So far we've talked about how to create a basic SELECTstatement and add qualifications to it to retrieve the data we want. In all of our previous examples, the data was located on a single table. More often than not, however, the data is going to be spread across multiple tables.

Data can be extracted from multiple tables at one time by *joining* the tables together using a common column. A **join** establishes a link between one or more tables whose rows share a common field or a field whose data is the same even if the name is different. More on this last point later.

**3.1 Relationships Between Data**

There are three types of relationships between tables: one-to-one (1 : 1), one-to-many (1 : many), and many-to-many (many : many). This relationship describes how two tables can be joined together.

**1 : 1 Relationships**

A 1 : 1 relationship means one row on the first table will only ever have one row on the second table. A PERSON will only ever have one birth date, a PERSON will only ever have one gender. It's not very common to have 1 : 1 relationships between two tables because the fields that store this type of data will generally always just exist on the first table.

Birth date and gender exist on the PERSON table. There is no point in creating a second table with a 1 : 1 relationship to store this type of information.

There are examples in Millennium of this relationship, however. The PERSON\_PATIENT table is an extension of the PERSON table. This table stores additional information about the patient not stored on the PERSON table.

< PERSON table vs PERSON\_PATIENT table.>

In cases like this, you will usually find a single table

<need a whole host of info here about 1:1, 1:many, and many:many relationships>

The PERSON\_ID is the primary key of the PERSON table and it is a foreign key on the ORDERS table. This relationship between PERSON table and the ORDERS table through the PERSON\_ID allows us to connect the data together.

<insert diagram here>

To do this, you would indicate in your query that each time the PERSON\_ID on the PERSON table is equal to a PERSON\_ID on the ORDERS table, then the information from both tables should be displayed together. Because each person has a unique PERSON\_ID, rows on both the PERSON and the ORDERS tables relate to the same patient.

**3.2 Inner Joins**

In the simplest form, you can use PLAN and JOIN clauses to link two tables together without adding any additional qualifications on either table. The following example reads the PERSON table and links to the ORDERS table by the PERSON\_ID. This means the common column in each table is the PERSON\_ID. The query will first find the row on the PERSON table and then find any matches for that person on the ORDERS table.

Don't actually run this example, it is for illustrative purposes only.

*select*

*\**

*from*

*person p*

*, orders o*

*plan p*

*join o*

*where o.person\_id = p.person\_id*

You will have one PLAN clause and as many JOIN clauses as there are additional tables in your query. Every table in the FROM clause must be listed in either a PLAN or a JOIN clause. For example:

from

tableA         A  
 , tableB B

, tableC C  
plan A  
join B where A.column = B.column  
join C where B.column = C.column

Every JOIN clause is followed with a qualification connecting that table to the previous one. In our example, we are connecting the PERSON table to the ORDERS table in the WHERE qualification.

Each PLAN or JOIN clause can have a WHERE clause with it to qualify data on only that table. Let's take our example from the previous chapter and expand upon it. In addition to viewing all of the "Smiths", let's look at their orders for the past year. Type in the following query:

*select*

*p.person\_id*

*, p.name\_full\_formatted*

*, p.birth\_dt\_tm*

*from*

*person p*

*, orders o*

*plan p*

*where p.name\_last\_key = "SMITH"*

*join o*

*where o.person\_id = p.person\_id*

*and o.orig\_order\_dt\_tm > cnvtdatetime(curdate - 365, 0)*

Let's add one more qualification to our query to make it more useful. Pick out one of the person\_ids from your own query and add a qualification under the WHERE clause of the PLAN. Better example without using name and person\_id maybe female vs male?

