>>> A crash course in SQL
>>> New Zealand Social Statistics Network

Daniel Fryer [†]

Nov, 2020

[†]daniel@vfryer.com

>>> Overview

Day 1

1. Introduction
2. The relational model
3. Tables and relationships
4. Connect to a database
5. Basic SQL retrieval

   Lunch!

6. Search conditions
7. Subqueries
8. Basic exercises
9. Joining and join exercises

   Go home and read the notes

Page is hyperlinked: click a topic above to jump to it.

>>> Overview

Day 2

1. Revision of day 1
2. Reading the docs
3. Aggregating
4. Big reveal and exercises

   Lunch!

5. Creating and editing tables
6. The Integrated Data Infrastructure
7. Putting it all together
8. Connecting and exporting

   Send me questions and give feedback

Click here to find the day 2 slides.

>>> Daily schedule

Session 1 09:00am - 10:30am
morning tea (15min)
Session 2 10:45am -12:30pm
lunch (1 hour)
Session 3 01:30pm - 03:00pm
afternoon tea (15 min)
Session 4 03:15pm - 04:30pm

>>> How to pronounce SQL

* S. Q. L. (Structured Query Language)
* 'SEQUEL' (Structured English Query Language)

We will be using Microsoft's Transact-SQL (T-SQL)

>>> A little about yourself

* What is your name?
* What is your favourite colour? (or NULL)
* What would you like to get out of this course?

>>> Show of hands

Past experience

>>> The Kahoots!

Definition

A Kahoot is a fun quiz thing that we'll do at the end of most sessions. Join in to test your skills. Use the same nickname every time if you want to join the leaderboard.

And now for a practice Kahoot…

>>> Where are we now?

Day 1

1. Introduction
2. The relational model
3. Tables and relationships
4. Connect to a database
5. Basic SQL retrieval

   Lunch!

6. Search conditions
7. Subqueries
8. Basic exercises
9. Joining and join exercises

   Go home and read the notes

Page is hyperlinked: click a topic above to jump to it.

>>> Let the learning begin

A Relational Database Management System (RDBMS).

