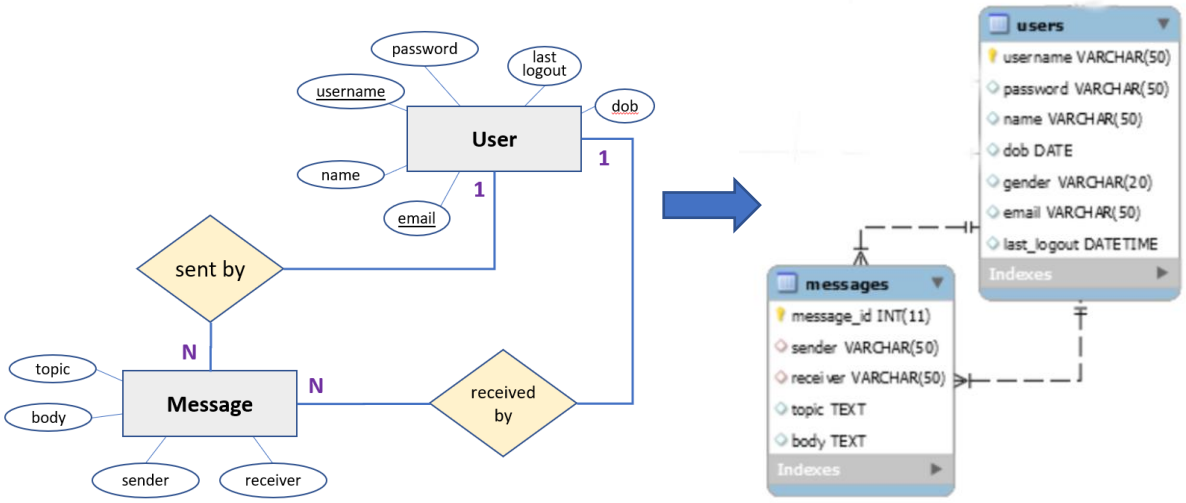
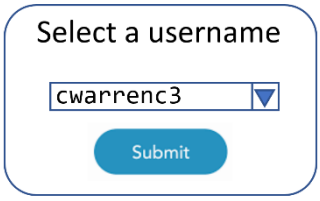
Views - A virtual abstraction layer that protects your applications from change over time

To get a handle on the chaos that can result when a database's physical structure changes, let's consider implementing a system where messages can be sent to only one user. Here it shows that a message can be received by only one user. When we transform this ER diagram, the below physical database structure may emerge.



Now, let's say we have a web application that allows one to select a username and see the full name of users who received messages from that account.



And the code behind this web page runs this query:

```
SELECT messages.message_id, messages.topic, messages.sender, users.name as recipient
FROM   jakeslist.messages
       JOIN jakeslist.users ON (messages.receiver = users.username)
WHERE  messages.sender = 'cwarren3c';
```

message_id	topic	sender	recipient
63	a libero nam	cwarren3c	Sharon Webb
135	tempus vel pede morbi porttitor lorem	cwarren3c	Scott Rose

But if we were to change the relationship "received by" from N:1 (message received by one user) to N:M (message can be received by many users) our physical model fails us. We need a bridging table. And we need to populate it with what used to be the primary key / foreign key pairings of `messages` and `users`.

```
CREATE TABLE jakeslist.received_messages as
  SELECT message_id, username
  FROM   jakeslist.messages
         JOIN jakeslist.users ON (messages.receiver = users.username);
```

(Note: To completely clean up the database, we would have to remove column "receiver" from `messages`.)

But now the query that finds cwarren3c's messages fails! There is no longer a column in `messages` called receiver. So the web page is broken until we replace the SQL code to this:

```
SELECT messages.message_id, messages.topic, messages.sender, users.name as recipient
FROM   jakeslist.messages
       JOIN jakeslist.received_messages USING (message_id)
       JOIN jakeslist.users USING (username)
WHERE  messages.sender = 'cwarren3c';
```

(Note: Since primary keys and foreign keys are now named the same, we can JOIN USING rather than JOIN ON)

Now our database allows multiple recipients of a message.

message_id	topic	sender	recipient
63	a libero nam	cwarren3c	Rose Collins
63	a libero nam	cwarren3c	Sharon Webb
135	tempus vel pede morbi porttitor lorem	cwarren3c	Scott Rose

If only we had used views from the start!

A view is a logical abstraction of a table which insulates the physical database structure from the users. Let's try this again.

Remember this...our original query with the N:1 relationship?

```
SELECT messages.message_id, messages.topic, messages.sender, users.name as recipient
FROM   jakeslist.messages
       JOIN jakeslist.users ON (messages.receiver = users.username)
WHERE  messages.sender = 'cwarren3c';
```

We could make a view that could be defined as a virtual table like this:

```
CREATE OR REPLACE VIEW jakeslist.received_messages_view AS
  SELECT messages.message_id, messages.sender, messages.receiver, messages.topic, messages.body,
         users.name as recipient, users.dob, users.gender, users.email, users.last_logout
  FROM   jakeslist.messages
         JOIN jakeslist.users ON (messages.receiver = users.username);
```

A few notes are in order:

- Note the syntax "CREATE OR REPLACE". This is a good practice. This code will CREATE the view if it does not already exist or replace it if it does.
- The view contains all the fields we want to expose to the end-users. It contains all of the fields from `messages` and most of the fields from `users`.
  - It did not contain the field `users.password`. This is an example of how you can hide fields from users for security purposes.
  - It did not contain the field `users.username` as this would have been redundant with `messages.receiver`
- It relabeled the field `users.name` to the more usefully named "recipient" using a column alias. When we did this, anyone who uses the view will think that "recipient" is a legitimate column. (But we know it is only virtual!)
  - We could have made a view that joined `messages` with `users` twice. Once to get the recipient's name and once to get the sender's name. If we had done that, we would HAVE to use column aliases to distinguish the names.

Now, not only has the abstracted view protected us from the underlying table structures, we now have what appears to be a table with both user and message fields. And our SQL is simpler to write now.

```
SELECT messages.message_id, messages.topic, messages.sender, users.name as recipient
FROM   jakeslist.messages
       JOIN jakeslist.users ON (messages.receiver = users.username)
WHERE  messages.sender = 'cwarren3c';
```

becomes

```
SELECT message_id, topic, sender, recipient
FROM   jakeslist.received_messages_view
WHERE  sender = 'cwarren3c';
```

And if we had to change our structure to move from N:1 to N:M, after making the physical database changes, all we need to do is redefine the view as below. But our query on the web page does not have to change at all!

```
CREATE OR REPLACE VIEW jakeslist.received_messages_view AS
  SELECT messages.message_id, messages.sender, messages.receiver, messages.topic, messages.body,
         users.name as recipient, users.dob, users.gender, users.email, users.last_logout
  FROM   jakeslist.messages
         JOIN jakeslist.received_messages USING (message_id)
         JOIN jakeslist.users USING (username);
```

Moral of the story: Always expose views to your users rather than tables whenever you can.