*select*

*p.person\_id*

*, p.name\_full\_formatted*

*, p.birth\_dt\_tm*

*, o.order\_mnemonic*

*, o.orig\_order\_dt\_tm*

*from*

*person p*

*, orders o*

*plan p*

*where p.name\_last\_key = "SMITH"*

*and p.person\_id = 22822461*

*join o*

*where o.person\_id = p.person\_id*

*and o.orig\_order\_dt\_tm > cnvtdatetime(curdate - 365, 0)*

Machine generated alternative text:
PERSON ID 
22222461 
22222461 
22822461 
22822461 
22222461 
NAME FULL FORMATTED 
BIRTH DT TM 
07/28/16 
07/28/16 
07/28/16 
07/28/16 
07/22/16 
ORDER MNEMONIC 
Est Pt Ofc or other 
ace t aminophen 
ibuprofen 
Est Pt cfc or other 
visit, 
visit, 
typi cal I y 
typi call y 
ORIG 
ORDER DT TM 
03/15/18 
03/21/18 
03/21/18 
03/21/18 
03/21/12 
SMITH, 
SMITH, 
SMITH, 
SMITH, 
SMITH, 
ÄDÄLINE 
ÄDÄLINE 
ÄDÄLINE 
ÄDÄLINE 
ÄDÄLINE 
Infect agent antigen detection by immunoassay 

When writing your query, the order of the tables is not extremely important. What is important is *how* your PLAN and JOIN clauses are set up. The most efficient method of joining tables is to **PLAN on a table while also using a qualification on an indexed column**. By doing this, the first table you read does an *indexed* read and each link is based on that index.

The reason I said *not* to run the first example is that there is no WHERE clause on the *PERSON* table. This means it does a full table read on the PERSON table (if you remember, in my environment there are over 24 million records) which is extremely inefficient.

If you have never seen joins before your head might be a large blur right now. Joins are not an easy concept to understand. The best way to fully understand them is through examples, so I recommend going through them (making sure to type them all out) and then re-read this section. We still have a lot more to discuss about joins, so it's vital that you have a strong understanding.

**Exercise 3.1**

If you are reading only two tables like we did in the previous examples, it is simple to determine which table to use in the PLAN clause and which to use in the JOIN clause. When you are reading from more than two, the task becomes more difficult.

Orders are encounter based, not patient specific. Therefore, a more realistic real-world example would be to find all orders on a specific encounter. For that, we need to join together three tables:

PERSON

ENCOUNTER

ORDERS

Let's first start by getting a person\_id. You might write your query with a test patient in your non-prod environment that doesn't have any orders. Don't fret. Keep looking, there are bound to be patients with orders.

select \* from person where name\_last\_key = "SMITH"

// → 22822461

So, which table do we start with first? Hint: The most efficient method of joining tables is to PLAN on a table with a using a qualification on an **indexed column on that table**.

For each query you write, you want to follow these steps:

1. Establish which table you will start with in the PLAN clause.
2. Now, you need to determine how to read the remaining tables. Start by defining which tables can be linked to the table in the PLAN clause and by what field. Common field names often end in \_ID or \_CD.
3. Establish a JOIN between the first and second table.
4. If there are more than two tables, determine how you can join that table to the second table and establish that join.
5. Continue building the joins until all table relationships have been established.

Using these steps, try and design a basic query joining the ORDERS, ENCOUNTER, and PERSON tables together. Aside from the join, the only qualification you should have is a qualification with the person\_id you found in your test environment.

Once you have the join path established, add the PERSON\_ID, full name, birth date, order mnemonic, and the order date/time to the SELECT clause and test your query out.

select

p.person\_id

, p.name\_full\_formatted

, p.birth\_dt\_tm

, o.order\_mnemonic

, o.orig\_order\_dt\_tm

from

person p

, encounter e

, orders o

plan p

where p.person\_id = 22822461

join e

where e.person\_id = p.person\_id

join o

where o.encntr\_id = e.encntr\_id

**Exercise 3.2**

As a general rule, it is best to read 1:1 relationships between tables before 1:many. To illustrate this point, we are going to join the PERSON, PERSON\_ALIAS, and ENCOUNTER tables. The *PERSON\_ALIAS* table holds all person level aliases (CPI, MRN, SSN etc).

A person has more than one alias, but in our example, we are going to limit the alias to only a Social Security Number (SSN). Since a patient only ever has one SSN, the *PERSON:PERSON\_ALIAS* is a 1:1 relationship, while the *PERSON:ENCOUNTER* relationship is a 1:many.

