# *Appendix C* Set-Comparison Queries in SQL

This appendix extends the coverage of SQL queries in Chapter 12 by discussing in some detail how queries involving set comparisons may be formulated in SQL. The discussion is based upon an earlier paper (Halpin 1989a), and assumes familiarity with the basic set theory covered in Section 6.2, as well as the following aspects of SQL: the group-by clause (Section 12.10), and correlated and existential subqueries (Section 12.11). Further aspects of SQL are discussed in later appendices.

A *set-comparison query* is a query that involves the comparison of two or more sets. For example: Who speaks at least all the official languages of Canada? Here we seek people whose set of languages contains the set of official languages of Canada. The techniques used here to formulate such queries involve the following three stages:

1. Picture the set-comparison using a Venn diagram
2. Translate this into a predicate calculus expression or a cardinality expression
3. Translate this expression into SQL

Let's begin by reviewing how Venn diagrams can be used to compare two sets. Figure C.1 shows the basic notation. The sets (here *A* and *B*) are drawn as named, overlapping circles or ellipses. The elements of both sets must belong to a common domain, shown as a surrounding rectangle.

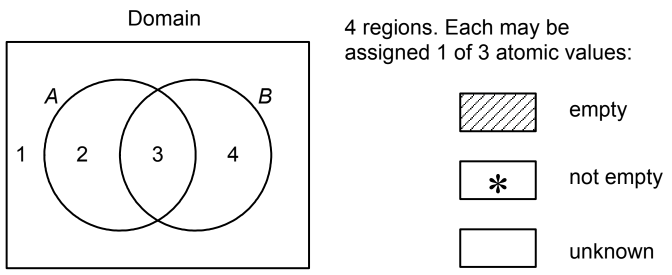


**Figure C.1** Venn diagram notation for comparing two sets.

There are four distinct regions, numbered here as 1..4. To indicate the relationships between the two sets and their domain, each of the four regions may be assigned one of three values: Empty, Not empty, or Unknown. An empty region is indicated by shading it (e.g., with a slash fill pattern). If something exists in a region, it is marked with an asterisk "*". If a region has no mark, this means it is unknown whether or not an element exists in that region. Thus standard Venn diagrams take the hypothetical viewpoint.

Since there are four regions, each of which may be assigned one of three values, there are $3^4$ or 81 possible atomic patterns for the 2-set case. There are also disjunctive possibilities. For example, an unadorned bar spanning regions indicates that something exists in at least one of those regions. A small circle at the end of a disjunction bar indicates that nothing exists in the region that contains that end. Figure C.2 shows two possible disjunctive patterns.

For database queries, the surrounding domain region may be ignored once we know that the sets are compatible. If we ignore the disjunctive patterns, we now have 27 atomic patterns for two sets. Of these, only the following seven cases are commonly encountered for comparing two sets *A* and *B*, as set out in Figure C.3.

If *A* and *B* are *disjoint* (mutually exclusive), then they have no elements in common. Hence their intersection is empty, as shown by shading. This may be expressed by the *predicate calculus* formula $\sim\exists x\ (x \in A\ \&\ x \in\ B)$. This is read "it is not the case that there exists an element *x* such that *x* belongs to *A* and *x* belongs to *B*". For any set *S*, the notation "#*S*" denotes the cardinality of *S* (i.e., the number of elements in *S*). If the intersection of the sets *A* and *B* is empty, then the number of elements in the intersection must equal zero. This may be expressed by the *cardinality formula* $\#(A \cap B) = 0$.

If *A* is a *subset* of *B* then each element of *A* is also in *B*. Hence the region in *A* that is not in *B* must be empty. The predicate calculus formula says that there does not exist an element in this region. It follows that the number of elements in the intersection of *A* and *B* must equal the number of elements in *A*.

The other five cases in Figure C.3 may be understood in a similar way. If A is a *proper subset* of B then it is a subset of *B*, and *B* contains at least one element not in *A*. The predicate calculus and cardinality formulae for this case have two conjuncts, one for the empty region (in *A* and not in *B*) and one for the populated region (in *B* and not in *A*). The *superset* cases are the inverses of the subset cases. *Identical* sets have exactly the same members, so all their elements are in their intersection. If sets *properly overlap* then they have an element in common, and each has an element not contained in the other.

