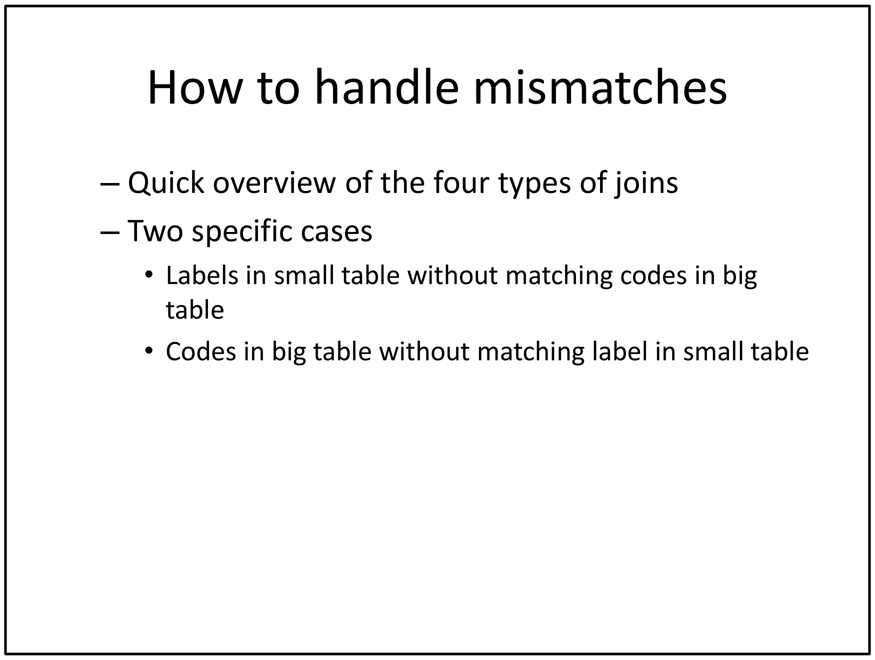
# Mismatches

Suman Sahil, Steve Simon

Creation date: 2019-10-16

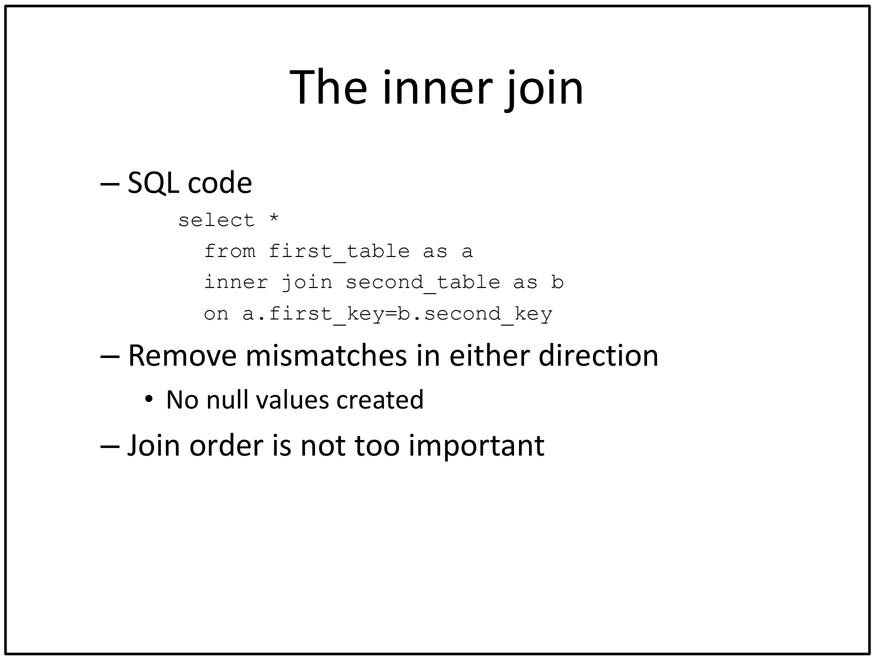


## How to handle mismatches

– Quick overview of the four types of joins
– Two specific cases
  - Labels in small table without matching codes in big table
  - Codes in big table without matching label in small table

Speaker notes:

In the previous module, I mentioned a simple case: every label in the small table has a code in the big table and vice versa. In this lecture, I'll discuss cases where you have mismatches. Before I do that, let me talk briefly about the four major types of joins.



## The inner join

- SQL code

```
select *
  from first_table as a
  inner join second_table as b
  on a.first_key=b.second_key
```

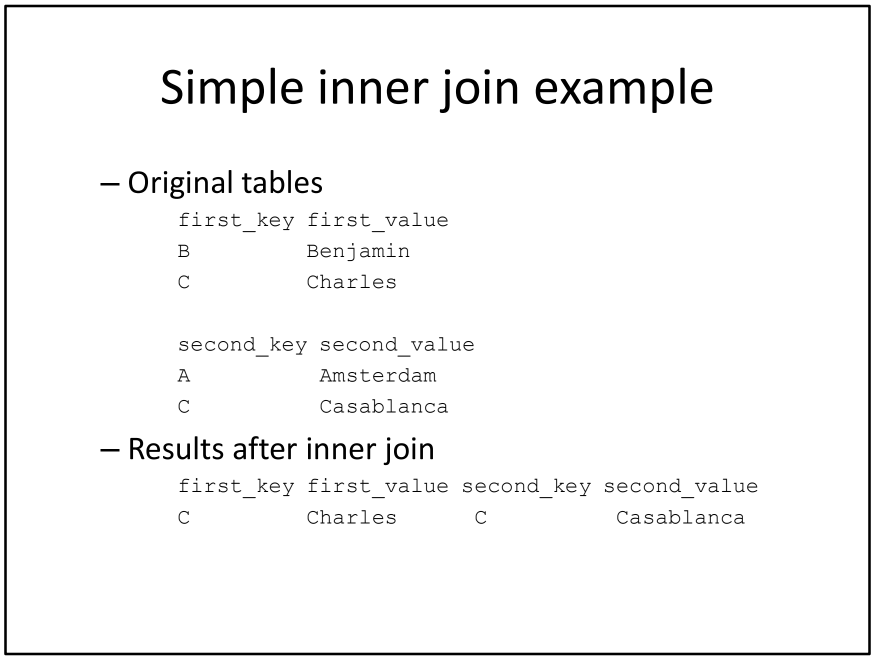
- Remove mismatches in either direction
  - No null values created
- Join order is not too important

Speaker notes:

We’ll describe this in more detail in a different video, but here’s a quick overview of the four basic types of joins.

The inner join removes any mismatches. If a key in the first table is not found in the second table, sorry but you’re gone. If a key in the second table is not found in the first table, sorry but you’re gone also.

Note that the order is not important here. If you swapped the first table and the second table, maybe the arrangement of the columns might change, and maybe the arrangement of the rows might change, but the table would be the same size and would effectively contain the same information.



## Simple inner join example

– Original tables

```
first_key first_value
B         Benjamin
C         Charles

second_key second_value
A          Amsterdam
C          Casablanca
```

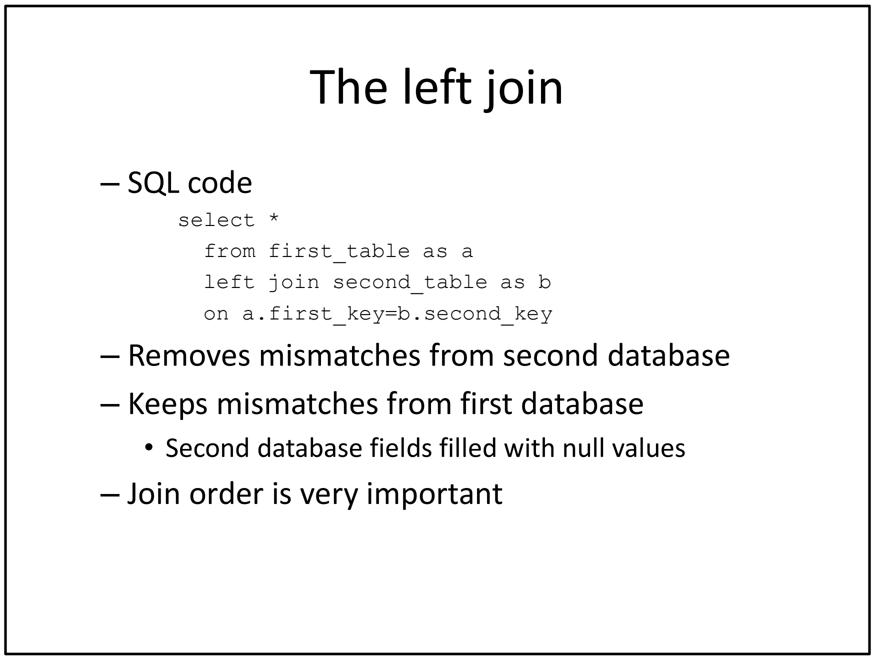
– Results after inner join

```
first_key first_value second_key second_value
C         Charles      C          Casablanca
```

Speaker notes:

Here’s a simple example of an inner join with mismatches going in both directions. The first table has Benjamin and Charles with keys of B and C. The second table has Amsterdam and Casablanca with keys of A and C.

Benjamin doesn’t make it because B has no matching key in the second table. Amsterdam doesn’t make it because A has no matching key in the first table. But you can match Charles with Casablanca because the C in the first table has a match in the second table. So the inner join has only one record.



## The left join

- SQL code

```
    select *
      from first_table as a
      left join second_table as b
      on a.first_key=b.second_key
```

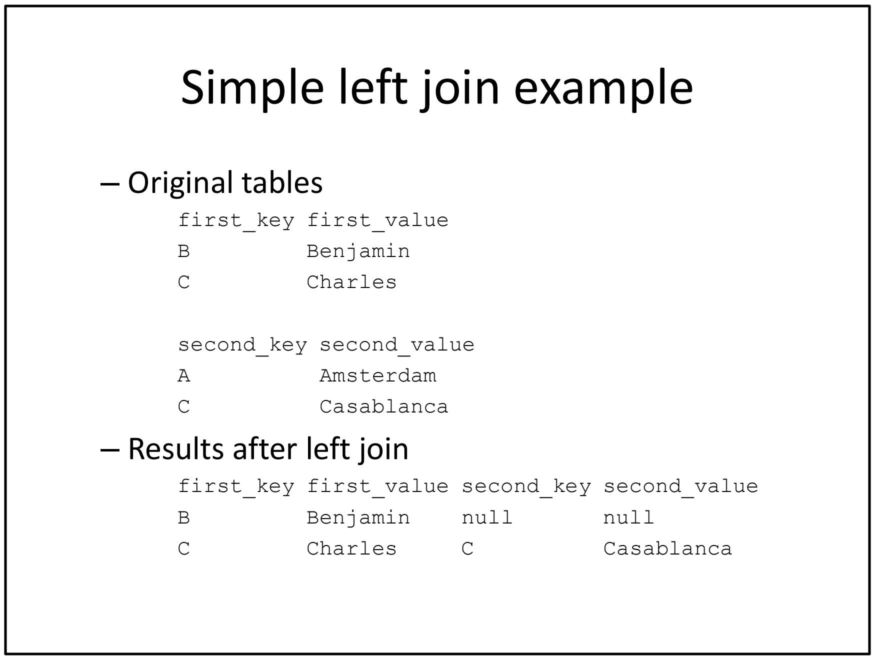
- Removes mismatches from second database
- Keeps mismatches from first database
  - Second database fields filled with null values
- Join order is very important

Speaker notes:

The left join will remove mismatches from the second database. If a key in the second database is not found in the first database, then sorry but you’re gone.

Mismatches from the first database will not get tossed, however. If a key in the first database is not found in the second database, that’s okay. Keep the record in the join. The fields that you couldn’t find in the second database because of the mismatch? Let’s set them to null.

Join order is very important for a left join.



## Simple left join example

– Original tables

```
first_key first_value
B         Benjamin
C         Charles

second_key second_value
A          Amsterdam
C          Casablanca
```

– Results after left join

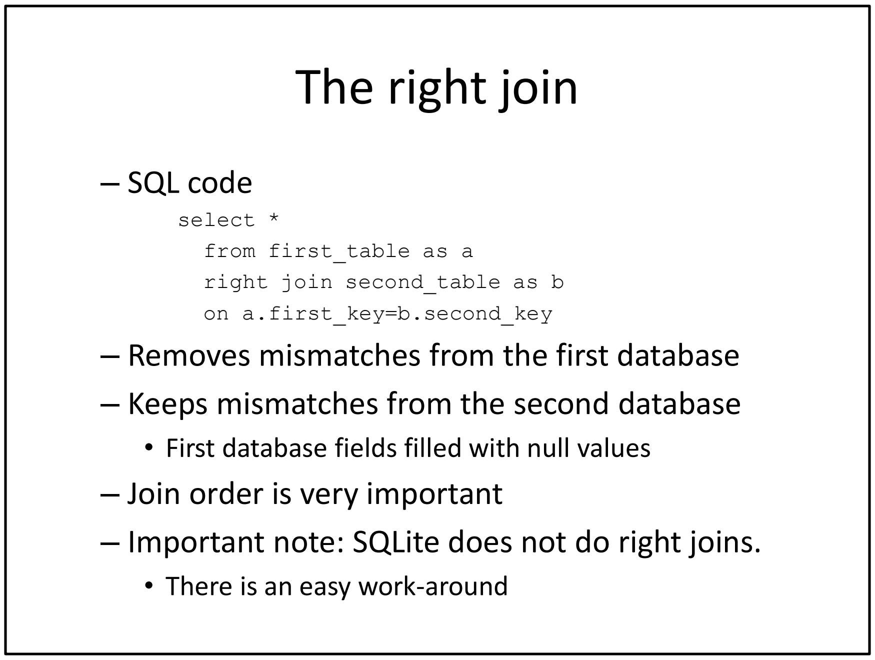
```
first_key first_value second_key second_value
B         Benjamin    null       null
C         Charles     C          Casablanca
```

Speaker notes:

Here's the same simple example, but now with a left join. Benjamin in the first table has no match in the second table, but he's lucky because the left join will keep him in the final database, even without a match. The second key and second value fields are both set to null.