Type the following query in CCL to load the Tables/Fields for the *PERSON\_ALIAS* table.

*select \* from person\_alias pa where pa.*

Machine generated alternative text:
Tables/ Fields 
v person alias 
Fields 
person alias 
9 
person Id 
rowid 
9 
txn id text 
updt_applctx 
updt_cnt 
9 
updt_dt tm 
updt_id 
updt_task 
visit_seq_nbr 
B134316 DVDI • 
select 
from person alias pa where pa. 
Macr... 
Table.. 
Code... 
Requ... 

This properties window should look familiar. However, since this is a \_cd (code value) field, the properties looks a little different. In the properties for a code value field, we have a lookup option. I'm not going to dive into code values and code sets in this exercise. Chapter 4 is dedicated to this topic.

Select the lookup button.

Machine generated alternative text:
T able Name: 
PERSON ALIAS 
Code Set: 
Description: 
ield Properti 
Field Name: 
PERSON ALIAS TYPE CD 
Lookup 
person alias type code value 
D efinition: 
person alias type code identifies a kind or type of alias (La, ssn, mm, 
financial number, community mm, etcl they have cerner pre-defined 
meanings in the common data foundation table allowing hna applications 
to look for a specific kind of alias 
Close 

A query should have displayed listing every code\_value. We are after the CODE\_VALUE for the SSN, which might be the same as the output below (18.00).

Machine generated alternative text:
CODE VALUE 
59889793B 
10 
11 
12.00 
995226.00 
995227 
489058743. 
99SE2E 
995229. 
614347 
633621 
13. 
30507997 
14. 
647572.00 
15.00 
16. 
17 
61434B 
19. 
Health Exchange 
MRN 
National Health 
Donor ID 
Pa t ent 
Numb e r 
Identifier 
CDF 
MPI 
MEANING 
DESCRIPTION 
Alias used within an Affir 
Medical Record Number 
National Health Number 
Donor ID 
Recipient ID 
Cerner Card Number 
opo Donor ID 
OPO Recipient ID 
Other person ID 
Cut reach Person Identifie2 
NTKCRDNBR 
CPCD 
CPCR 
pus s PORT 
puss WORD 
PATID 
Recipient ID 
Cerner Card Number 
opc Donor ID 
OPO Recipient ID 
Other 
Cut reach Person Identifier 
Passport 
Password 
Patient ID — Classic 
PBS Patient ID 
PERSON NAME 
Placer Order 
PRN 
Px1D 
Referring MRN 
State Health Number 
ssN 
OCE 
Passport number 
Password 
vgoo Classic OCE 
PBS Patient ID 
PERSON NAME 
Placer Order 
Personnel Number 
Px1D 
Pa t ent 
PBSID 
PERSON 
PLACER 
REF MRN 
SHIN 
ssN 
NAME 
ORDER 
Referring Medical Record 
State Health Number 
Social Security 

Now that we have the code\_value for the person\_alias\_type\_cd for SSN, let's use it to finish our query and find a patient with a SSN.

*select \* from person\_alias pa where pa.person\_alias\_type\_cd = 18.00 with maxrec=1*

The reason we went through this exercise before even building our query joining the *PERSON*, *PERSON\_ALIAS*, and *ENCOUNTER* tables is that it is very likely a majority of patients in your non-production environment have SSNs. You now have a working example.

PLAN on the *PERSON* table using the person\_id you just gathered from running the above query. Join the *PERSON\_ALIAS* table to the *PERSON* table using the person\_alias\_type\_cd qualification. Lastly, join the *ENCOUNTER* table to the *PERSON* table.