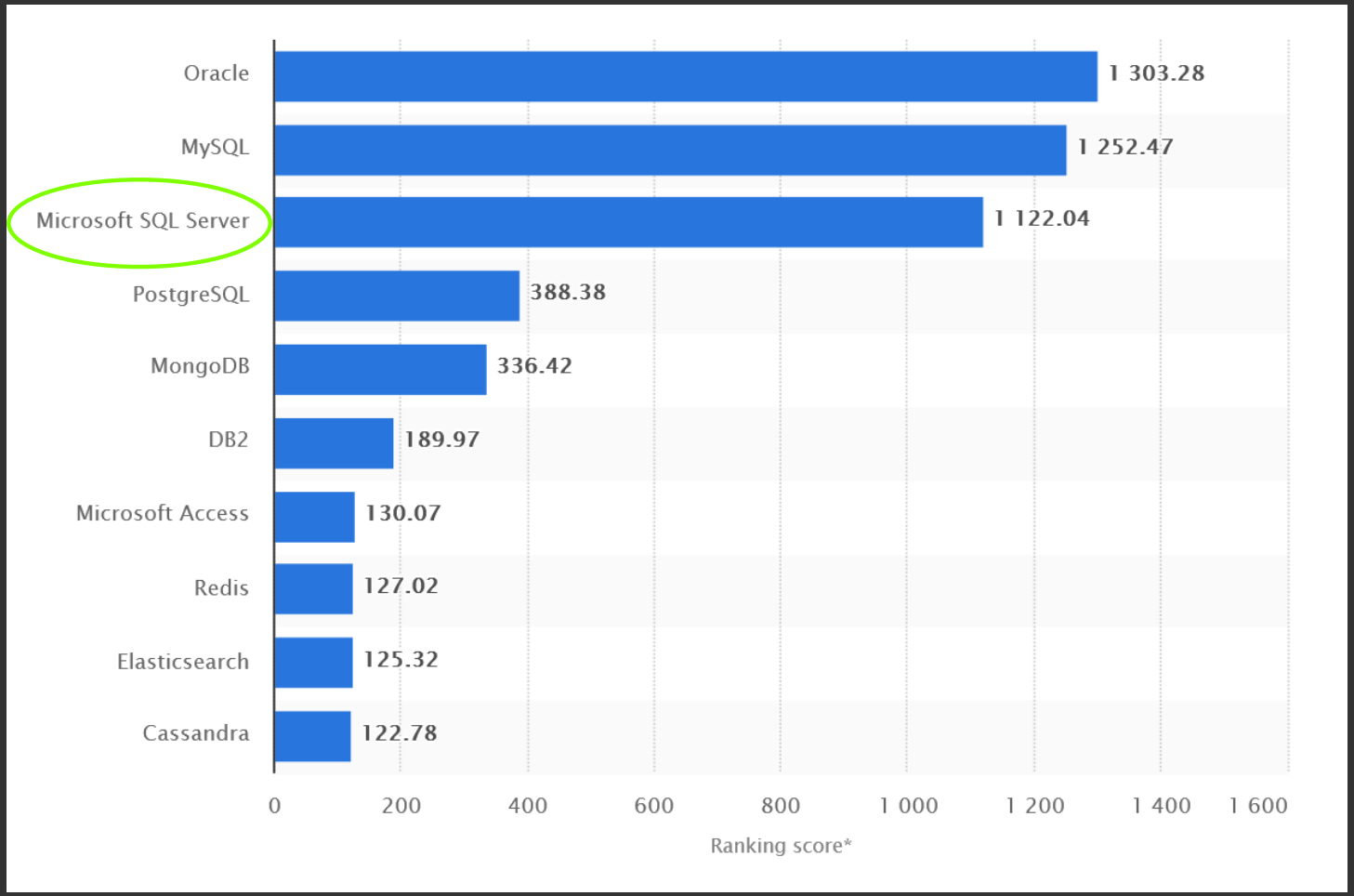
Definition

Kind of complicated. A DBMS is a large collection of interdependent programs all working together to define, construct, manipulate, protect and otherwise manage a database. An RDBMS is the most popular kind of DBMS.

SQL is a programming language for talking to your RDBMS.