Amersterdam, however, is not so lucky. The left join will toss out any mismatches in the second table.

Charles and Casablanca get matched, of course, just like before.



## The right join

- SQL code

```
select *
  from first_table as a
  right join second_table as b
  on a.first_key=b.second_key
```

- Removes mismatches from the first database
- Keeps mismatches from the second database
  - First database fields filled with null values
- Join order is very important
- Important note: SQLite does not do right joins.
  - There is an easy work-around

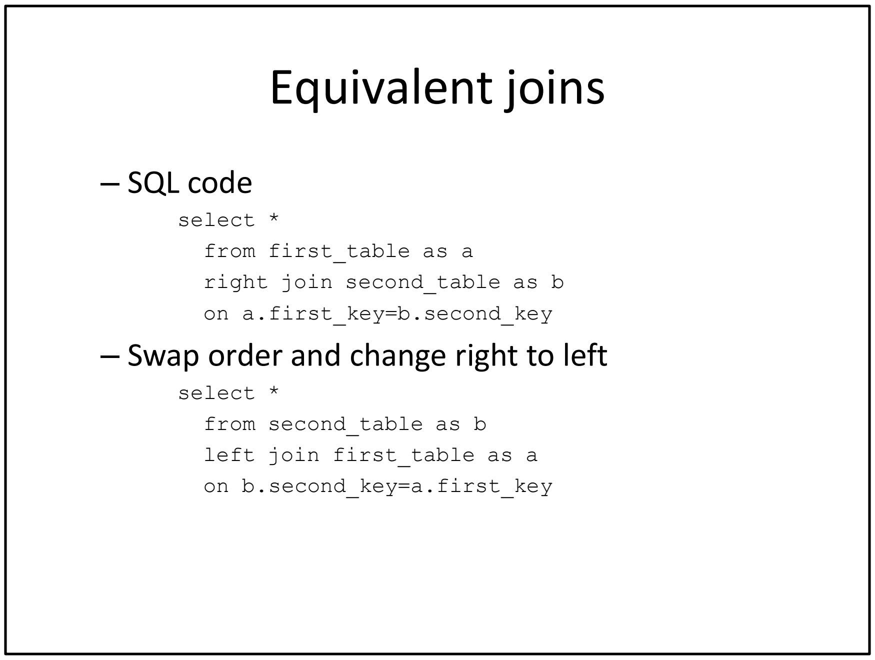
Speaker notes:

The right join will remove mismatches from the first database. If a key in the first database is not found in the second database, then sorry but you're gone.

Mismatches from the second database will not get tossed, however. If a key in the second database is not found in the first database, that's okay. Keep the record in the database. The fields that you couldn't find in the first database because of the mismatch? Let's set them to null.

Join order is very important for a right join.

Note that SQLite cannot do a right join. It can only do an inner join and a left join. This is the only SQL database, as far as I know, that cannot do a right join. There is an easy work around for the right join. Just swap the order of the tables and do a left join.



## Equivalent joins

– SQL code

```
select *
  from first_table as a
  right join second_table as b
  on a.first_key=b.second_key
```

– Swap order and change right to left

```
select *
  from second_table as b
  left join first_table as a
  on b.second_key=a.first_key
```

Speaker notes:

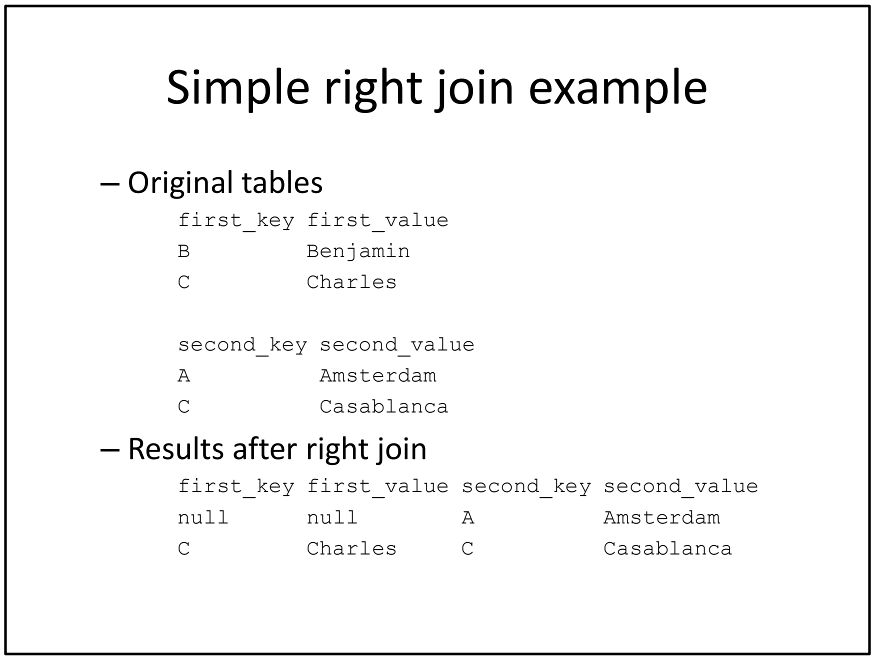
It’s not too hard to see how the second set of SQL commands will produce the same effective results as the first set of commands.

The order of the fields will be different, and the order of the records might or might not be different, but you get the same effective results.

Let me elaborate a bit on order. When you say "select *“, you will get back all the fields in the table mentioned in the”from" statement followed by all the fields in the table mentioned in the “join” statement.

Order in SQL is often arbitrary, but what typically happens is that the join starts with the first record of the table mentioned in the “from” statement. That would be “Benjamin” if you coded things as “from table_one, but it would be”Amsterdam" if you coded things as “from table_two”.

Of course, you can control the order in which fields appear by specifying that order in the “select” statement and you can control the order in which records appear by using an “order by” statement.



## Simple right join example

– Original tables

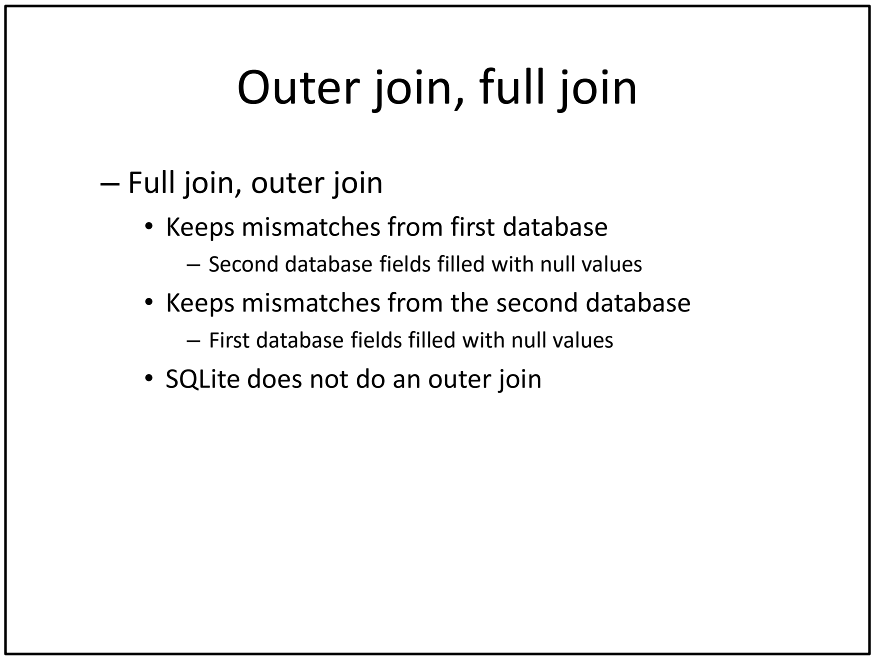
```
first_key first_value
B         Benjamin
C         Charles

second_key second_value
A          Amsterdam
C          Casablanca
```

– Results after right join

```
first_key first_value second_key second_value
null      null        A          Amsterdam
C         Charles     C          Casablanca
```

Speaker notes:

Here’s that same set of tables with a right join. Benjamin gets the boot because he has no match in the second table. But Amsterdam makes the cut on a right join. Charles and Casablanca get matched, of course.



## Outer join, full join

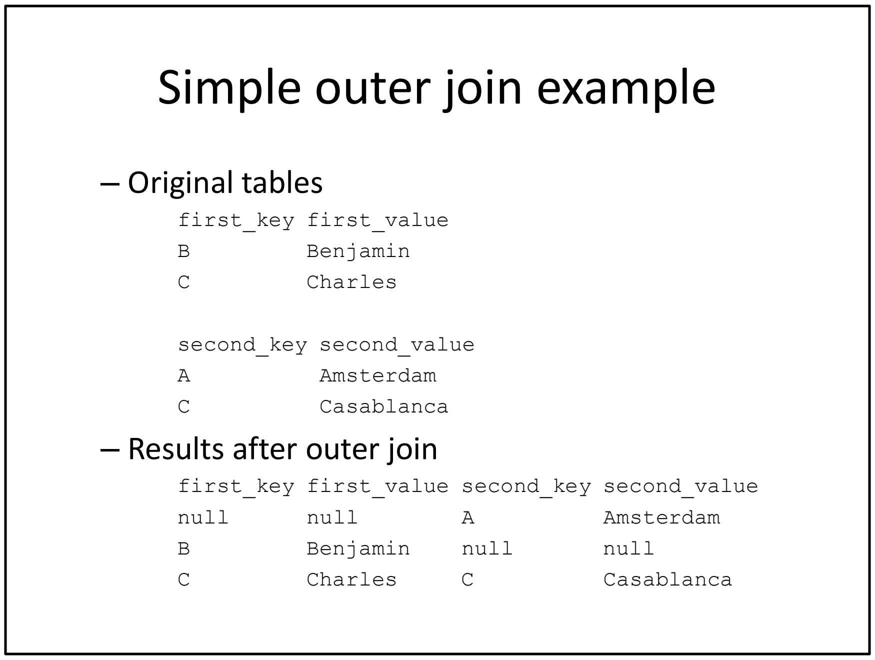
- Full join, outer join
  - Keeps mismatches from first database
    - Second database fields filled with null values
  - Keeps mismatches from the second database
    - First database fields filled with null values
  - SQLite does not do an outer join

Speaker notes:

A full join lets everyone join the party. A key in the first database that doesn’t have a match in the second database? Come on it. We’ll patch things up by using null values for the fields in the second database.

A key in the second database that doesn’t have a match in the first database? Why, you’re welcome too. We’ll use null values for the fields in the first database.

SQLite does not do an outer join. This is a surprising limitation, and it does cause problems. There are ways to work around this limitation, but they are complex and awkward to code. As far as I know, SQLite is the only SQL database with this limitation.



## Simple outer join example

– Original tables

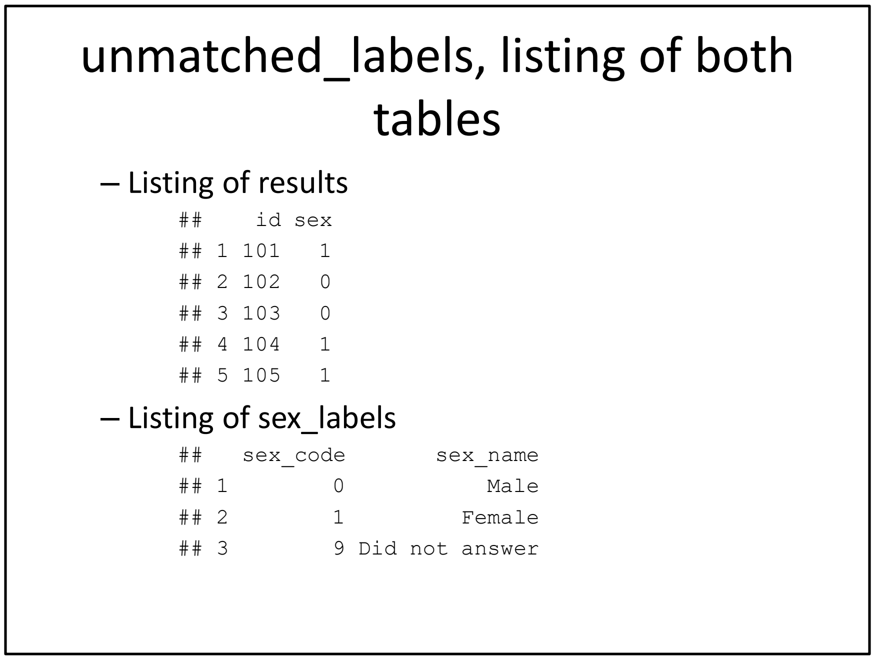
```
first_key first_value
B         Benjamin
C         Charles

second_key second_value
A          Amsterdam
C          Casablanca
```

– Results after outer join

```
first_key first_value second_key second_value
null      null        A          Amsterdam
B         Benjamin    null       null
C         Charles     C          Casablanca
```

Speaker notes:

Here's an example of an outer join. Benjamin gets in and Amsterdam gets in, in spite of their mismatches. Charles and Casablanca? They get in for any join–inner, left, right, or outer.