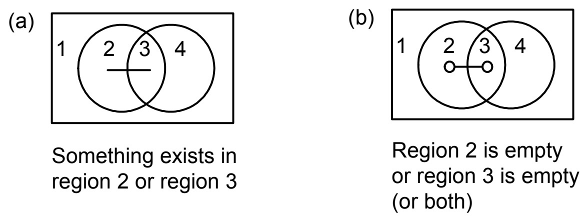

(a)
1
2
3
4
Something exists in region 2 or region 3
(b)
1
2
3
4
Region 2 is empty or region 3 is empty (or both)

**Figure C.2** Disjunction bars on Venn diagrams.

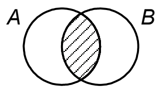
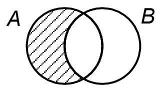
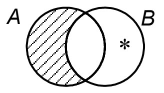
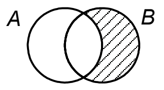
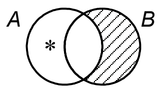
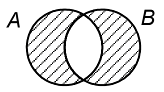
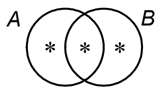
| | *Case* | *Venn diagram* | *Predicate calculus* | *Cardinality formula* |
|---|---|---|---|---|
| (a) | disjoint $A \cap B = \{\ \}$ |  | $\sim\exists x\,(x \in A\ \&\ x \in B)$ | $\#(A \cap B) = 0$ |
| (b) | subset $A \subseteq B$ |  | $\sim\exists x\,(x \in A\ \&\ x \notin B)$ | $\#(A \cap B) = \#A$ |
| (c) | proper subset $A \subset B$ |  | $\sim\exists x\,(x \in A\ \&\ x \notin B)$ & $\exists x\,(x \in B\ \&\ x \notin A)$ | $\#(A \cap B) = \#A$ $< \#B$ |
| (d) | superset $A \supseteq B$ |  | $\sim\exists x\,(x \in B\ \&\ x \notin A)$ | $\#(A \cap B) = \#B$ |
| (e) | proper superset $A \supset B$ |  | $\exists x\,(x \in A\ \&\ x \notin B)$ & $\sim\exists x\,(x \in B\ \&\ x \notin A)$ | $\#(A \cap B) = \#B$ $< \#A$ |
| (f) | identity $A = B$ |  | $\sim\exists x\,(x \in A\ \&\ x \notin B)$ & $\sim\exists x\,(x \in B\ \&\ x \notin A)$ | $\#(A \cap B) = \#A$ $= \#B$ |
| (g) | proper overlap |  | $\exists x\,(x \in A\ \&\ x \notin B)$ & $\exists x\,(x \in A\ \&\ x \in B)$ & $\exists x\,(x \notin A\ \&\ x \in B)$ | $\#(A \cap B) > 0$ $< \#A$ $< \#B$ |

**Figure C.3** Seven cases for comparing two sets.

Now that we know how to diagram and translate the seven cases into logical and cardinality formulae, let's consider how to translate the formulae into SQL. Since the predicate calculus expression involves existential quantifiers, its translation technique is known as the *existence technique*. Translation of the cardinality expressions involves use of SQL's group-by clause and is known as the *grouping technique*. We'll discuss these techniques in turn.

Consider a UoD concerning people who speak one or more languages. To store who speaks what language we use the table scheme *Speaks*( person, "language" ). Since "language" is a reserved word in SQL, we delimit its name with double-quotes. Table C.1 shows a sample population. For simplicity, this assumes that people may be identified by their first name. Now consider the query:

Q1: Who speaks *at least* those language(s) spoken by Eve?

To express this query in SQL, the most commonly used approach (e.g., Date 2000) is to use nested, negative existential subqueries, as shown in S1.

**Table C.1** Relational table storing facts about who speaks what language.

***Speaks:***

| *person* | *"language"* |
|---|---|
| Ann | English |
| Bill | English |
| Chris | English |
| David | English |
| David | Japanese |
| Eve | English |
| Eve | Japanese |
| Fred | English |
| Fred | French |
| Fred | Japanese |
| Gina | Italian |
| Gina | Japanese |
| Helen | Greek |

S1:

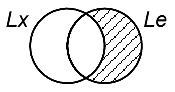
```
select distinct person
from Speaks as X
where not exists
    (select * from Speaks as Y
      where person = 'Eve'
          and not exists
           (select * from Speaks
             where person = X.person
                 and "language" = Y."language"))
```

Here the nesting of one negated subquery inside another tends to make the query difficult to understand. With more complex queries of this nature, this approach becomes even less comprehensible. An alternative, systematic approach to the formulation of such queries is provided by the existence technique. The key to this approach is to realize that the query is based on a comparison between sets. The set comparison may then be depicted on a Venn diagram, expressed in predicate calculus and finally mapped to SQL using the translation shown in Table C.2.