If you joined the tables properly, you should see that the person\_id and the alias (SSN) repeats for each unique encntr\_id.

select

p.person\_id

, pa.alias

, e.encntr\_id

from

person p

, person\_alias pa

, encounter e

plan p

where p.person\_id = 2110427

join pa

where pa.person\_id = p.person\_id

and pa.person\_alias\_type\_cd = 18

join e

where e.person\_id = p.person\_id

Machine generated alternative text:
Edit Query 
PERSON ID 
2110427 
2110427 
2110427 
2110427 
2110427 
ALIAS 
715626477 
715626477 
715626477 
715626477 
715626477 
ENCNTR ID 
31SIE4E6.oo 
ISEES777 
21255320 
2212435B 

The take away from this example is that it's generally best to join tables that are 1:1 relationship before a 1:many. Since we only wanted to return the SSN of the patient, the *PERSON:PERSON\_ALIAS* was a 1:1. If we didn't have the SSN qualification it would have been a 1:many relationship.

Try removing the qualification with the person\_alias\_type\_cd and see what happens. You should see that for each alias, the encntr\_id's repeat themselves. If your patient has 5 aliases and two encounters, you should see a total of 10 rows.

**Exercise 3.3**

Let's say we want to list all of a patient's encounters and the FIN associated to each. For the same patient (person\_id) as exercise 3.2, write a query that joins the following tables:

PERSON

ENCOUNTER

ENCNTR\_ALIAS

1. Use the same strategy as exercise 3.2 to find the code\_value associated with the FIN for the encntr\_alias\_type\_cd field.
2. Write your PLAN and JOIN clauses joining the three tables using the person\_id as a qualification on the *PERSON* table and the code\_value you found in step 1 as a qualification on the *ENCNTR\_ALIAS* table.
3. Display the patient's name, registration date of the encounter, and the FIN.

select

p.name\_full\_formatted

, e.reg\_dt\_tm

, ea.alias

from

person p

, encounter e

, encntr\_alias ea

plan p

where p.person\_id = 2110427

join e

where e.person\_id = p.person\_id

join ea

where ea.encntr\_id = e.encntr\_id

and ea.encntr\_alias\_type\_cd = 1077

**Exercise 3.4**

Data can be extracted from multiple tables at one time by *joining* the tables together using a common column. That common column *does not* have to have the same name on both tables as long as the data on each column is the same. We will be looking at patients address' to illustrate this.

If you study the *ADDRESS* table you will not find a person\_id. So how do you join these two tables? Certain tables in Millennium, including the ADDRESS table, store unique identifiers in a field called parent\_entity\_id.

Machine generated alternative text:
T able Name: 
ADDRESS 
Description: 
parent entity id 
D efinition: 
-eld 
perti 
Field Name: 
PARENT ENTITY ID 
the value of the primary identifier of the table to which the address row is 
related (La, person_id, organization_id, etc) 
Close 

Tables that use a field called the parent\_entity\_id store data for more than one type or entity. Addresses exist not only for patients, but also for organizations and other entities.

1. Join the PERSON table to the ADDRESS table using your person\_id from exercise 3.3 as a qualification on the PERSON table.
2. Add a qualification to the ADDRESS table to only look for active addresses (a.active\_ind = 1).
3. Add the patient's name, street address, city, state, and zipcode to the query.

<image to illustrate how unique ID can be on both the person table and organization table at the same time>

There is one major problem with the query that you just wrote. Study the screen shot and see if you can find the issue. Notice the parent\_entity\_name field in conjunction with the parent\_entity\_id field.

Machine generated alternative text:
Edit Query 
ADDRESS ID 
4848259.00 
4831713.00 
IS129SOE 
PARENT ENTITY 
ORGANIZATION 
PERSON 
PERSON 
NAME 
PARENT ENTITY 
4987232.00 
4987232.00 
4927232.00 
ADDRESS 
TYPE CD 
751 
756.00 

The parent\_entity\_id field stores our person\_id, but it also stores the organization\_id from the ORGANIZATION table! This means that the parent\_entity\_id on the ADDRESS table *is not unique*. There could be instances when a person and an organization have the same identifier. If you don’t add a qualification to fix this, your query could return an extraneous row.