## unmatched_labels, listing of both tables

– Listing of results

```
##    id sex
## 1 101   1
## 2 102   0
## 3 103   0
## 4 104   1
## 5 105   1
```

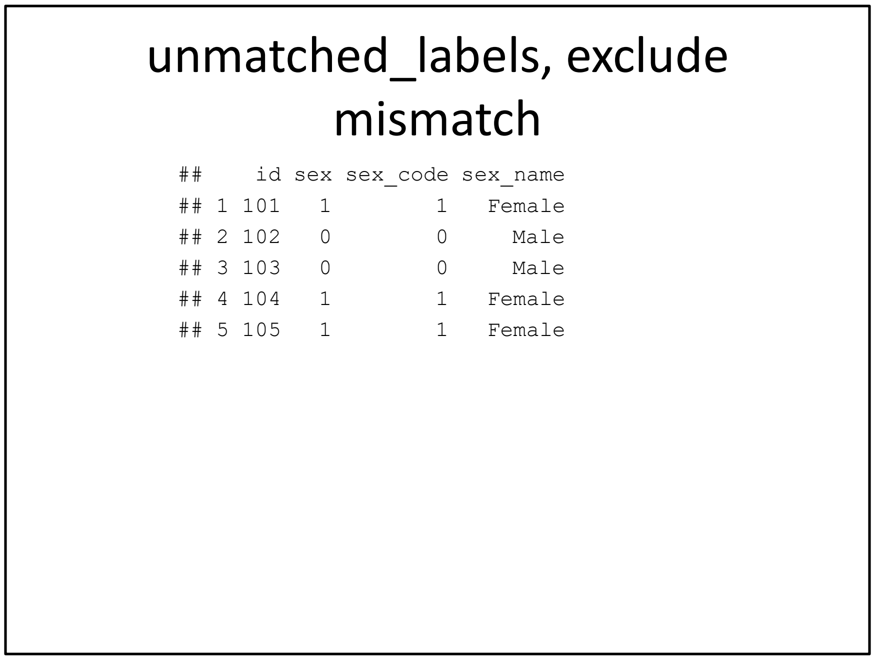
– Listing of sex_labels

```
##   sex_code       sex_name
## 1        0           Male
## 2        1         Female
## 3        9 Did not answer
```

Speaker notes:

Let’s look at an example with joining using labels. This is similar to the example you saw in an earlier video, but I’ve stripped it down so there are onle five records in the results table and we are only worried about labels for the sex variable.

There’s a change here, though. There is a label “Did not answer” in the sex table, but the code of 9 associated with that label never appears in the results. This is not a surprise. There are many studies where everyone answers this question.

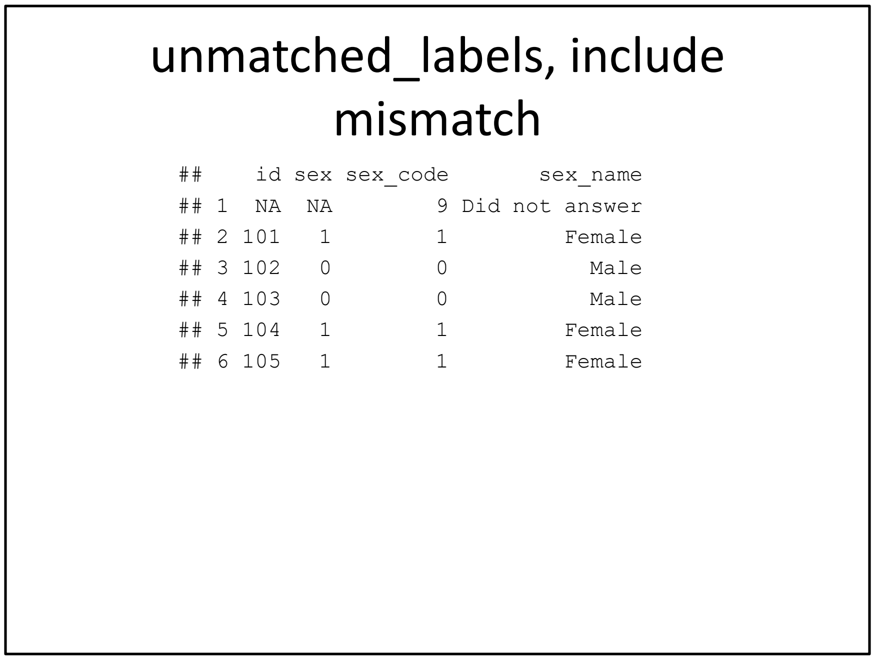


## unmatched_labels, exclude mismatch

```
##     id sex sex_code sex_name
## 1 101   1        1   Female
## 2 102   0        0     Male
## 3 103   0        0     Male
## 4 104   1        1   Female
## 5 105   1        1   Female
```

Speaker notes:

If you exclude the record in sex_labels that does not have a match in the results, you get these results. You get five records total and you don't see a "9" or a "Did not answer" anywhere.



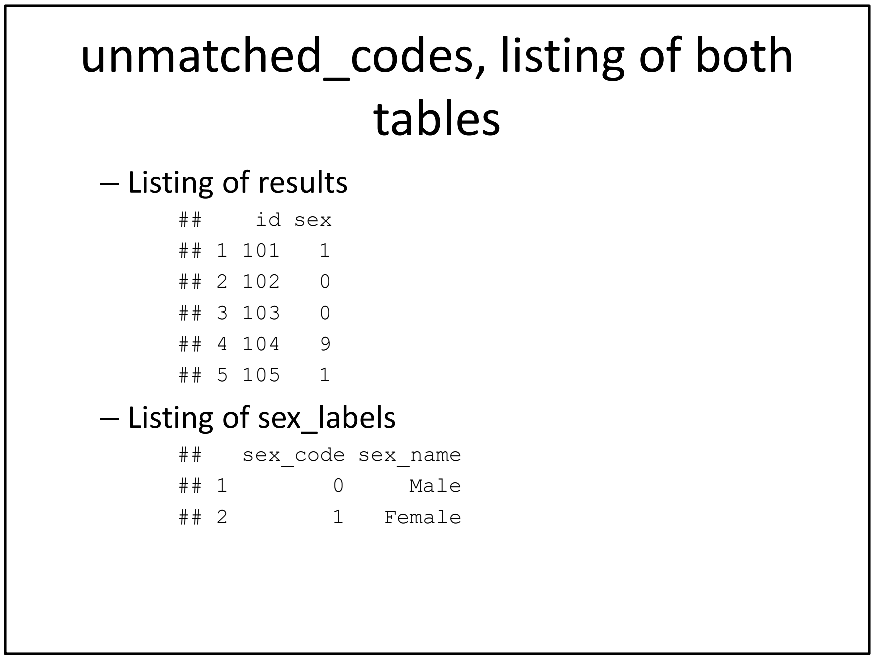
## unmatched_labels, include mismatch

```
##    id sex sex_code       sex_name
## 1  NA  NA        9 Did not answer
## 2 101   1        1         Female
## 3 102   0        0           Male
## 4 103   0        0           Male
## 5 104   1        1         Female
## 6 105   1        1         Female
```

Speaker notes:

This is what the results look like if you allow the mismatched table to get in. There are six rows now, one more than the earlier result and you have a missing value for id.

There may be times when this is okay, but adding an extra phantom subject seems like a bad idea to me. Having a null value for the id variable seems like a bad idea to me. If a label is unmatched, you alomst always want to leave it out.



## unmatched_codes, listing of both tables

– Listing of results

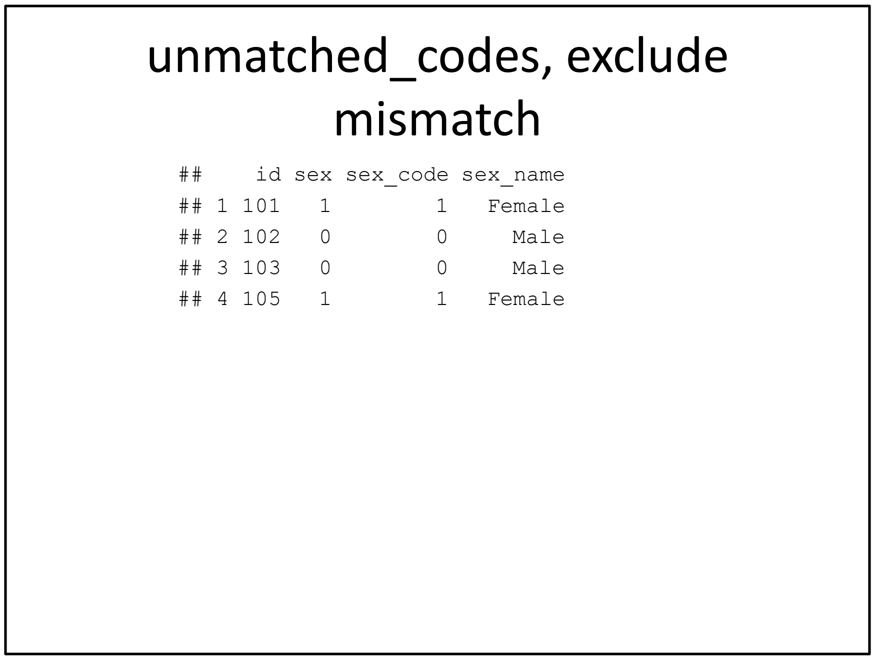
```
##    id sex
## 1 101   1
## 2 102   0
## 3 103   0
## 4 104   9
## 5 105   1
```

– Listing of sex_labels

```
##   sex_code sex_name
## 1        0     Male
## 2        1   Female
```

Speaker notes:

Here is a different case. In these two tables, there is a code in results that does not have a match in sex_labels.

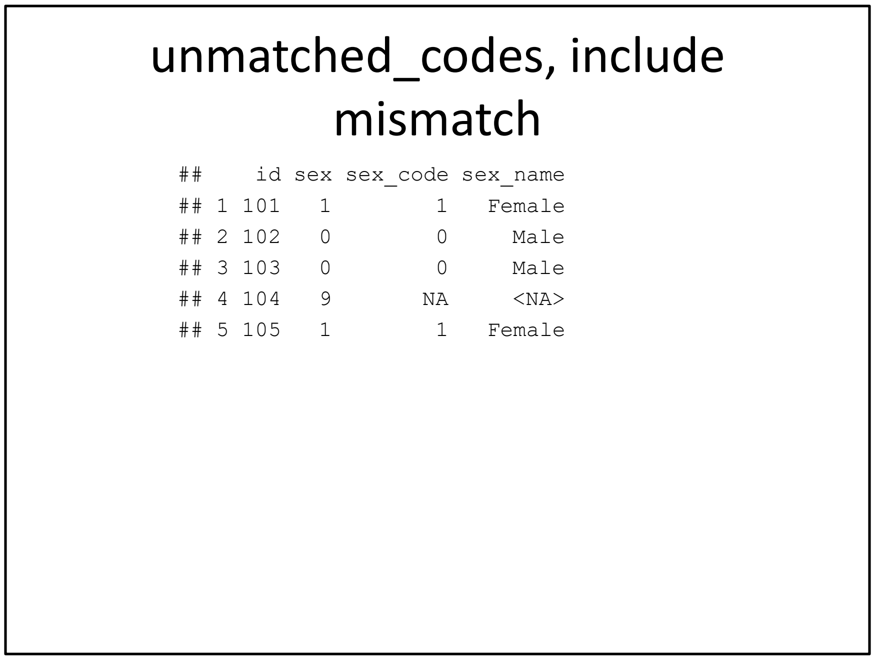


## unmatched_codes, exclude mismatch

```
##    id sex sex_code sex_name
## 1 101   1        1   Female
## 2 102   0        0     Male
## 3 103   0        0     Male
## 4 105   1        1   Female
```

Speaker notes:

If you exclude the mismatch, one of the five subjects gets tossed out. From a data analysis perspective, this is almost always a bad idea.

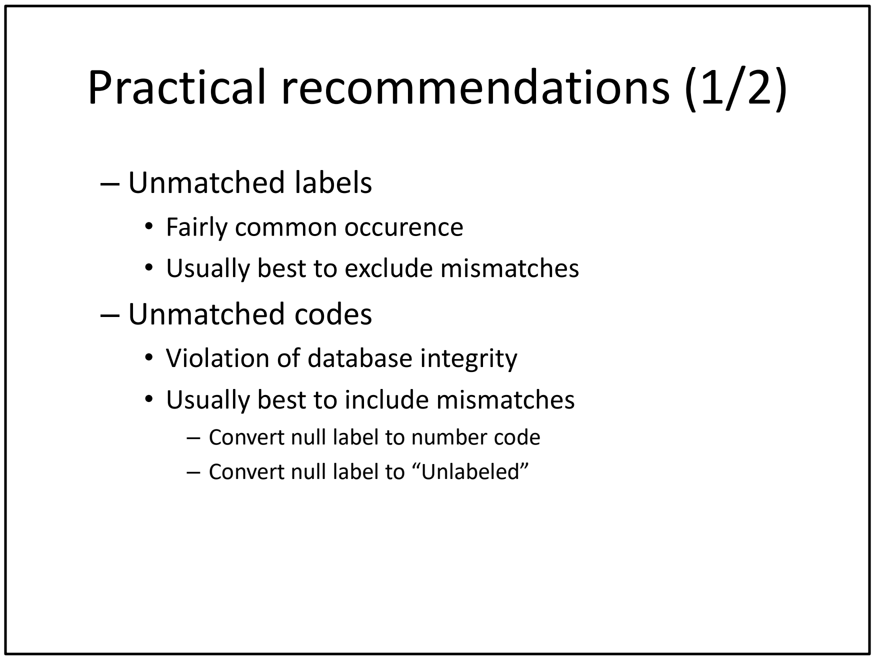


## unmatched_codes, include mismatch

```
##     id sex sex_code sex_name
## 1 101   1        1   Female
## 2 102   0        0     Male
## 3 103   0        0     Male
## 4 104   9       NA     <NA>
## 5 105   1        1   Female
```

Speaker notes:

Here are the results where you leave the mismatch in. It has the same number of subjects as the original results has. This seems like a better approach. You could replace the null value for label with the number code, or you could replace the null value with a string such as “Unlabeled.”