**Table C.2** Correspondence between operators in predicate calculus and SQL.

| *Predicate calculus* | *SQL* |
|---|---|
| $\exists$ | **exists** |
| $\in$ | **in** |
| $\notin$ | **not in** |
| ~ | **not** |
| & | **and** |
| $\vee$ | **or** |

Let's see how this works with our current example. Query Q1 may be recast as a set expression thus: List each person *x* where the set of languages spoken by *x* is a superset of the languages spoken by Eve. This may be diagrammed and formalized as shown in Figure C.4.



$Lx$ = Languages spoken by $x$
$Le$ = Languages spoken by Eve

$Lx \supseteq Le$ $\sim\exists l\ (l \in Le \ \&\ l \notin Lx)$

**Figure C.4** Venn diagram and logical formula for the set comparison underlying Q1.

Here *Lx* denotes the set of languages spoken by some person *x*, and *Le* is the set of languages spoken by Eve. We use *l* as a variable ranging over languages, and *x* as variable ranging over persons. On the Venn diagram, we shade the region to indicate that nothing exists there (i.e., Eve does not speak a language not spoken by *x*). This is translated in predicate calculus as:

$\sim\exists l\ (l \in Le \ \&\ l \notin Lx)$

In logic we can use a simple domain variable *x* ranging over people. In SQL however, we use a tuple variable *X* ranging over rows from the Speaks table. To specify the domain variable in SQL we then use the qualified column name *X*.person. So to satisfy the syntax of SQL, we replace "*x*" in our logical formula with "*X*.person". Although SQL is not case-sensitive, we'll use upper case for tuple variables to help distinguish them from the lower case variables used in logic. Using the logic-to-SQL conversion from Table C.2, the query may be formulated in SQL as shown in S1':

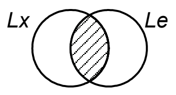
S1':
```
select distinct person
from Speaks as X
where not exists                -- language spoken by Eve and not by x
    (select * from Speaks
     where person = 'Eve'
       and "language" not in           -- languages spoken by x
           (select "language" from Speaks
            where person = X.person))
```

To clarify the query, the argument of each **exists** quantifier is described in a comment immediately after the quantifier. If you like, you may replace "*" in the query with "language" or a constant (e.g., 1). For the sample population, the query returns the answer set {'David', 'Eve', 'Fred'}. The use of the **distinct** qualifier ensures the result is a set rather than a bag. Note also the correlation condition "person = *X*.person". With this technique, the mapping from logic to SQL is straightforward, so the only challenging aspect of such queries is specifying the original set comparison.

As another example, consider the following query, where the set comparison is one of disjointness (mutual exclusion):

Q2: Who speaks *none* of the languages spoken by Eve?

The query may be formalized as shown in Figure C.5. The shading on the Venn diagram indicates that the intersection of the sets is empty. The logical formula says that nothing exists in this region, which belongs to both sets. The SQL query may now be easily formulated as shown in S2. For the sample population, this returns the result set {'Helen'}.



*Lx* = Languages spoken by *x*
*Le* = Languages spoken by Eve

$Lx \cap Le = \{\ \}$ $\quad\quad$ $\sim\exists l\ (l \in Lx\ \&\ l \in Le)$

**Figure C.5** Formalization of the set comparison underlying query Q2.

```
S2:   select distinct person
      from Speaks as X
      where not exists          -- language spoken by x and Eve
         (select * from Speaks
          where person = X.person
            and "language" in              -- languages spoken by Eve
                (select "language" from Speaks
                 where person = 'Eve'))
```

We now consider the grouping technique. As shown in the final column of Figure C.3 , the seven cases of set comparisons can be translated into cardinality comparisons. The six cases (b)–(g) can be mapped directly to SQL by using a group-by clause to form a group corresponding to the set intersection. This grouping technique can't be used with case (a), since disjoint sets have an empty intersection, and SQL ignores empty groups. So if the sets are disjoint, we should use the existence technique. For the other cases, the grouping technique is normally preferred, since on most DBMSs the SQL query it generates will execute faster than the equivalent SQL query generated by the existence technique.