The simple fix is that you must use the parent\_entity\_name in addition to the parent\_entity\_id as a qualification on the ADDRESS table. Make the necessary modification now.

select

p.name\_full\_formatted

, a.street\_addr

, a.city

, a.state

, a.zipcode

from

person p

, address a

plan p

where p.person\_id = 2110427

join a

where p.person\_id = a.parent\_entity\_id

and a.parent\_entity\_name = "PERSON"

and a.active\_ind = 1

**Exercise 3.5**

In this exercise, we will be looking at the location data model and how organizations connect to nursing units, ambulatory locations, rooms and beds. We are excluding facilities and buildings for simplicity. After this exercise, you will have a better understanding of how your organization is set up! We need five tables to achieve our goal.

ORGANIZATION, LOCATION, NURSE\_UNIT, ROOM, BED

1. List the tables in a FROM clause and give each table an alias.
2. Study each table to determine your PLAN and JOIN clauses. Not all of the field names in this example will be the same. Remember - it's the data on the field that is important not the field name itself.
3. Add the following fields to the SELECT clause

Organization name

LOCATION.location\_type\_cd

LOCATION.location\_cd

ROOM.location\_cd

BED.location\_cd

1. Try running the query to see what displays.

At this point, you should definitely have a lot of information, but it doesn't mean a lot. The data is very unordered (you might have difficulty seeing this since almost everything returned is a number), and the code\_value numbers mean nothing.

1. Surround each \_cd field in the SELECT clause with the function uar\_get\_code\_display(). It looks like this:

*uar\_get\_code\_display(l.location\_type\_cd)*

Re-run the query and you should see sensible information that matches your organization's location hierarchy.

1. Step 5 allowed us to see the information we wanted, but it also changed the column name for each column we used the uar\_get\_code\_display() on. Notice how each of those column names is now entitled "EXP." Add a user-defined variable to each of these columns to change its name from "EXP" to org\_desc, loc\_type, lloc\_desc, room\_desc, and bed\_desc respectively.
2. Finally, the query is unordered. Let's fix that. Add an ORDER BY clause after the last JOIN clause to order the information returned in our query.

*ORDER BY*

*org\_desc*

*, loc\_type*

*, loc\_desc*

If you have trouble with this query take a peek at the answer, but come back and finish writing it on your own. A number of new concepts were introduced that are important including the uar\_get\_code\_display() function, which we will be talking about in the next chapter, and also the ORDER BY clause, another reason why using user-defined functions is helpful.

select

org\_desc = o.org\_name

, loc\_type = uar\_get\_code\_display(l.location\_type\_cd)

, loc\_desc = uar\_get\_code\_display(l.location\_cd)

, room\_desc = uar\_get\_code\_display(r.location\_cd)

, bed\_desc = uar\_get\_code\_display(b.location\_cd)

from

organization o

, location l

, nurse\_unit nu

, room r

, bed b

plan o

where o.active\_ind = 1

join l

where l.organization\_id = o.organization\_id

and l.active\_ind = 1

join nu

where nu.location\_cd = l.location\_cd

and nu.active\_ind = 1

join r

where r.loc\_nurse\_unit\_cd = nu.location\_cd

and r.active\_ind = 1

join b

where b.loc\_room\_cd = r.location\_cd

order by

org\_desc

, loc\_type

, loc\_desc

**3.4 Outer Joins**

Up to this point we have been talking about joining tables together in instances where the unique identifiers exist on each table. A join between two tables where the information exists on both tables is called an **inner join**. However, it is not always the case that information resides on both tables.

There are times when you want to display information on a preceding table, even if no information exists on subsequent tables that was joined. These types of joins are called **outer joins**. Outer joins are a type of join that display rows from a preceding table even if information does not exist on subsequent tables.

To see this in action, run the following example. This query returns every patient with the last name "SMITH" that has a SSN.

*select*

*p.person\_id*

*, p.name\_full\_formatted*

*, p.birth\_dt\_tm*

*, pa.alias*

*, ssn = uar\_get\_code\_display(pa.person\_alias\_type\_cd)*

*from*

*person p*

*, person\_alias pa*

*plan p*

*where p.name\_last\_key = "SMITH"*

*join pa*

*where pa.person\_id = p.person\_id*

*and pa.person\_alias\_type\_cd = 18*

*and pa.active\_ind = 1*

In my environment, there are 99 total examples.