## Practical recommendations (1/2)

- Unmatched labels
  - Fairly common occurence
  - Usually best to exclude mismatches
- Unmatched codes
  - Violation of database integrity
  - Usually best to include mismatches
    - Convert null label to number code
    - Convert null label to “Unlabeled”

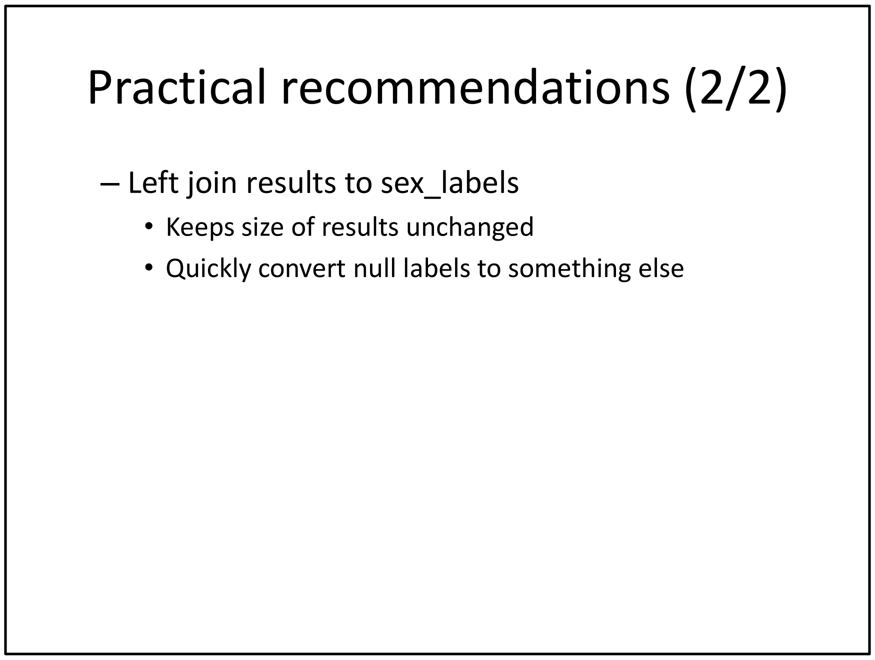
Speaker notes:

Here’s some practical advice that applies only to the setting where your extra tables include labels to match number codes in the main table.

Unmatched labels are a fairly common occurence. If you have 68,000 codes for diagnosis, it would be unrealistic to expect each of those codes to appear in the main table. You are usually best off ignoring the unmatched labels.

Unmatched codes, however, should almost never be ignored. There’s a technical term associated with this, and I am not the one to talk about it, but the codes in the large table are foreign keys and a foreign key should always have a match in the table with the labels. A mismatch here is a violation of a fundamental rule of database integrity.

It is usually best to allow mismatches. Having a null value for the label is a small problem, but tossing out a legitimate record in the larger database is a bigger problem. Almost always, you want to include these types of mismatches.



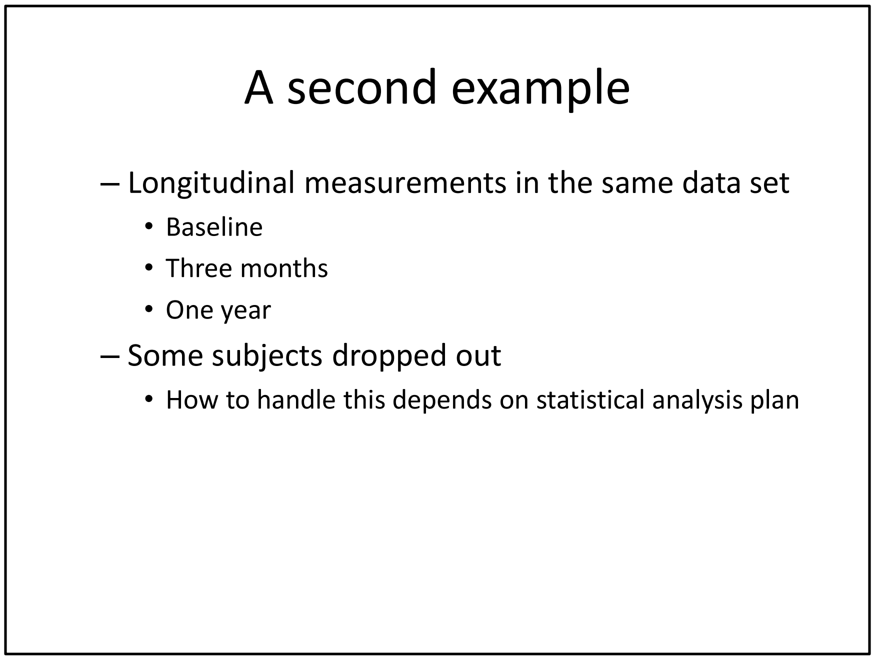
## Practical recommendations (2/2)

- Left join results to sex_labels
  - Keeps size of results unchanged
  - Quickly convert null labels to something else

Speaker notes:

With these two rules in mind, ignoring unmatched labels but including unmatched codes, you should use a left join of results to the sex_labels. This guarantees that no one in the results gets removed and also insures that you don’t add extra “phantom” records with null id values.

It is usually good practice to quickly convert the null label back to the code number or to a generic term like “unlabeled.”

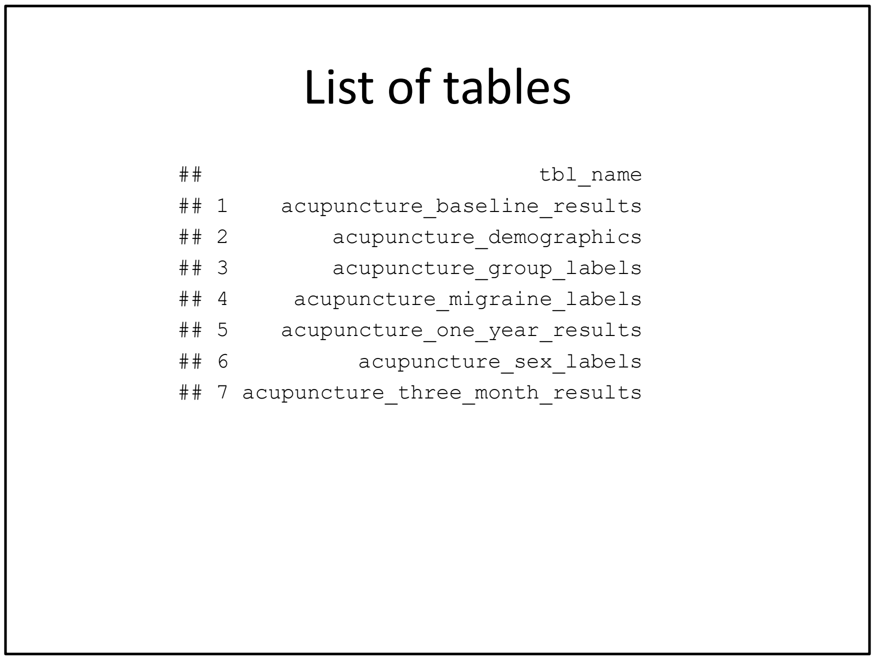


## A second example

- Longitudinal measurements in the same data set
  - Baseline
  - Three months
  - One year
- Some subjects dropped out
  - How to handle this depends on statistical analysis plan

Speaker notes:

This dataset has tables for results at baseline, three months, and one year. It has a separate table for demographics, and it has tables with labels for some of the demographics.



## List of tables

```
##                            tbl_name
## 1    acupuncture_baseline_results
## 2        acupuncture_demographics
## 3        acupuncture_group_labels
## 4     acupuncture_migraine_labels
## 5    acupuncture_one_year_results
## 6          acupuncture_sex_labels
## 7 acupuncture_three_month_results
```

Speaker notes:

Here is a list of the tables in this database.

## List of fields in acupuncture_demographics

```
##    id age sex migraine chronicity
## 1 100  47   1        1         35
## 2 101  52   1        1          8
## 3 104  32   1        1         14
## 4 105  53   1        1         10
##   acupuncturist practice_id grp
## 1            12          35   1
## 2            12          35   0
## 3            12          35   0
## 4             9          25   0
```

Speaker notes:

The demographics table has eight fields.

## List of fields in acupuncture_baseline_results

```
##     id   pk1 f1
## 1  100 10.75  8
## 2  101  9.50  4
## 3  104 16.00 12
## 4  105 32.50 21
## 5  108 16.50 14
## 6  112  9.25 15
## 7  113 42.50 25
## 8  114 24.25 14
## 9  126 21.00 11
## 10 130 21.75 11
```

Speaker notes:

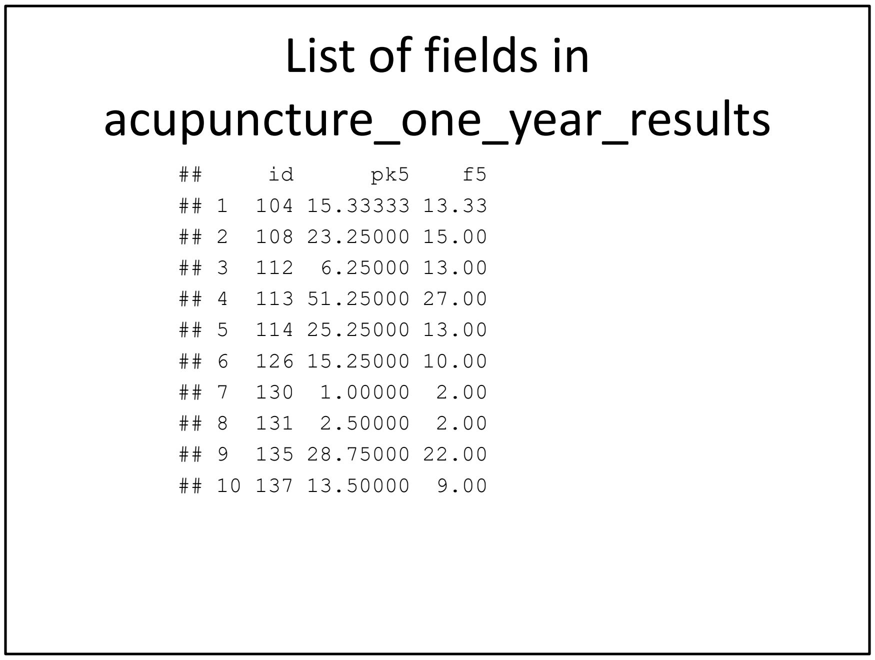
Here are first few rows of the baseline results.

## List of fields in acupuncture_three_month_results

```
##     id   pk2 f2
## 1  105 44.00 18
## 2  108 17.50 19
## 3  112  4.75  8
## 4  113 34.50 24
## 5  114 16.25 11
## 6  126 18.50 10
## 7  130  2.00  2
## 8  131 20.00 10
## 9  135 37.25 23
## 10 137 11.25  8
```

Speaker notes:

Here are first few rows for the three month results.