>>> RDBMS

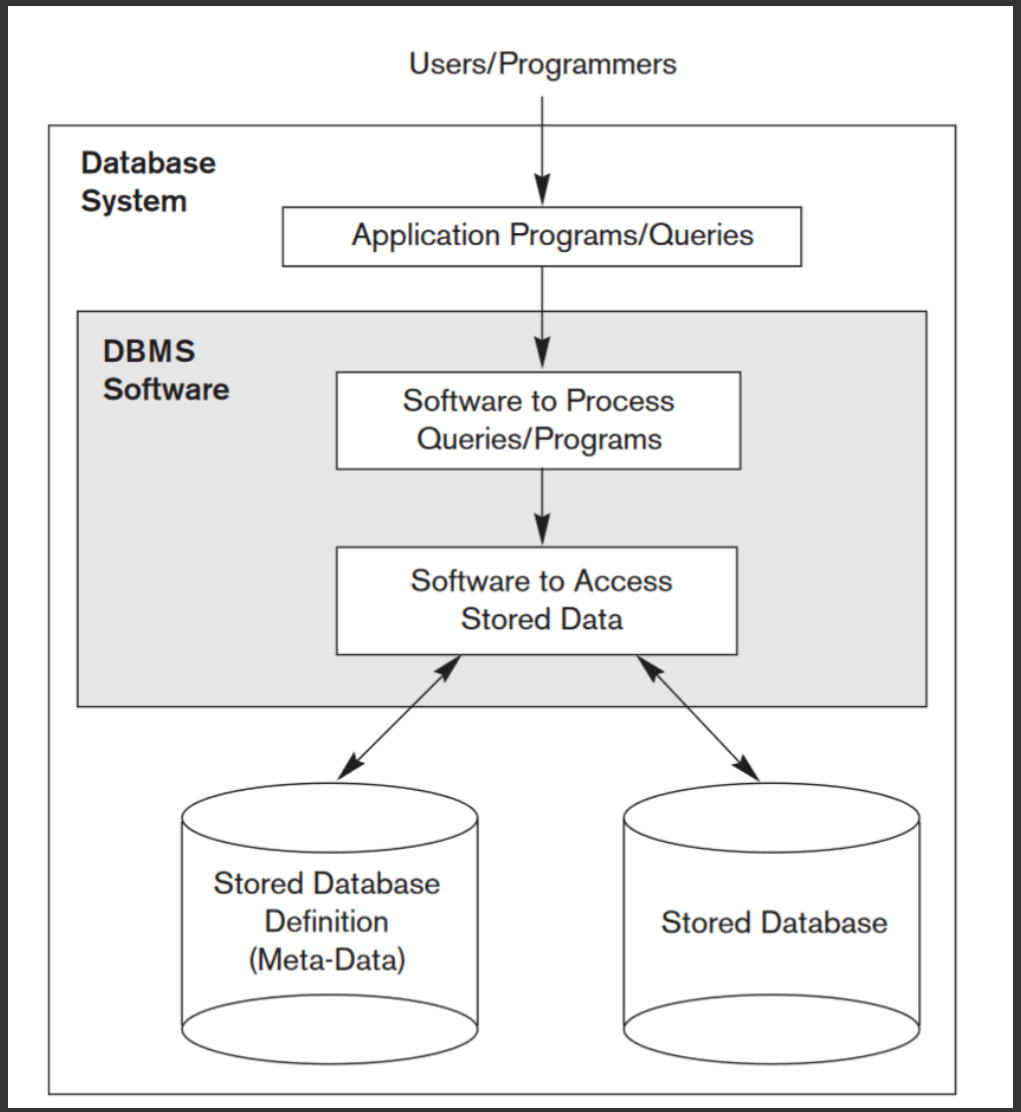
The most popular Relational Database Management Systems



Source: statista.com

>>> RDBMS

A layer of abstraction between human and machine



>>> DBMS

Grandfather of SQL and RDBMS, in the 1970s:

> *''Future users of databases should be protected from having to know how the data is organised in the machine.'' - Ted Codd (IBM researcher).*

To talk to humans and machines, the RDBMS should have a model of the world that is intuitive to both. This model is called the Relational Model.

Definition

The Relational Model is the 'common tongue' between the humans and the machines. It has a nice formal mathematical definition, so it is easy for machines to work with.
For the humans, it has a simple intuitive description in terms of tables and relationships between tables!

>>> Our very first table

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

>>> What's the takeaway from all this??

When using SQL, you'll always be working with tables. This is (deceptively) simple and intuitive. Underlying that, there is a really powerful system that let's you talk to the machine in a fairly ideal way. This makes SQL very efficient.

The tradeoff? Some parts of SQL will be really simple and intuitive. Others can at first be frustrating and confusing. A little practice goes a looooooong way.

>>> The anatomy of a table

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

>>> The anatomy of a table

Table name

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

>>> The anatomy of a table

Row (record)

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

>>> The anatomy of a table

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

Column (attribute)

>>> The anatomy of a table

Column names (attribute names)

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

>>> The anatomy of a table

Primary key	Friends		
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

>>> The anatomy of a table

Primary key	Friends		
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

* Every table should have a primary key
* No two rows can have the same entry
* There must be no NULL entries

>>> One more thing: The data types of attributes

Friends(FriendID, FirstName, LastName, FavColour)

>>> One more thing: The data types of attributes

Friends(FriendID, FirstName, LastName, FavColour)

int

Definition

An integer is a positive or negative whole number.

>>> One more thing: The data types of attributes

Definition

Varchar stands for 'variable length character.'
It is a string of characters of undetermined length.

>>> Where are we now?