C:\Users\b134316\AppData\Local\Temp\msohtmlclip1\02\clip_image008.png

However, there are a total of 140 Smith's in my database.

*select count(\*) from person p where p.name\_last\_key = "SMITH"*

Machine generated alternative text:
Edit Query 
EXP 
140.00 

41 Smiths do not have SSNs and are therefore not returned. What if we wanted to list all Smiths whether they have a SSN or not? This can be accomplished using the OUTERJOIN keyword on each qualification of the table we want to outer join.

*select*

*p.person\_id*

*, p.name\_full\_formatted*

*, p.birth\_dt\_tm*

*, pa.alias*

*, ssn = uar\_get\_code\_display(pa.person\_alias\_type\_cd)*

*from*

*person p*

*, person\_alias pa*

*plan p*

*where p.name\_last\_key = "SMITH"*

*join pa*

*where pa.person\_id = outerjoin(p.person\_id)*

*and pa.person\_alias\_type\_cd = outerjoin(18)*

*and pa.active\_ind = outerjoin(1)*

Notice how the OUTERJOIN keyword is used for *each* qualification of the PERSON\_ALIAS table. If you don't use OUTERJOIN on each qualification, the join used will be an inner join.

**Exercise 3.6**

Using nested selects with exist

**3.5 Joining the Same Table Together**

A natural assumption to make is that a table can only be listed in the FROM clause a single time. In reality, you can list the same table multiple times by simply defining a unique alias for each table and creating a join between them. When possible, it's more efficient to read a table once rather than multiple times. However, there are use cases where joining the same table twice is advantageous.

Many of the examples I used in this book were created by using this method. In example 3.4 we looked at instances where the same PARENT\_ENTITY\_ID could exist for a PERSON as well as an ORGANIZATION. I wasn't about to spend hours looking through the database to find an example to illustrate this point, so I wrote a simple query to do so.

*select*

*\**

*from*

*address a*

*, address a2*

*plan a*

*where a.parent\_entity\_name = "PERSON"*

*join a2*

*where a2.parent\_entity\_id = a.parent\_entity\_id*

*and a2.parent\_entity\_name = "ORGANIZATION"*

*with maxrec = 2, time = 90*

There are two different rows on the ADDRESS table for the ORGANIZATION for the PERSON address. To connect the two together, two tables are needed, one for the PERSON and one for the ORGANIZATION.

The PARENT\_ENTITY\_ID joins the two ADDRESS tables together. One table has a qualification on the PARENT\_ENTITY\_NAME of "PERSON" and the other table for "ORGANIZATION". Creating a join path between these two tables will return a single row with both the PERSON and ORGANIZATION information.

<Show diagram>

Joining two tables is necessary when information you want to gather is stored on two separate rows on the same table. Let's say you want to gather a list of encounters that have a Hemoglobin a*nd* a BUN order. Each order is stored on its own separate row on the ORDERS table.

*select*

*e.encntr\_id,*

*, o.order\_mnemonic*

*, o1.order\_mnemonic*

*from        encounter e,*

*orders o,*

*orders o1*

*plan e*

*join o*

*where e.encntr\_id = o.encntr\_id*

*and o.order\_mnemonic = "BUN"*

*join o1*

*where e.encntr\_id = o1.encntr\_id*

*and o1.order\_mnemonic = "Hemoglobin\*"*

Since this query has two inner joins connected to the ENCOUNTER table, both conditions must be met for a row to return. That is, a BUN order and a Hemoglobin order must exist.

**3.6 Non-linear joins**

A JOIN between two or more tables resulting in an intersection between them is called a *linear join path*. We will be covering *non-linear join paths* later, which is when this distinction will be important to understand.

**Chapter 3 Recap**

Congratulations on getting through a lot of information. Joins are a very complex and very important topic