To illustrate the grouping technique, we'll use the table scheme *Eats*( person, food ) to store facts about who eats what foods. Table C.3 shows a sample population.

**Table C.3** Relational table storing facts about who eats what foods.

***Eats***:

| *person* | *food* |
|---|---|
| Ann | apple |
| Ann | beef |
| Ann | potato |
| Bill | apple |
| Bill | potato |
| Chris | apple |
| Chris | potato |
| Fred | peas |
| Humphrey | apple |
| Humphrey | beef |
| Humphrey | chicken |
| Humphrey | orange |
| Humphrey | peas |
| Humphrey | potato |
| Sue | apple |
| Sue | chicken |
| Sue | orange |
| Sue | peas |

To help understand the grouping technique, consider first the following simple cardinality queries.

Q3: Who eats *exactly* two foods?
Q4: Who eats *at most* three foods?
Q5: Who eats *at least* four foods?

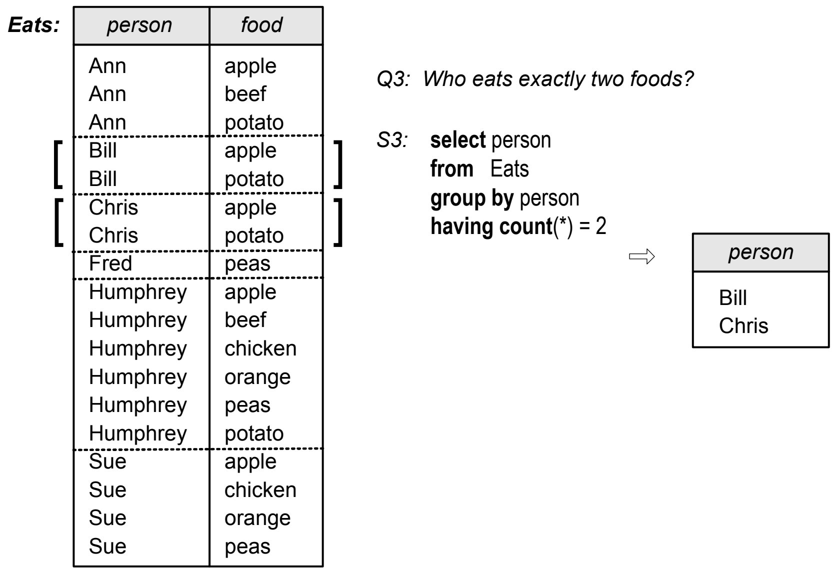
Each of these involves a comparison with a fixed number. These queries may be formulated using a group-by clause applied to the Eats table to form one group for each person. Because of the uniqueness constraint spanning the (person, food) tuples, each food appears at most once in each group. Hence the number of rows in each group equals the number of foods eaten by that person. The result may now be computed by comparing this count with the number of foods mentioned in the query.

For example, query Q3 may be formulated as the SQL query S3 shown in Figure C.6. Here the groups in the table are separated by dotted lines, and the groups selected by the query are highlighted with square brackets. we've used square brackets here as a reminder that SQL groups in general are bags of tuples (recall that we use square brackets to delimit bags). In this case however the bags are sets, since the uniqueness constraint ensures that no tuples are duplicated in a group.

Similarly, Queries Q4 and Q5 may be formulated in SQL by S4 and S5 as shown:

```
S4:   select person from Eats
      group by person
      having count(*) <= 3      → { 'Ann', 'Bill', 'Chris', 'Fred' }

S5:   select person from Eats
      group by person
      having count(*) >= 4      → { 'Humphrey', 'Sue' }
```



**Eats:**

| person | food |
|---|---|
| Ann | apple |
| Ann | beef |
| Ann | potato |
| Bill | apple |
| Bill | potato |
| Chris | apple |
| Chris | potato |
| Fred | peas |
| Humphrey | apple |
| Humphrey | beef |
| Humphrey | chicken |
| Humphrey | orange |
| Humphrey | peas |
| Humphrey | potato |
| Sue | apple |
| Sue | chicken |
| Sue | orange |
| Sue | peas |

*Q3: Who eats exactly two foods?*

```
S3:   select person
      from   Eats
      group by person
      having count(*) = 2
```