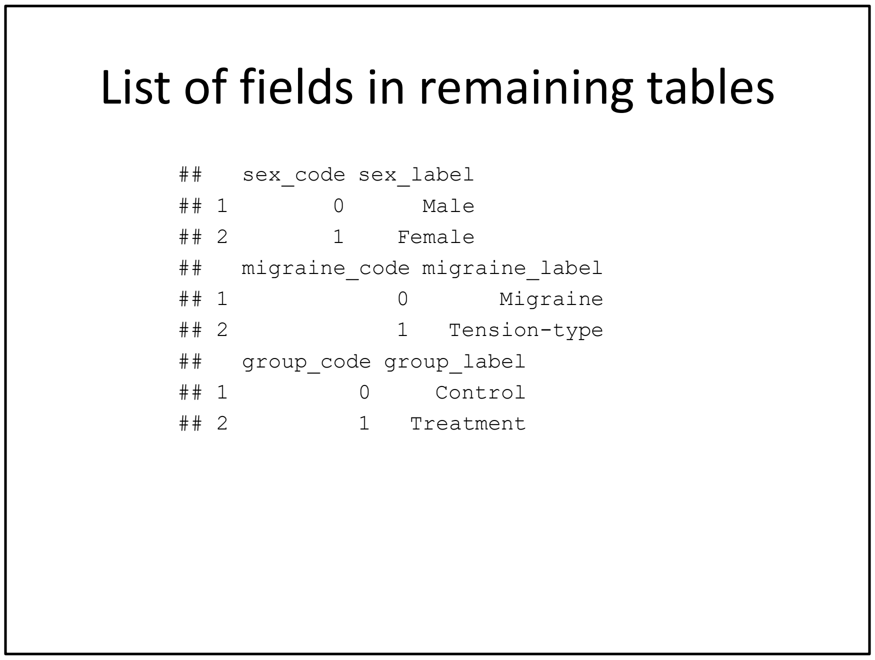


## List of fields in acupuncture_one_year_results

```
##     id      pk5    f5
## 1  104 15.33333 13.33
## 2  108 23.25000 15.00
## 3  112  6.25000 13.00
## 4  113 51.25000 27.00
## 5  114 25.25000 13.00
## 6  126 15.25000 10.00
## 7  130  1.00000  2.00
## 8  131  2.50000  2.00
## 9  135 28.75000 22.00
## 10 137 13.50000  9.00
```

Speaker notes:

Here are first few rows of the one year results

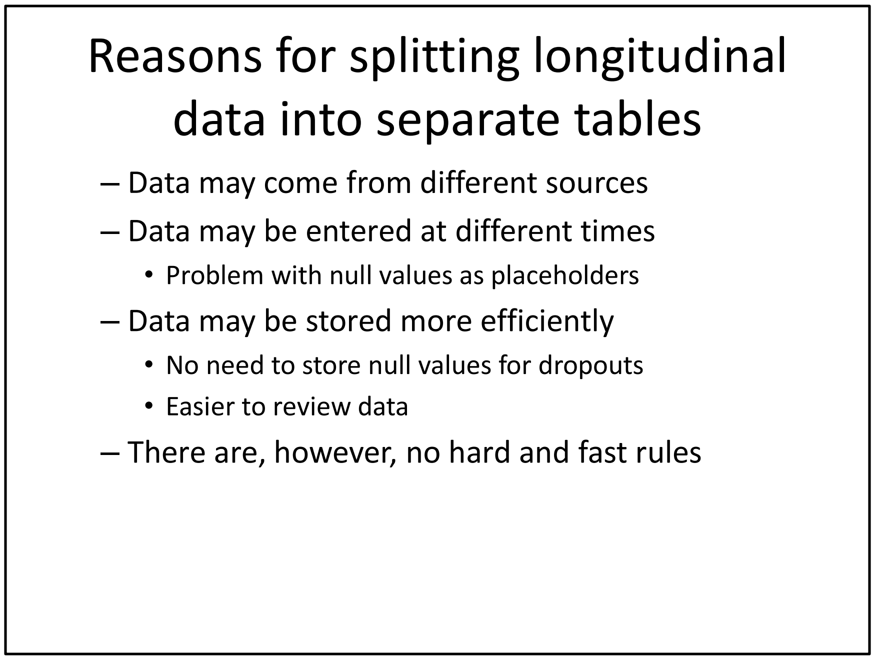


## List of fields in remaining tables

```
##   sex_code sex_label
## 1        0      Male
## 2        1    Female
##   migraine_code migraine_label
## 1             0       Migraine
## 2             1   Tension-type
##   group_code group_label
## 1          0     Control
## 2          1   Treatment
```

Speaker notes:

Here are the remaining tables, used for assigning labels to number codes.


## Reasons for splitting longitudinal data into separate tables

– Data may come from different sources
– Data may be entered at different times
  • Problem with null values as placeholders
– Data may be stored more efficiently
  • No need to store null values for dropouts
  • Easier to review data
– There are, however, no hard and fast rules

Speaker notes:

Now if you took the time to look at the original data set, you would see that it is stored as a single worksheet in Excel. It would be very easy to read this entire worksheet into a single table in a SQL database. I took the extra time and trouble to split it into separate tables because it made for an interesting teaching example, but there is more to it than just that.

You will often find longitudinal data stored in separate tables out in the real world. There are several reasons for this.

First, the data may come from separate sources. The demographic data, for example, might come from a REDCap survey while the measurements at baseline, three months, and one year might come from the electronic health records. It is simpler in this case to import data from separate sources into separate tables and do the joining later.

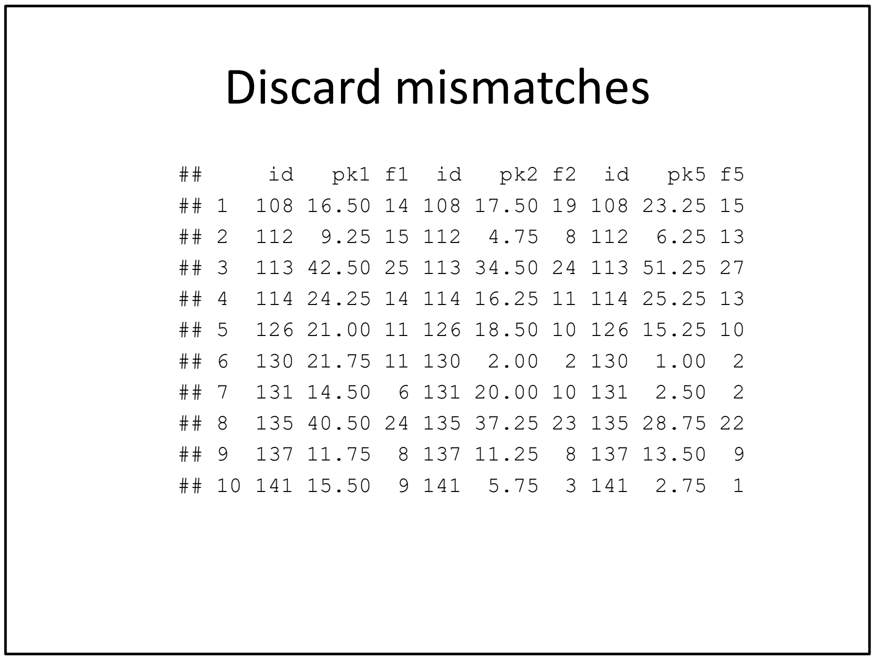
The data might be completed at different times. That's very likely because of the nature of longitudinal data. So you might have a database that starts out with the demographics and baseline values. When the three month data arrives, rather than trying to link the data up and possibly get things misaligned, you may prefer to put that in a fresh table. Then you might do the same for the one year data. It seems simpler to enter the new data fresh rather than trying to update the existing table that has demographics and baseline values.

If you put everything in a single table, then partway through the study, you would have null values that could simultaneously represent two different things. They could be placeholders that tell you that this data has not arrived yet or has arrived but has not been entered. But it could also represent patients who have dropped out. Having a null value possibly represeting two very different cases is asking for trouble.

There is also an efficiency issue. When a patient drops out of the study, you don’t have to keep storing information for that patient. With a dropout rate of 50%, your later tables would be half the size. Now that’s not quite accurate, because you need to keep the keys that link the tables together and they take up a bit of extra space. Also, as cheap as computer storage is these days, there is less need to worry about storage issues.

But efficiency here could mean efficiency in reviewing your data. The “holes” that dropouts create in your database make it harder to scan through your data.

Now, there are no right or wrong answers here. The choices on how to store data are very much dependent on the context of the data itself. Two very intelligent people might chose two very different database designs and neither one would be wrong. Choices like this are sometimes informed by rules relating to efficiency, but these rules are very flexible.



## Discard mismatches

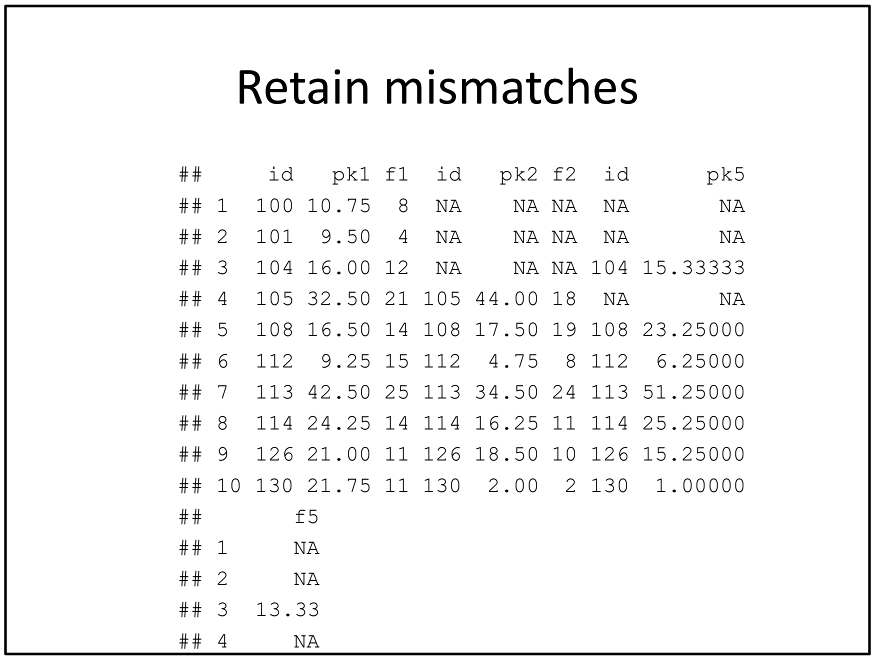
```
##     id   pk1 f1  id   pk2 f2  id   pk5 f5
## 1  108 16.50 14 108 17.50 19 108 23.25 15
## 2  112  9.25 15 112  4.75  8 112  6.25 13
## 3  113 42.50 25 113 34.50 24 113 51.25 27
## 4  114 24.25 14 114 16.25 11 114 25.25 13
## 5  126 21.00 11 126 18.50 10 126 15.25 10
## 6  130 21.75 11 130  2.00  2 130  1.00  2
## 7  131 14.50  6 131 20.00 10 131  2.50  2
## 8  135 40.50 24 135 37.25 23 135 28.75 22
## 9  137 11.75  8 137 11.25  8 137 13.50  9
## 10 141 15.50  9 141  5.75  3 141  2.75  1
```

Speaker notes:

If you discarded any mismatches, this is what your data would look like. Notice that the subject id starts at 108 because ids for 100, 101, 104, and 105 have mismatch either at three months or at one year.

There is a term in statistics, complete case analysis, that represents this situation. If a patient drops out at any point in the study, even if they change their mind and come back, you exclude them from the analysis. From a research perspective, this is usually a bad idea because it is likely to produce a biased sample. It does have the advantage of simpllcity.

How you analyze longitudinal data with dropouts, of course, is beyond the scope of this class.

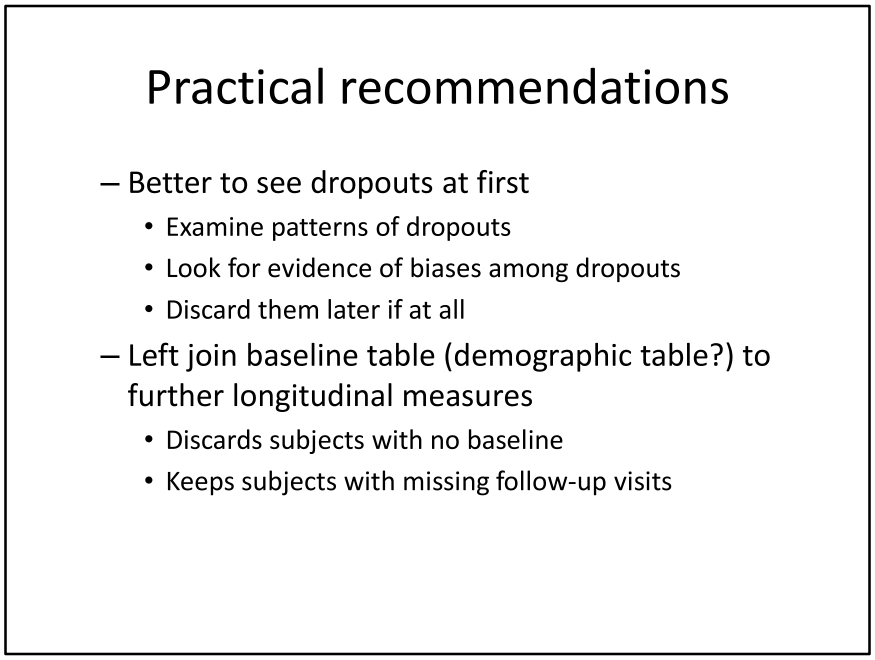


## Retain mismatches

```
##     id   pk1 f1  id   pk2 f2  id      pk5
## 1  100 10.75  8  NA    NA NA  NA       NA
## 2  101  9.50  4  NA    NA NA  NA       NA
## 3  104 16.00 12  NA    NA NA 104 15.33333
## 4  105 32.50 21 105 44.00 18  NA       NA
## 5  108 16.50 14 108 17.50 19 108 23.25000
## 6  112  9.25 15 112  4.75  8 112  6.25000
## 7  113 42.50 25 113 34.50 24 113 51.25000
## 8  114 24.25 14 114 16.25 11 114 25.25000
## 9  126 21.00 11 126 18.50 10 126 15.25000
## 10 130 21.75 11 130  2.00  2 130  1.00000
##       f5
## 1     NA
## 2     NA
## 3  13.33
## 4     NA
```

Speaker notes:

Here is the data where unmatched data is kept in, and the missing values caused by dropouts is represented by null values. Notice that subjects 100 and 101 drop out before the three month measurement. Subject 105 drops out after the three month measurement but before the one year measurement. Subject 104 is not really a drop out. There were in the study at the end of one year, but they must have missed their three month evaluation for some reason.