Day 1

1. Introduction
2. The relational model
3. Tables and relationships
4. Connect to a database
5. Basic SQL retrieval

   Lunch!

6. Search conditions
7. Subqueries
8. Basic exercises
9. Joining and join exercises

   Go home and read the notes

Page is hyperlinked: click a topic above to jump to it.

>>> What are relationships between tables?

>>> What are relationships between tables?

>>> Relationships between tables overview

1. One-to-many relationships
2. Primary and foreign keys
3. Many-to-many relationships
4. One-to-one relationships

>>> One-to-many relationships

* For each car there are *many* wheels.

>>> One-to-many relationships

* For each car there are *many* wheels.

>>> One-to-many relationships

* For each car there are *many* wheels.
  But each wheel belongs to only *one* car.

>>> One-to-many relationships

* For each car there are *many* wheels.
  But each wheel belongs to only *one* car.
* One bank can have *many* accounts.
  But each account belongs to *one* bank.

>>> One-to-many relationships

* For each friend there are *many* pets.
  But each pet belongs to only *one* friend.

Where do we put the extra pets?

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

>>> One-to-many relationships

* For each friend there are *many* pets.
  But each pet belongs to only *one* friend.

Where do we put the extra pets?

Friends				
FriendID	FirstName	…	PetName_1	PetName_2
1	X	…	NULL	NULL
2	Y	…	Chikin	NULL
3	Z	…	Cauchy	Gauss

>>> Problems with putting them in the same table

Ideas?

Friends				
FriendID	FirstName	…	PetName_1	PetName_2
1	X	…	NULL	NULL
2	Y	…	Chikin	NULL
3	Z	…	Cauchy	Gauss

>>> Problems with putting them in the same table

* Have to store NULL in every entry with no pet

>>> Problems with putting them in the same table

* Have to store NULL in every entry with no pet
* What if I meet a friend with 3+ pets? Many more NULLs

>>> Problems with putting them in the same table

* Have to store NULL in every entry with no pet
* What if I meet a friend with 3+ pets? Many more NULLs
* New one-to-many relationship between pets and toys?

>>> Problems with putting them in the same table

* Have to store NULL in every entry with no pet
* What if I meet a friend with 3+ pets? Many more NULLs
* New one-to-many relationship between pets and toys?
* Pets are tied to owners. Delete an owner → delete pets

>>> Problems with putting them in the same table

* Have to store NULL in every entry with no pet
* What if I meet a friend with 3+ pets? Many more NULLs
* New one-to-many relationship between pets and toys?
* Pets are tied to owners. Delete an owner $\rightarrow$ delete pets
* Ambiguity. Is information related to pets or owners?

>>> So what do we do instead?

>>> So what do we do instead?

Suspense.

The first Kahoot.

>>> What we do instead is…

Create another table.

>>> What we do instead is…

Create another table.

Pets			
PetID	PetName	PetDOB	FriendID
1	Chikin	24/09/2016	2
2	Cauchy	01/03/2012	3
3	Gauss	01/03/2012	3

>>> What we do instead is…

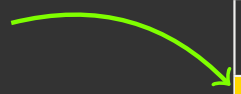
Create another table.

Pets			
PetID	PetName	PetDOB	FriendID
1	Chikin	24/09/2016	2
2	Cauchy	01/03/2012	3
3	Gauss	01/03/2012	3

Foreign key

>>> The foreign key 'points at' the primary key

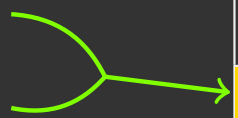
Pets			
PetID	PetName	…	FriendID
1	Chikin	…	2
2	Cauchy	…	3
3	Gauss	…	3



Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

>>> The foreign key 'points at' the primary key

Pets			
PetID	PetName	…	FriendID
1	Chikin	…	2
2	Cauchy	…	3
3	Gauss	…	3



Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Many

>>> The foreign key 'points at' the primary key

Pets			
PetID	PetName	…	FriendID
1	Chikin	…	2
2	Cauchy	…	3
3	Gauss	…	3

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

>>> Check that we fixed all these problems

* Have to store NULL in every entry with no pet
* What if I meet a friend with 3+ pets? Many more NULLs
* New one-to-many relationship between pets and toys?
* Pets are tied to owners. Delete an owner $\rightarrow$ delete pets
* Ambiguity. Is information related to pets or owners?