⇨

| person |
|---|
| Bill |
| Chris |

**Figure C.6** The group-by clause forms one group for each person.

Notice that both our table schemes *Speaks*( person, “language” ) and *Eats*( person, food ) have the form *R*( *a*, *b* ), which itself is a special case of *R*( *a*, *b*, … ) where the columns *a* and *b* are spanned by a uniqueness constraint, and other columns not involved in this constraint may exist. Generalizing from our earlier discussion, and using Θ to denote a numeric comparator (=, <, >, <=, >=, <>), any query of the form

Which instances of *a* bear the relationship Θ to *n* instances of *b* in *R*( *a*, *b*, … )?

may be formulated as:

```
select a
from  R
group by a
having count(*) Θ  n
```

The following cardinality query goes beyond this pattern by requiring a subquery to compute the value of *n*.

Q6: Who eats *at least as many* foods as Ann?

To formulate this in SQL, we use a subquery to compute the number of foods that Ann eats (in our sample population this is 3), and then apply the usual approach, as shown in S6.

S6:
```
select person
from   Eats
group by person
having count(*) >=
        (select count(*)
         from    Eats
         where person = 'Ann')→  { 'Ann', 'Humphrey', 'Sue' }
```

Now consider the following query:

Q7: Who eats *at least all* the foods that Ann eats?

This goes beyond Q6 since its condition involves a set comparison, not just a number comparison. If the set comparison in Q7 is satisfied, the number comparison in Q6 is satisfied too, but the converse does not hold. Using our grouping technique, however, we can translate the set comparison into an equivalent number comparison, as in Figure C.7.

This query involves the superset case, just as query Q1 did, except this time we are expressing the superset condition as a cardinality equation. Since the region of *Fa* that is not in *Fx* is empty, it follows that the foods eaten by both *x* and Ann are the same as Ann’s foods. So the number of foods that *x* and Ann have in common equals the number of foods that Ann eats. Hence, #($Fx \cap Fa$) = #*Fa*.

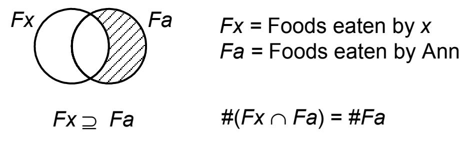


**Figure C.7** Venn diagram and cardinality equation for the set comparison in Q7.

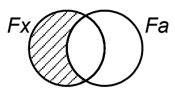
This query may now be readily formulated in SQL as shown in S7. The first count computes #($Fx \cap Fa$) and the second count computes #$Fa$.

```
S7:   select  person              -- person x
      from    Eats
      where food in               -- consider only those foods of x that are in Ann's foods
         (select  food from Eats
          where person = 'Ann')
      group by person
      having count(*) =           -- #common_foods = #Ann's_foods
         (select count(*) from Eats
          where person = 'Ann')         →  { 'Ann', 'Humphrey' }
```

Now consider the following query:

Q8: Who eats *at most* those foods eaten by Ann?

This involves the subset condition $Fx \subseteq Fa$, so can be diagrammed as shown in Figure C.8.



$Fx$ = Foods eaten by $x$
$Fa$ = Foods eaten by Ann

$Fx \subseteq Fa$ $\qquad$ $\#(Fx \cap Fa) = \#Fx$

**Figure C.8** Venn diagram and cardinality equation for query Q8.

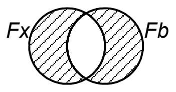
Using the grouping technique, this leads to the SQL formulation shown in S8. In this case, the subquery count concerns the person x, so we introduce a tuple variable *X* in the outer from-clause to enable the correlation back from the subquery to the person in the outer query.

```
S8:   select person from Eats as X -- person x
      where food in               -- Ann's foods
         (select food from Eats
          where person = 'Ann')
      group by person
      having count(*) =           --  #common_foods = #x_foods
         (select count(*) from Eats
          where person = X.person)    →  { 'Ann', 'Bill', 'Chris' }
```

Now consider the following query.

Q9: Who eats *exactly the same* (all and only those) foods eaten by Bill?

This involves a set identity condition, and may be formalized as in Figure C.9.