## Practical recommendations

- – Better to see dropouts at first
  - Examine patterns of dropouts
  - Look for evidence of biases among dropouts
  - Discard them later if at all
- – Left join baseline table (demographic table?) to further longitudinal measures
  - Discards subjects with no baseline
  - Keeps subjects with missing follow-up visits

Speaker notes:

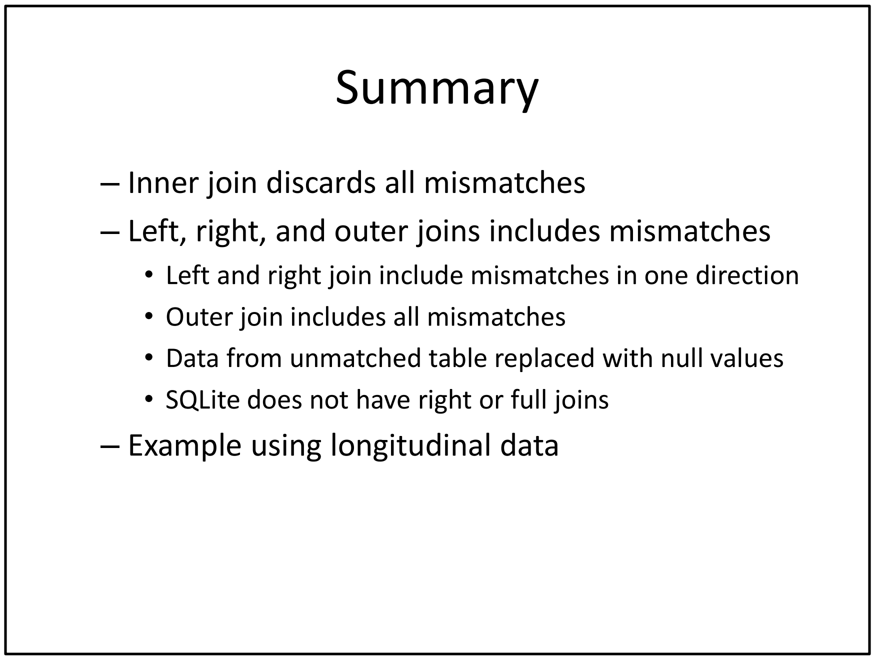
If you have longitudinal data, you almost certainly will have dropouts. Maybe a few, maybe a lot. It is generally best to see who the dropouts are, at least at first. You often want to examine patterns of dropouts. Are there, for example, patients who drop out for part of the study but then return, or is it the case that once you drop out, you never come back. The difference between these two cases has important implications on how you analyze your data.

Having the data on dropouts also allows for interesting investigations. Are there differences, for example, between the baseline values of those who dropped out and those who stayed in? Could it be that the dropouts were a lot sicker at baseline than those who stayed in? Are younger patients more likely to drop out?

If you decide to discard patients who dropped out, it is generally better to do so after you have carefully examined things. This means that you exclude not in the SQL code but in the R or SAS program.

In general, left joins from the baseline table to the other tables are better than other types of joins. A left join will insure that anyone who was in the study at baseline will be in your final query. Now a left join would exclude a patient who had no baseline, but did have a later visit. This would be unusual and it would be hard to justify keeping such a patient in the study. A patient who had no baseline values, for exmaple, would probably not have data in your demographics table and would probably not have information about what treatment group there were in.

In the rare case where you have demographics on a patient with no baseline values, you might do a left join on the demographic table. The important point, however, is that you almost always want to keep information about dropouts, and a left join guarantees that drop outs will not be discarded from your query.



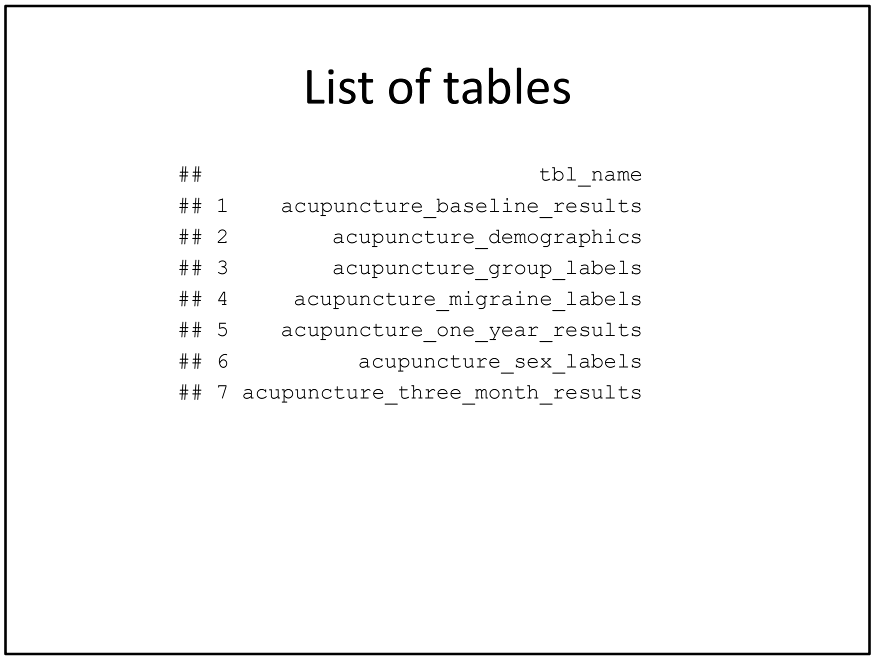
## Summary

– Inner join discards all mismatches
– Left, right, and outer joins includes mismatches
  - Left and right join include mismatches in one direction
  - Outer join includes all mismatches
  - Data from unmatched table replaced with null values
  - SQLite does not have right or full joins
– Example using longitudinal data

Speaker notes:

In this video, you saw simple examples of inner, left, right, and outer joins. Inner joins will discard any mismatched values. Left right and outer joins will include mismatches. Left and right joins are directional. A left join includes mismatches where you have data in the first database but not the second. A right join includes mismatches where you have data in the second database but not the first. A full join includes all mismatches. SQL will fill in null values for fields from the table without a match.

SQLite does not have a right join or a full join. The lack of a right join is not too troubling because you can always switch the order of the tables, but lack of a full join is a major limitation. As far as I know, SQLite is the only SQL database that does not have an outer join.

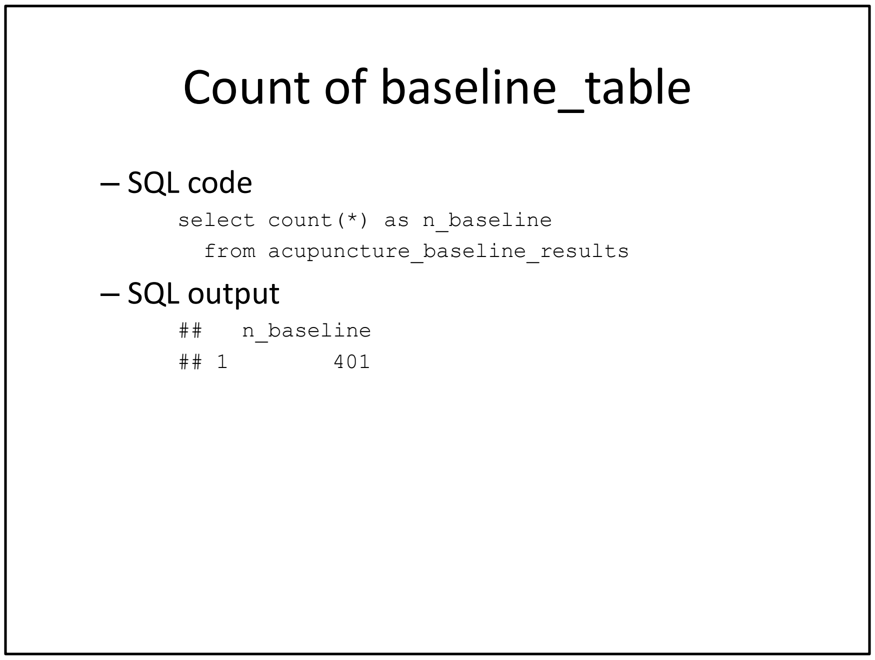


## List of tables

```
##                            tbl_name
## 1    acupuncture_baseline_results
## 2        acupuncture_demographics
## 3        acupuncture_group_labels
## 4     acupuncture_migraine_labels
## 5    acupuncture_one_year_results
## 6           acupuncture_sex_labels
## 7 acupuncture_three_month_results
```

Speaker notes:

The examples shown here will use the database longitudinal_examples_db. Here is a list of the tables in this database.



## Count of baseline_table

– SQL code

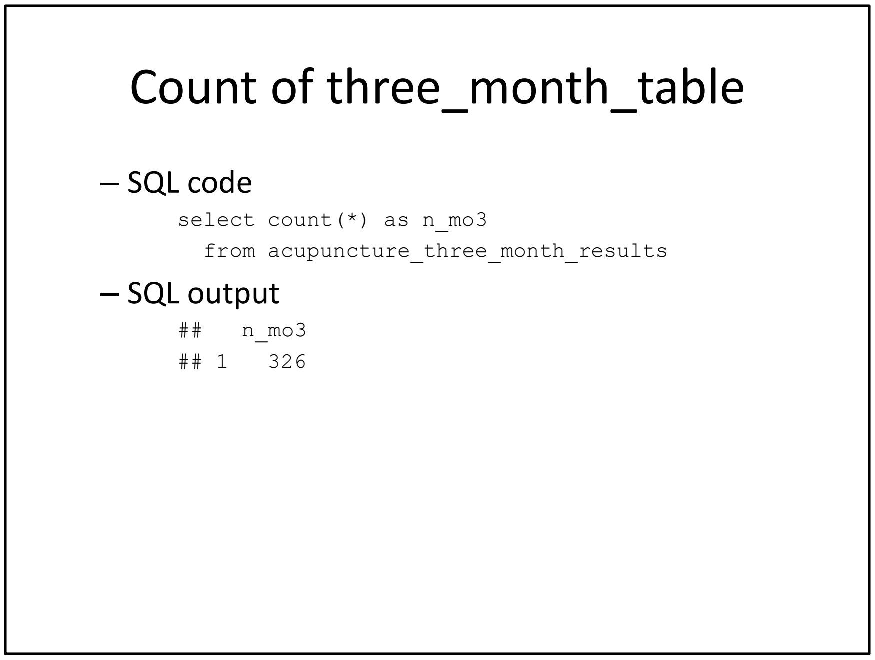
```
select count(*) as n_baseline
  from acupuncture_baseline_results
```

– SQL output

```
##   n_baseline
## 1        401
```

Speaker notes:

Here's the count of the number of records in baseline_table.



## Count of three_month_table

– SQL code

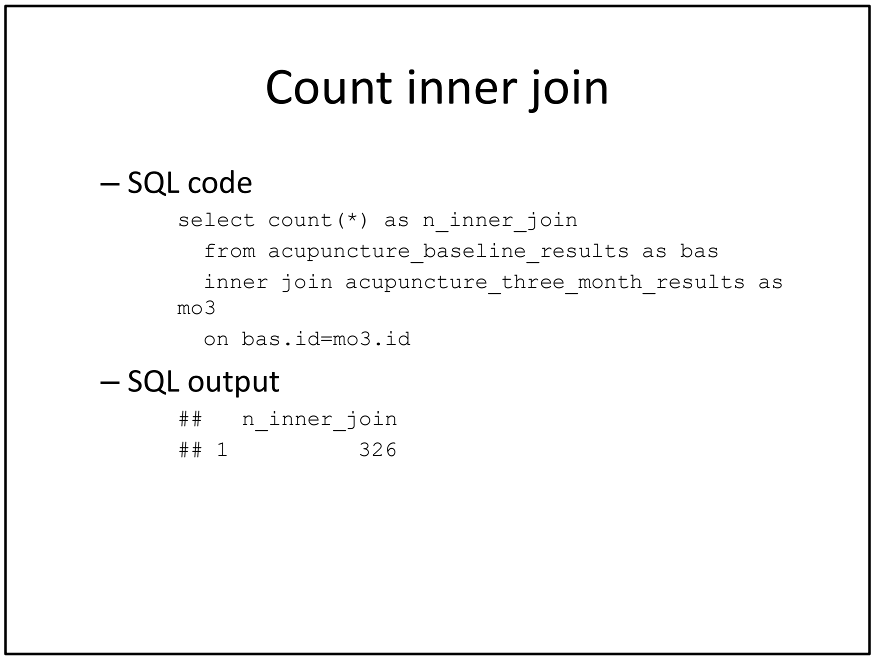
```
select count(*) as n_mo3
  from acupuncture_three_month_results
```

– SQL output

```
##   n_mo3
## 1   326
```

Speaker notes:

And here's the count of the number of records in three_month_table.



## Count inner join

– SQL code

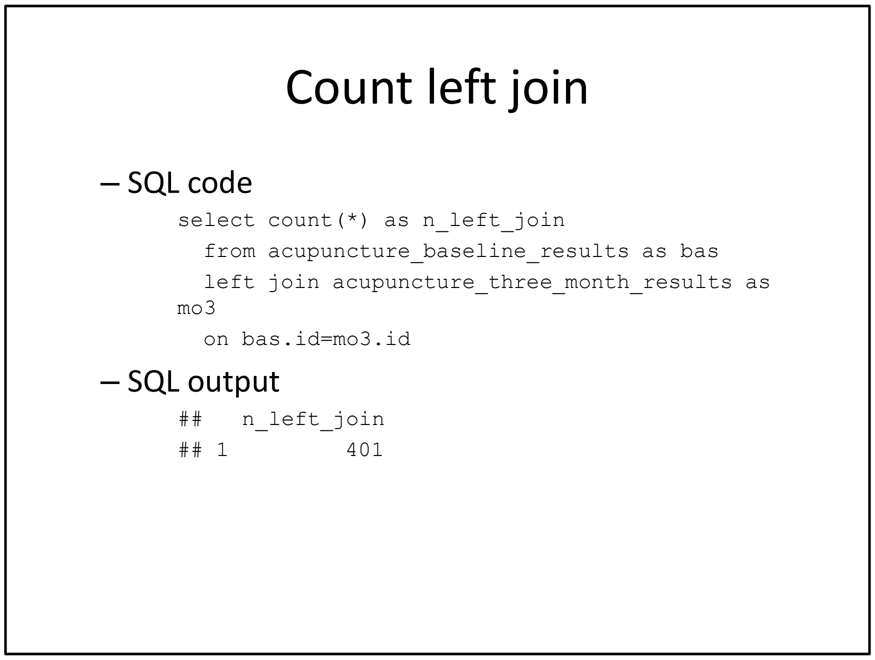
```
select count(*) as n_inner_join
  from acupuncture_baseline_results as bas
  inner join acupuncture_three_month_results as
mo3
  on bas.id=mo3.id
```

– SQL output

```
##   n_inner_join
## 1          326
```

Speaker notes:

HEre is the count of the inner join.



