*	*year*	*grade*	*name*	*dept_name*	*tot_cred*
00128	CS-101	1	Fall	2009	A	Zhang	Comp. Sci.	102
00128	CS-347	1	Fall	2009	A-	Zhang	Comp. Sci.	102
12345	CS-101	1	Fall	2009	C	Shankar	Comp. Sci.	32
12345	CS-190	2	Spring	2009	A	Shankar	Comp. Sci.	32
12345	CS-315	1	Spring	2010	A	Shankar	Comp. Sci.	32
12345	CS-347	1	Fall	2009	A	Shankar	Comp. Sci.	32
19991	HIS-351	1	Spring	2010	B	Brandt	History	80
23121	FIN-201	1	Spring	2010	C+	Chavez	Finance	110
44553	PHY-101	1	Fall	2009	B-	Peltier	Physics	56
45678	CS-101	1	Fall	2009	F	Levy	Physics	46
45678	CS-101	1	Spring	2010	B+	Levy	Physics	46
45678	CS-319	1	Spring	2010	B	Levy	Physics	46
54321	CS-101	1	Fall	2009	A-	Williams	Comp. Sci.	54
54321	CS-190	2	Spring	2009	B+	Williams	Comp. Sci.	54
55739	MU-199	1	Spring	2010	A-	Sanchez	Music	38
70557	*null*	*null*	*null*	*null*	*null*	Snow	Physics	0
76543	CS-101	1	Fall	2009	A	Brown	Comp. Sci.	58
76543	CS-319	2	Spring	2010	A	Brown	Comp. Sci.	58
76653	EE-181	1	Spring	2009	C	Aoi	Elec. Eng.	60
98765	CS-101	1	Fall	2009	C-	Bourikas	Elec. Eng.	98
98765	CS-315	1	Spring	2010	B	Bourikas	Elec. Eng.	98
98988	BIO-101	1	Summer	2009	A	Tanaka	Biology	120
98988	BIO-301	1	Summer	2010	*null*	Tanaka	Biology	120

Figure 4.5 The result of *takes* **natural right outer join** *student*.

The full outer join is a combination of the left and right outer-join types. After the operation computes the result of the inner join, it extends with nulls those tuples from the left-hand-side relation that did not match with any from the right-hand side relation, and adds them to the result. Similarly, it extends with nulls those tuples from the

right-hand-side relation that did not match with any tuples from the left-hand-side relation and adds them to the result.
As an example of the use of full outer join, consider the following query:
“Display a list of all students in the Comp. Sci. department, along with the course sections, if any, that they have taken in Spring 2009; all course sections from Spring 2009 must be displayed, even if no student from the Comp. Sci. department has taken the course section.” This query can be written as:

```
select *
from (select *
from student
where dept name= ’Comp. Sci’)
natural full outer join
(select *
from takes
where semester = ’Spring’ and year = 2009);
```

The on clause can be used with outer joins. The following query is identical to the first query we saw using “student natural left outer join takes,” except that the attribute ID appears twice in the result.

```
select *
from student left outer join takes on student.ID= takes.ID;
```

As we noted earlier, on and where behave differently for outer join. The reason for this is that outer join adds null-padded tuples only for those tuples that do not contribute to the result of the corresponding inner join. The on condition is part of the outer join specification, but a where clause is not.
In our example, the case of the student tuple for student “Snow” with ID 70557, illustrates this distinction. Suppose we modify the preceding query by moving the on clause predicate to the where clause, and instead using an on condition of true.

```
select *
from student left outer join takes on true
where student.ID= takes.ID;
```

The earlier query, using the left outer join with the on condition, includes a tuple (70557, Snow, Physics, 0, null, null, null, null, null, null), because there is no tuple in takes with ID = 70557. In the latter query, however, every tuple satisfies the join condition true, so no null-padded tuples are generated by the outer join. The outer join actually generates the Cartesian product of the two relations. Since there is no tuple in takes with ID = 70557, every time a tuple appears in the outer join with name = “Snow”, the values for student.ID and takes.ID must be different, and such tuples would be eliminated by the where clause predicate. Thus student Snow never appears in the result of the latter query.

Join Types and Conditions

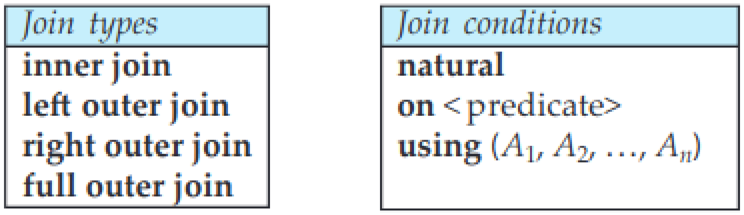
To distinguish normal joins from outer joins, normal joins are called inner joins in SQL. A join clause can thus specify inner join instead of outer join to specify that a normal join is to be used. The keyword inner is, however, optional. The default join type, when the join clause is used without the outer prefix is the inner join.
Thus,

```
select *
from student join takes using (ID);
```

is equivalent to:

```
select *
from student inner join takes using (ID);
```

Similarly, natural join is equivalent to natural inner join.
Figure 4.6 shows a full list of the various types of join that we have discussed. As can be seen from the figure, any form of join (inner, left outer, right outer, or full outer) can be combined with any join condition (natural, using, or on).



Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on < predicate>
using $(A_1, A_2, \ldots, A_n)$

Figure 4.6 Join types and join conditions.