$Fx$ = Foods eaten by $x$
$Fb$ = Foods eaten by Bill

$Fx = Fb$ $\qquad$ $\#(Fx \cap Fb) = \#Fb$
$\qquad\qquad\qquad\qquad = \#Fx$

**Figure C.9** Person x eats exactly the same foods as Bill.

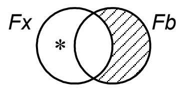
The SQL formulation of this query is shown in S9.

```
S9:    select person from Eats as X
       where food in                    -- Bill's foods
            (select food from Eats
             where person = 'Bill')
       group by person
       having count(*) =                -- #common_foods = #Bill's_foods
            (select count(*) from Eats
             where person = 'Bill')
           and count(*) =                 -- #common_foods = #x's_foods
            (select count(*) from Eats
             where person = X.person)    → { 'Ann', 'Bill', 'Chris' }
```

The grouping technique is more efficient than the existence technique for establishing equality between sets, since the intersection count is computed only once and then compared to both the other counts. This same advantage is enjoyed for the proper subset, proper superset and proper overlap cases. For example, consider the following query:

Q10: Who eats all Bill's foods and more besides?

This involves a proper superset comparison that may be formalized as in Figure C.10. The SQL query is the same as S9, except that the final "=" is replaced by "<".



*Fx* = Foods eaten by *x*
*Fb* = Foods eaten by Bill

$Fx \supseteq Fb$

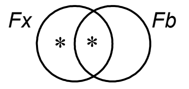
$\#(Fx \cap Fb) = \#Fb$
$< \#Fx$

**Figure C.10** A proper superset example.

As discussed earlier, the grouping technique cannot handle the disjoint sets case (e.g., Who has no foods in common with Fred?). However it can be applied to many cases other than the six (b)-(g) cases in Figure C.3. As a simple example involving two sets, consider:

Q11: Who has some foods in common with Bill and some different?

The set comparison underlying this query is formalized in Figure C.11. This is not the same as proper overlap, since it leaves the question open as to whether Bill eats a food not eaten by *x*. For the sample data, the query result is {'Ann', 'Humphrey', 'Sue'}. Each of these three people has at least one food in common with Bill. Bill does not eat any food not eaten by Ann or Humphrey, but Bill does eat a food not eaten by Sue.



*Fx* = Foods eaten by *x*
*Fb* = Foods eaten by Bill

$\#(Fx \cap Fb) > 0$
$< \#Fx$

**Figure C.11** Person *x* has some foods in common with Bill, and some different.

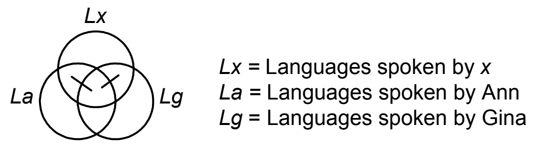
This query may be formulated in SQL as shown in S11. Since SQL ignores empty groups, the only groups retained will have members, so there is no need to add any explicit code for the condition $\#(Fx \cap Fb) > 0$.

S11:
```
select person from Eats as X
where food in                   -- Bill's foods
     (select food from Eats
      where person = 'Bill')
group by person
having count(*) <               -- #common_foods < #x_foods
     (select count(*) from Eats
      where person = X.person)     → {'Ann', 'Humphrey', 'Sue'}
```

The existence and grouping techniques may be extended to cater for set comparison cases involving more than two sets, as well as cases without spanning uniqueness constraints. As an example with disjunction bars and three sets, consider the following query applied to our earlier table scheme *Speaks*( person, "language" ).

Q12: Who speaks some of Ann's and some of Gina's languages?

For cases where Ann and Gina have no common language, this query finds each person who can perform the necessary language translations to allow Ann and Gina to communicate. The formalization for the existence technique is shown in Figure C.12. The translation into SQL is straightforward and is left as an exercise.


Lx
La
Lg
Lx = Languages spoken by x
La = Languages spoken by Ann
Lg = Languages spoken by Gina

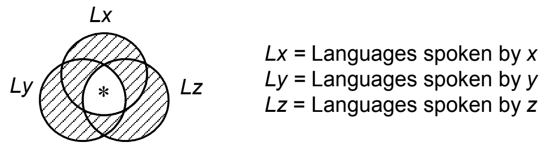
$\exists l\ (l \in Lx\ \&\ l \in La)\ \&\ \exists l\ (l \in Lx\ \&\ l \in Lg)$

**Figure C.12** A case involving disjunction bars with three sets.

As a more complex case involving three set variables, consider query Q13.