## Count left join

– SQL code

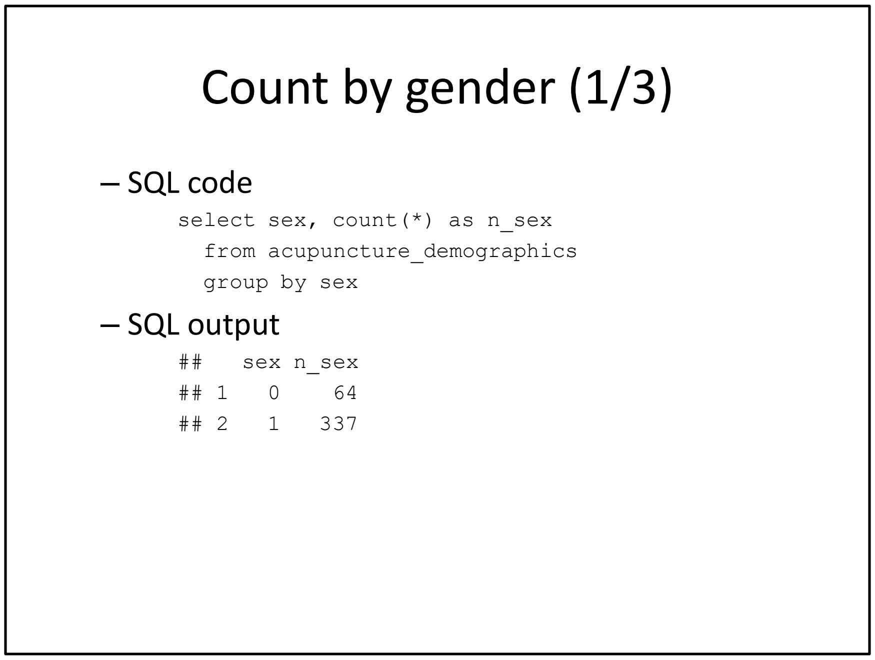
```
select count(*) as n_left_join
  from acupuncture_baseline_results as bas
  left join acupuncture_three_month_results as
mo3
  on bas.id=mo3.id
```

– SQL output

```
##   n_left_join
## 1         401
```

Speaker notes:

Here is the count of the left join.



## Count by gender (1/3)

– SQL code

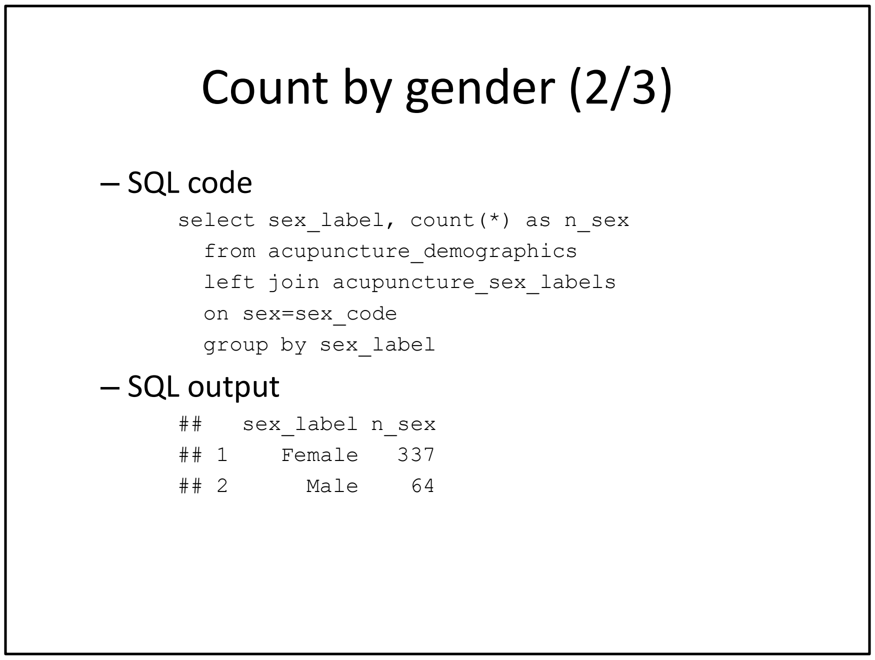
```
select sex, count(*) as n_sex
  from acupuncture_demographics
  group by sex
```

– SQL output

```
##   sex n_sex
## 1   0    64
## 2   1   337
```

Speaker notes:

Here is the count of males and females, but it uses the number code.



## Count by gender (2/3)

– SQL code

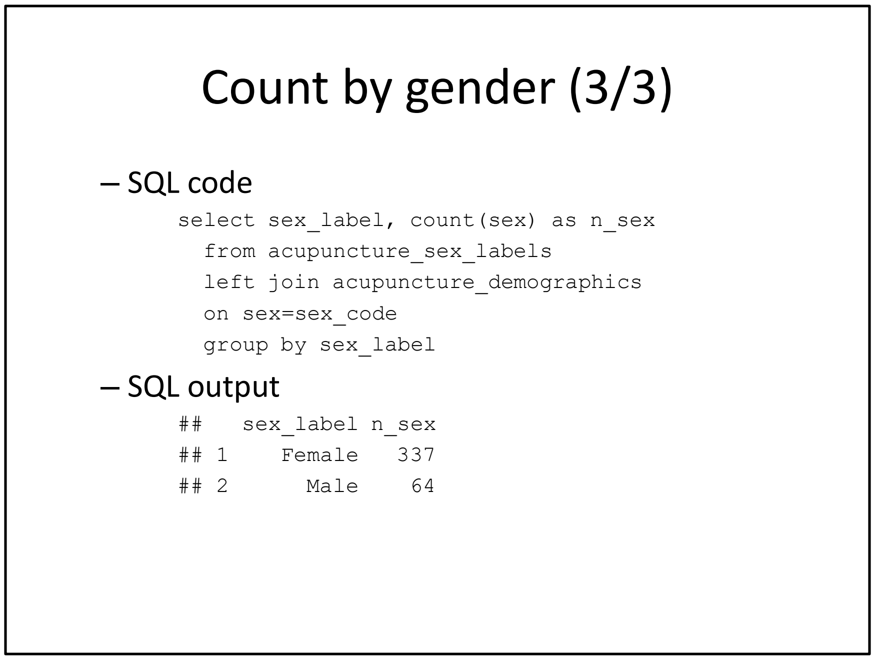
```
select sex_label, count(*) as n_sex
  from acupuncture_demographics
  left join acupuncture_sex_labels
  on sex=sex_code
  group by sex_label
```

– SQL output

```
##   sex_label n_sex
## 1    Female   337
## 2      Male    64
```

Speaker notes:

Here is the count of males and females, using the sex_name field that is only available after the merge.



## Count by gender (3/3)

– SQL code

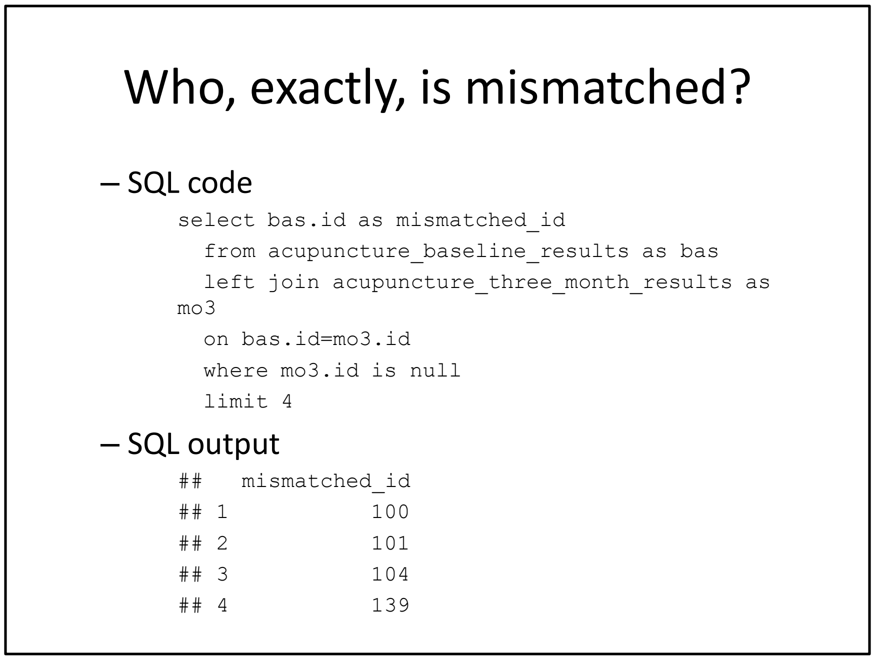
```
select sex_label, count(sex) as n_sex
  from acupuncture_sex_labels
  left join acupuncture_demographics
  on sex=sex_code
  group by sex_label
```

– SQL output

```
##   sex_label n_sex
## 1    Female   337
## 2      Male    64
```

Speaker notes:

There is an unmatched label (9=unknown) and if you wanted to include that in the table of counts, switch the order of tables and use count(sex) rather than count(*). There will be one row in the database for the mismatch, but since that will have a missing value for sex, the count function will not register for that record, producing a count of 0.



## Who, exactly, is mismatched?

– SQL code

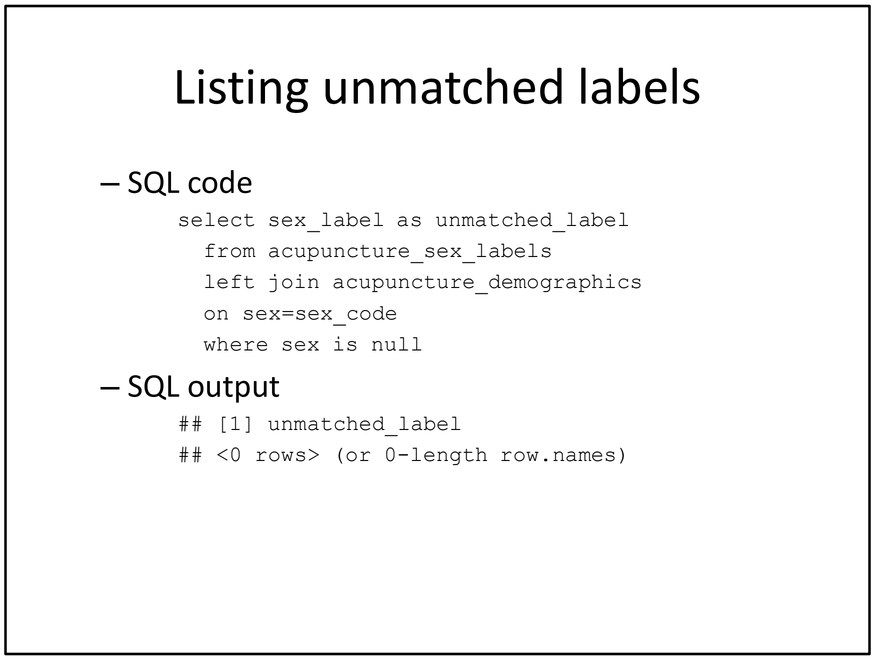
```
select bas.id as mismatched_id
  from acupuncture_baseline_results as bas
  left join acupuncture_three_month_results as
mo3
  on bas.id=mo3.id
  where mo3.id is null
  limit 4
```

– SQL output

```
##   mismatched_id
## 1           100
## 2           101
## 3           104
## 4           139
```

Speaker notes:

If you want a list of ids where two tables fail to match, look for a null value in one of the tables. There are 75 ids, so I could not print out all of them on this slide.



## Listing unmatched labels

– SQL code

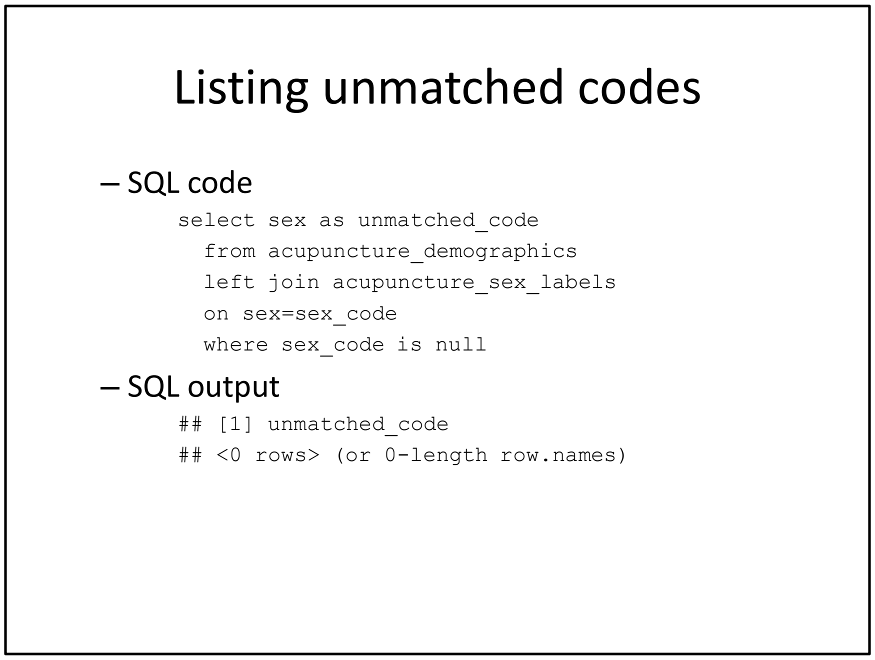
```
select sex_label as unmatched_label
  from acupuncture_sex_labels
  left join acupuncture_demographics
  on sex=sex_code
  where sex is null
```

– SQL output

```
## [1] unmatched_label
## <0 rows> (or 0-length row.names)
```

Speaker notes:

You can use the same approach to look for unmatched labels or unmatched codes. Here is ab example of looking for unmatched labels. There are no unmatched labels, so SQL returns 0 rows.