>>> Joining the tables

FriendsPets						
PetID	PetName	…	FriendID	FriendID	FirstName	…
1	Chikin	…	2	2	Y	…
2	Cauchy	…	3	3	Z	…
3	Gauss	…	3	3	Z	…

>>> Joining the tables

FriendsPets						
PetID	PetName	…	FriendID	FriendID	FirstName	…
1	Chikin	…	2	2	Y	…
2	Cauchy	…	3	3	Z	…
3	Gauss	…	3	3	Z	…

Primary/foreign key pair

>>> Group practice

1. Come up with a one-to-many relationship between either table (Friends or Pets) and something new
2. Make up 2 attributes for the new table (aside from the primary and foreign keys)
3. Make up 3 records for the new table
4. Draw up the two tables
5. Join the two tables

6. Challenge: Can you create a one-to-many relationship between Friends and Friends? How will you model it?

>>> A solution to the challenge question

* Game in which friends fight to the death. A friend can beat many others, but can only be beaten by one at most.

Friends				
FriendID	FirstName	LastName	FavColour	DefeatedByID
1	X	A	red	2
2	Y	B	blue	NULL
3	Z	C	NULL	2

>>> Primary and foreign keys

* Foreign key 'points at' the primary key
* Two rows CAN share same foreign key value
* Two rows CAN NOT share same primary key value
* Primary key can never be NULL
* All tables should have a primary key
* A PK or FK can be made of more than one column.

>>> Primary and foreign keys

* Foreign key 'points at' the primary key
* Two rows CAN share same foreign key value
* Two rows CAN NOT share same primary key value
* Primary key can never be NULL
* All tables should have a primary key
* A PK or FK can be made of more than one column.

For example, a company might sell group holiday packages and the primary key of their Customer table might be made of a GroupID and GroupMemberNumber.

>>> Many-to-many relationship

* A class has many students,
  and a student attends many classes
* A company has many investors,
  and an investor invests in many companies
* A person engages with many government departments,
  and a government department engages with many people

>>> Many-to-many relationship

* Each friend can scratch many backs,
  and a back can be scratched by many friends

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Scratched			
ScratcherID	Date	Time	ScratcheeID
1	05/09/2018	12:00pm	2
1	05/09/2018	12:30pm	3
2	06/09/2018	11:00am	1
3	07/09/2018	10:00am	1

>>> Many-to-many relationship

* Each friend can scratch many backs,
  and a back can be scratched by many friends

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Scratched			
ScratcherID	Date	Time	ScratcheeID
1	05/09/2018	12:00pm	2
1	05/09/2018	12:30pm	3
2	06/09/2018	11:00am	1
3	07/09/2018	10:00am	1

>>> Many-to-many relationship

* Each friend can scratch many backs,
  and a back can be scratched by many friends

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Scratched			
ScratcherID	Date	Time	ScratcheeID
1	05/09/2018	12:00pm	2
1	05/09/2018	12:30pm	3
2	06/09/2018	11:00am	1
3	07/09/2018	10:00am	1

>>> Many-to-many relationship

* Each friend can scratch many backs,
  and a back can be scratched by many friends

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Scratched			
ScratcherID	Date	Time	ScratcheeID
1	05/09/2018	12:00pm	2
1	05/09/2018	12:30pm	3
2	06/09/2018	11:00am	1
3	07/09/2018	10:00am	1

>>> Many-to-many relationship

* Each friend can scratch many backs,
  and a back can be scratched by many friends

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Friends		
FriendID	FirstName	…
1	X	…
2	Y	…
3	Z	…