Q13: List all *triples* whose members speak *exactly the same* languages.

This means that the three people in each triple must speak exactly the same languages. It is possible that different triples speak different languages. This may be formalized for the grouping technique as shown in Figure C.13.


Lx
Ly
Lz
*
Lx = Languages spoken by x
Ly = Languages spoken by y
Lz = Languages spoken by z

$\#(Lx \cap Ly \cap Lz) = \#Lx = \#Ly = \#Lz$

**Figure C.13** For each triple ($x$, $y$, $z$), the members speaks exactly the same languages.

The translation to SQL is shown in S13. The condition "*X*.person < *Y*.person **and** *Y*.person < *Z*.person" ensures that each group is comprised of three different people, and that each group is listed in one order only.

```
S13:  select X.person, Y.person, Z.person
      from   Speaks as X, Speaks as Y, Speaks as Z
      where X.person < Y.person and Y.person < Z.person
         and  X."language" = Y."language"
         and  Y."language" = Z."language"
      group by X.person, Y.person, Z.person
      having count(*) = (select count(*) from Speaks
                          where person = X.person)
          and count(*) = (select count(*) from Speaks
                          where person = Y.person)
          and count(*) = (select count(*) from Speaks
                          where person = Z.person)
```

The techniques discussed work with *a-b* (sub)relations with no nulls or duplicate pairs. If nulls are allowed, the query must be carefully understood, and relevant "**is not null**" conditions applied.

If duplicates are allowed, no change is needed to the existence technique, but the grouping technique requires **count**(**distinct** …) to avoid counting duplicates. Some SQL dialects restrict the number of times the **distinct** option may be used with a single query. As a final example, suppose the relation scheme *UsesOn*( person, "language", project ) is used to store facts about who uses what programming language on what project. A sample population is shown in Table C.4.

**Table C.4** A relational table listing who uses what languages on what projects.

***UsesOn*:**

| *person* | *language* | *project* |
|---|---|---|
| Ann | C# | Alpha |
| Ann | C# | Omega |
| Ann | SQL | Omega |
| Bill | C# | Omega |
| Chris | Visual BASIC | Beta |
| Chris | C# | Beta |
| Chris | SQL | Omega |

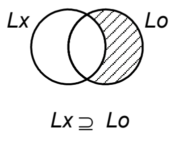
Since this three-column relation is all-key, the following derived bag projections allow duplicates:

*Uses*[ person, "language" ]
*UsedOn*[ "language", project ]
*WorksOn*[ person, project ]

Now consider the following query.

Q14: Who uses *at least those* languages used on project Omega?

The superset condition in this query may be formalized for the grouping technique as shown in Figure C.14.



*Lx* = Languages used by *x*
*Lo* = Languages used on project Omega

#(*Lx* ∩ *Lo*) = #*Lo*

**Figure C.14** Person *x* uses at least those languages used on project Omega.

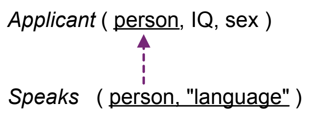
The query may be mapped to SQL as shown in S14. Notice the use of "**distinct**" to convert bags to sets.

S14:
```
select person from UsesOn
where "language" in                  -- languages used on Omega
     (select "language" from UsesOn
      where project = 'Omega')
group by person
having count (distinct "language") =  -- #languages used on Omega
     (select count(distinct "language")
      from UsesOn
      where project = 'Omega')         → {'Ann', 'Chris'}
```

With these new techniques under your belt, you must be keen to try them out (at least we hope so!). The following exercise gives you an opportunity.

### *Exercise C1*

1. The following relational schema is designed to store data about applicants for positions as language interpreters.



A sample population is shown on the following page. Formulate the following queries in SQL. Where relevant, use the existence technique, mapping a Venn diagram of the underlying set comparison to predicate calculus.

(a) Who has an IQ above 120, and speaks both English and Japanese?
(b) Who speaks at least one of the languages spoken by Fumie?
(c) Who speaks at least all those languages spoken by Chris?
(d) Who speaks a language not spoken by any of the other applicants?
(e) Who speaks none of the languages spoken by Chris?
(f) Who speaks at most those languages spoken by David?
(g) Who speaks all the languages mentioned?
(h) Who speaks at least three languages?
(i) Who, besides David, speaks exactly the same (i.e., all and only) languages as David?
(j) Who has the highest IQ for his or her sex?
(k) Which languages are spoken by all the males?