## Listing unmatched codes

– SQL code

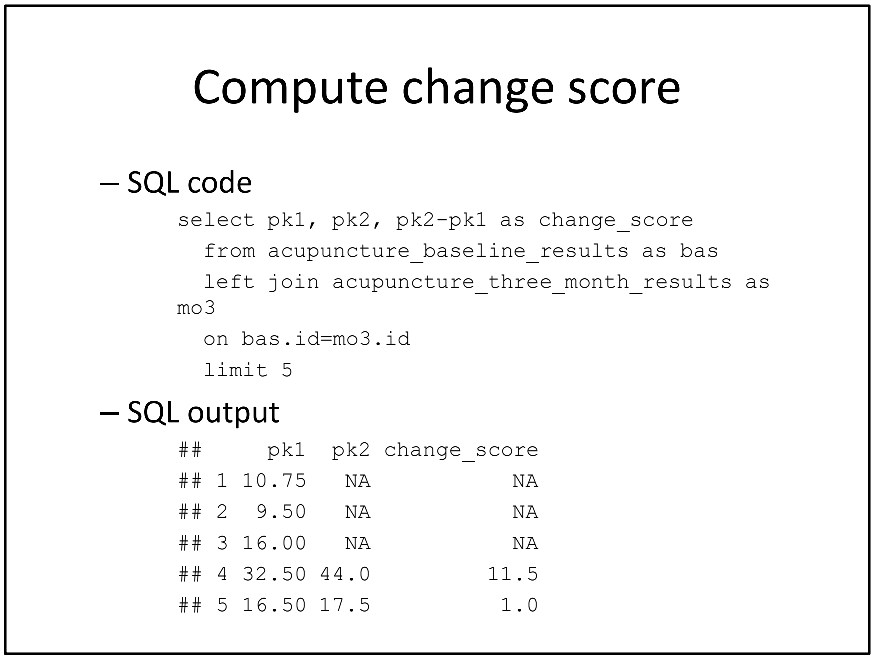
```
select sex as unmatched_code
  from acupuncture_demographics
  left join acupuncture_sex_labels
  on sex=sex_code
  where sex_code is null
```

– SQL output

```
## [1] unmatched_code
## <0 rows> (or 0-length row.names)
```

Speaker notes:

By swapping the order of tables (or changing to a right join) and making a few other small changes, you can find codes in the demography database that do not have a corresponding label. There are no unmatched codes, so SQL returns 0 rows.



## Compute change score

– SQL code

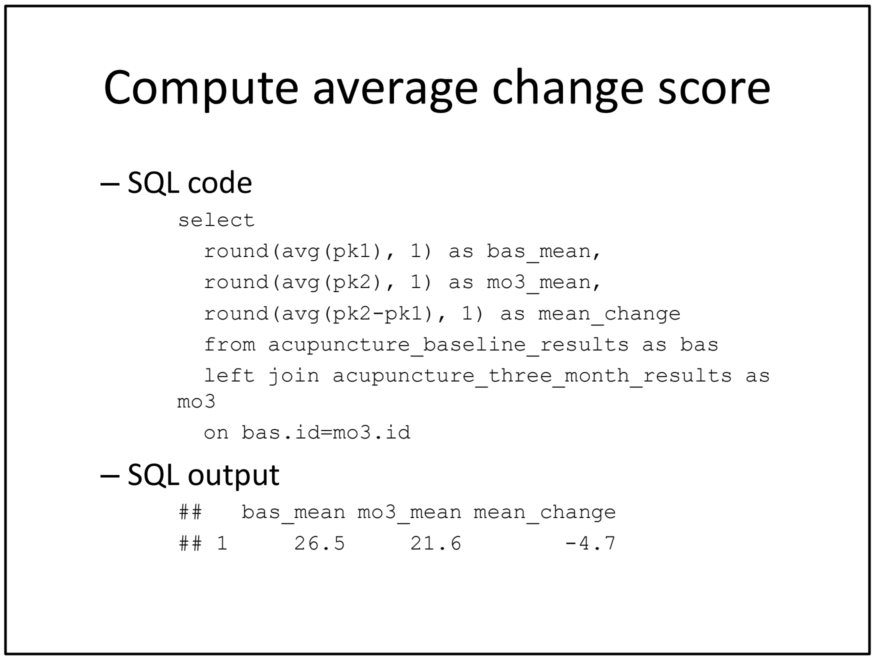
```
select pk1, pk2, pk2-pk1 as change_score
  from acupuncture_baseline_results as bas
  left join acupuncture_three_month_results as
mo3
  on bas.id=mo3.id
  limit 5
```

– SQL output

```
##     pk1  pk2 change_score
## 1 10.75   NA           NA
## 2  9.50   NA           NA
## 3 16.00   NA           NA
## 4 32.50 44.0         11.5
## 5 16.50 17.5          1.0
```

Speaker notes:

You can compute values across different tables. This example shows a change score. Notice that the change score is null if either value is null.



## Compute average change score

– SQL code

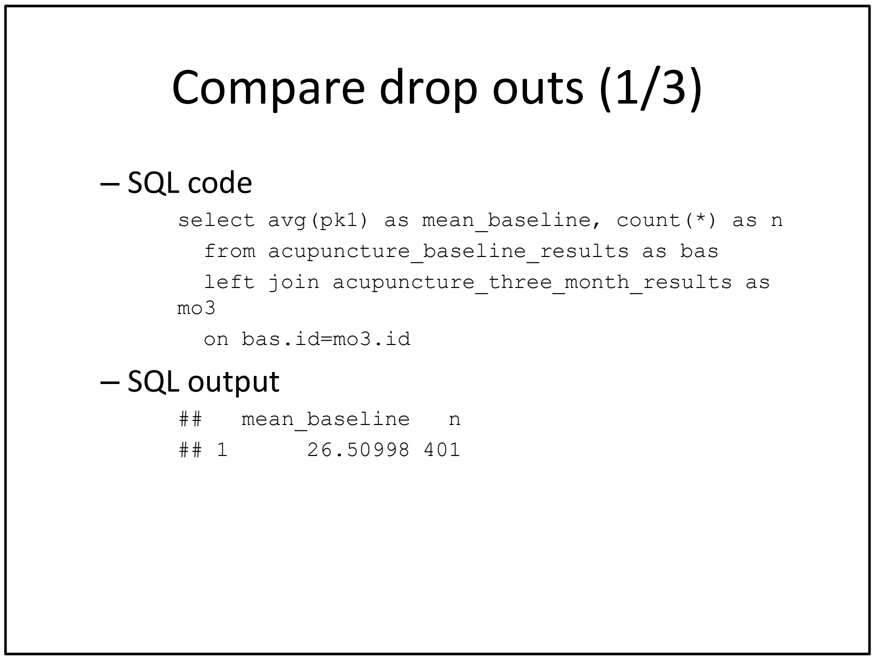
```
select
  round(avg(pk1), 1) as bas_mean,
  round(avg(pk2), 1) as mo3_mean,
  round(avg(pk2-pk1), 1) as mean_change
  from acupuncture_baseline_results as bas
  left join acupuncture_three_month_results as
mo3
  on bas.id=mo3.id
```

– SQL output

```
##   bas_mean mo3_mean mean_change
## 1     26.5     21.6        -4.7
```

Speaker notes:

You can even average these values. Notice that the mean change is not exactly equal to the difference in the two means. That would normally be the case, but remember that the baseline average is an average that includes 75 patients wihtout a corresponding three month value. You could restrict the data so that the baseline average is only computed for those patients who have a matching three month value. You could do this using an inner join or by restricting the data to cases where pk2 is not null.



## Compare drop outs (1/3)

– SQL code

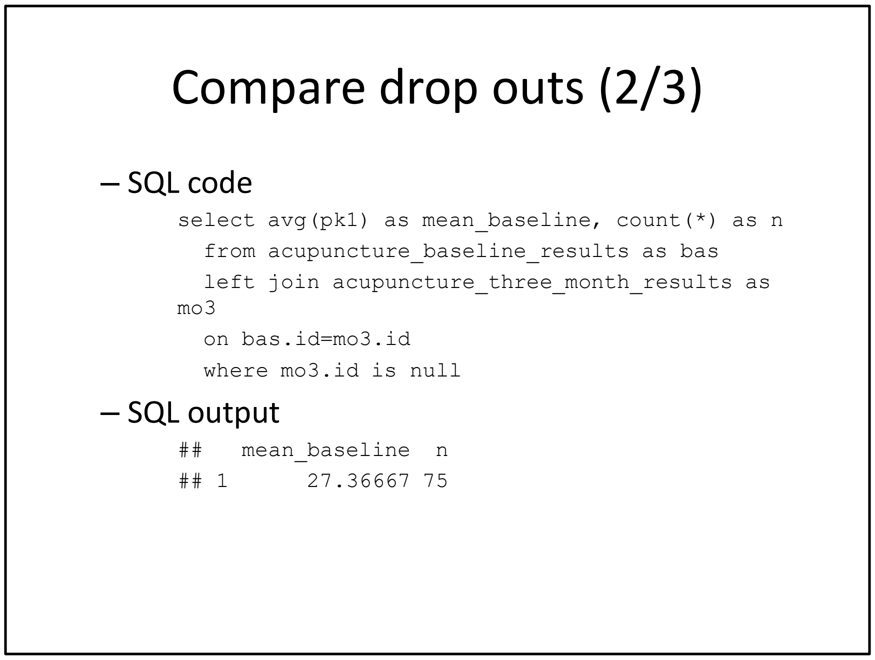
```
select avg(pk1) as mean_baseline, count(*) as n
  from acupuncture_baseline_results as bas
  left join acupuncture_three_month_results as
mo3
  on bas.id=mo3.id
```

– SQL output

```
##   mean_baseline   n
## 1      26.50998 401
```

Speaker notes:

An important comparison is what the baseline values look like for the drop outs compared to those who stay in the study. This code shows how to get an overall mean baseline score.



## Compare drop outs (2/3)

– SQL code

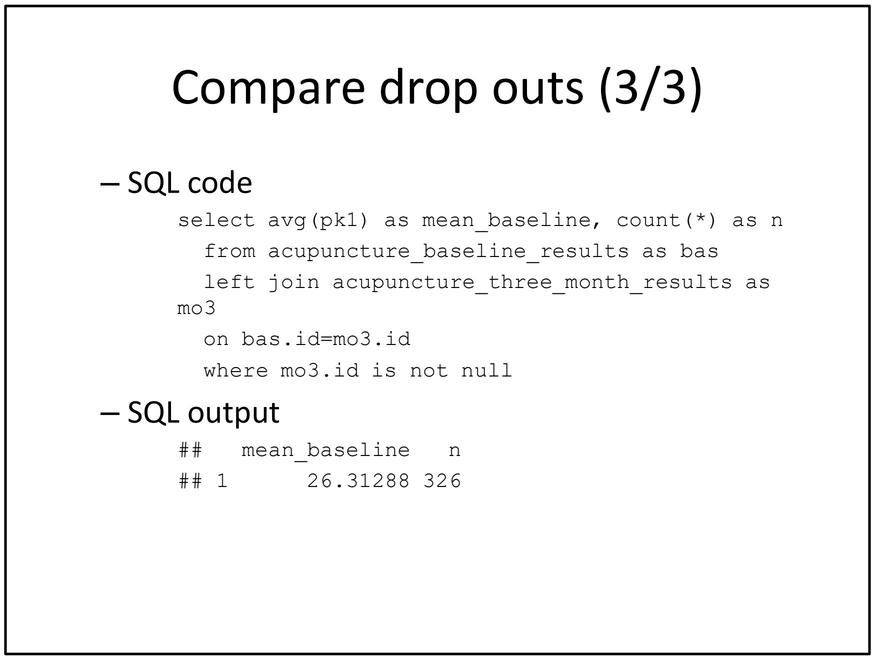
```
select avg(pk1) as mean_baseline, count(*) as n
  from acupuncture_baseline_results as bas
  left join acupuncture_three_month_results as
mo3
  on bas.id=mo3.id
  where mo3.id is null
```

– SQL output

```
##   mean_baseline  n
## 1      27.36667 75
```

Speaker notes:

Here is the mean for those who dropped out.



## Compare drop outs (3/3)

– SQL code

```
select avg(pk1) as mean_baseline, count(*) as n
  from acupuncture_baseline_results as bas
  left join acupuncture_three_month_results as
mo3
  on bas.id=mo3.id
  where mo3.id is not null
```

– SQL output

```
##   mean_baseline   n
## 1      26.31288 326
```

Speaker notes:

Here is the mean for those who did not drop out.

## Summary

- Counts are very important
  - Before AND after joins
- How to select ids of mismatched values
- How to find mismatched category labels
- How to compute change scores
- Comparing averages of drop outs to others