***Applicant:***

| *person* | *IQ* | *sex* |
|---|---|---|
| Ann | 120 | F |
| Bill | 135 | M |
| Chris | 130 | F |
| David | 125 | M |
| Ernie | 115 | M |
| Fumie | 120 | F |

***Speaks:***

| *person* | *"language"* |
|---|---|
| Ann | English |
| Ann | Japanese |
| Bill | English |
| Bill | Japanese |
| Bill | Russian |
| Chris | English |
| Chris | Russian |
| David | English |
| David | Japanese |
| David | Russian |
| Ernie | English |
| Ernie | French |
| Ernie | Japanese |
| Fumie | Japanese |
| Fumie | Korean |

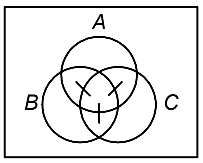
For each of the following queries, name the previous query to which it is equivalent, or, if equivalent to none of these state "none".

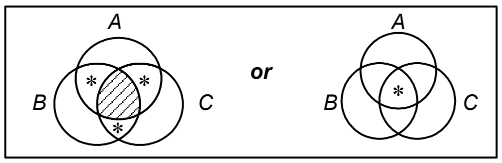
(l) Who speaks each language that is spoken by some applicant
(m) Who is not a person who speaks all of the languages spoken by Chris?
(n) Who has some language in common with Fumie?
(o) Who speaks a language spoken by no other applicant?
(p) Who, besides David, speaks a language if and only if David speaks it?
(q) Who speaks a language only if David speaks it?
(r) Who does not speak some of the languages spoken by Chris?
(s) Who does not speak any of the languages spoken by Chris?
(t) Whose languages include those spoken by Chris?

2. The following queries relate to the Applicant and Speaks tables of the previous exercise, which dealt with correlated and existential subqueries (see the answers). Use of **group by** and **in** often enables existence-based queries to be formulated more efficiently (e.g., using the grouping technique for set comparisons). Questions (a)-(m) of this exercise exploit this approach.
(a) Answer Question 1(c) without using a correlated subquery.
(b) Answer Question 1(d) without using a correlated subquery.
(c) Answer Question 1(e) without using **exists** or a correlated subquery.
(d) Answer Question 1(f) without using **exists** or a correlated subquery.
(e) Answer Question 1(k) without using **exists** or a correlated subquery.

Formulate SQL queries for the following:
(f) Which languages are spoken by exactly one person?
(g) Who speaks a language not spoken by Ernie?
(h) Which languages are spoken by, and only by, Ernie?
(i) Who speaks the same number of languages as David?
(j) Whose IQ is unique?
(k) Who has an IQ that is the same as that of another applicant?
(l) Who speaks all of Ann's languages as well as another language?
(m) List the name, sex and number of languages spoken by each applicant.

3. Figure (a) below is a Venn diagram, and Figure (b) is a disjunction of two Venn diagrams.

(a)


(b)


Which of the following options correctly describes the relationship between the figures?

A. (a) implies (b) but not conversely
B. (b) implies (a) but not conversely
C. (a) is equivalent to (b)
D. None of the above

4. The conceptual fact type Food contains Chemical in Percentage is mapped to the relation scheme *FoodComposition*( food, chemical, percent ). Consider the following query:

   List the food triples (*x*, *y*, *z*) where all three foods have exactly two chemicals in common, *x* and *y* have at least one other chemical in common, and *z* has no other chemicals (so *z* has just 2 chemicals, both of which occur in *x* and also in *y*).

   Mark this condition on a Venn diagram for the three sets: *Cx* = Chemicals in food *x*, *Cy* = Chemicals in food *y*, *Cz* = Chemicals in food *z*. Use "*" for existence, "*2" for exactly 2 exist, and shading for nonexistence. Map this condition to cardinality equations, using "#" for set cardinality. Now use the grouping technique to express this query in SQL. Include a condition to ensure that any given set of three foods appears on at most one row in the result. For example, if the triple (f1, f2, f3) occurs in the result, then the triple (f2, f1, f3) should not.

5. Consider the relation scheme *Membership*( team, member ) where an extensional uniqueness constraint applies to member (i.e., no two teams have exactly the same set of members). Write an SQL check clause to enforce this constraint.