Scratched			
ScratcherID	Date	Time	ScratcheeID
1	05/09/2018	12:00pm	2
1	05/09/2018	12:30pm	3
2	06/09/2018	11:00am	1
3	07/09/2018	10:00am	1

>>> Joining the tables

Friend_Scratched_Friend								
FrID	FriendName	…	SrID	…	SeID	FrID	FriendName	…
1	X	…	1	…	2	2	Y	…
1	X	…	1	…	3	3	Z	…
2	Y	…	2	…	1	1	X	…
3	Z	…	3	…	1	1	X	…

>>> Joining the tables

Friend_Scratched_Friend								
FrID	FriendName	…	SrID	…	SeID	FrID	FriendName	…
1	X	…	1	…	2	2	Y	…
1	X	…	1	…	3	3	Z	…
2	Y	…	2	…	1	1	X	…
3	Z	…	3	…	1	1	X	…

Pair 1

>>> Joining the tables

Friend_Scratched_Friend								
FrID	FriendName	…	SrID	…	SeID	FrID	FriendName	…
1	X	…	1	…	2	2	Y	…
1	X	…	1	…	3	3	Z	…
2	Y	…	2	…	1	1	X	…
3	Z	…	3	…	1	1	X	…

Pair 2

>>> Group practice (join)

* A friend can play with many pets,
  and a pet can play with many friends

Pets		
PetID	PetName	…
1	Chikin	
2	Cauchy	
3	Gauss	

Friends		
FriendID	FirstName	…
1	X	
2	Y	
3	Z	

PlayCount		
PetID	Count	FriendID
1	3	1
1	5	2
3	4	2

>>> One-to-one relationship

* A person can have at most one head,
  and each head belongs to only one person
* A table record has exactly one primary key value,
  and each primary key value belongs to exactly one record
* A user has one set of log-in details,
  and each set of log-in details belong to one user

>>> One-to-one relationship

* One friend can have at most one passport, and each passport belongs to only one friend

Friends					
FriendID	FirstName	…	PptCountry	PptNo	PptExpiry
1	X		Australia	E1321	12/03/2021
2	Y		New Zealand	LA123	01/09/2032
3	Z		Monaco	S9876	19/06/2028

>>> Why not keep one-to-one relationships in the same table?

>>> Why not keep one-to-one relationships in the same table?

* NULLs (many passport attributes? few people have them?)

>>> Why not keep one-to-one relationships in the same table?

* NULLs (many passport attributes? few people have them?)
* Dependence: Delete friend $\rightarrow$ delete passport.

>>> Goodbye, Mr. X

Friends					
FriendID	FirstName	…	PptCountry	PptNo	PptExpiry
2	Y		New Zealand	LA123	01/09/2032
3	Z		Monaco	S9876	19/06/2028

>>> Solution

>>> Solution

Passports			
PptNo	PptCountry	PptExpiry	FriendID
E1321	Australia	12/03/2021	NULL
LA123	New Zealand	01/09/2032	2
S9876	Monaco	19/06/2028	3

>>> Solution

Passports			
PptNo	PptCountry	PptExpiry	~~FriendID~~
E1321	Australia	12/03/2021	NULL
LA123	New Zealand	01/09/2032	2
S9876	Monaco	19/06/2028	3

Mr. X

>>> Any problems with this approach though?

Passports			
PptNo	PptCountry	PptExpiry	FriendID
E1321	Australia	12/03/2021	NULL
LA123	New Zealand	01/09/2032	2
S9876	Monaco	19/06/2028	3

>>> Any problems with this approach though?

Passports			
PptNo	PptCountry	PptExpiry	FriendID
E1321	Australia	12/03/2021	NULL
LA123	New Zealand	01/09/2032	2
S9876	Monaco	19/06/2028	3

Deleting a friend will delete the owner's name

>>> Any idea how to fix this?

We should avoiding keeping the person's name in both tables, since otherwise we have redundant data.

>>> Any idea how to fix this?

* Create 'people' table with binary variable for friend?

Definition

A binary variable is always either 0, 1 or NULL.
Usually, 0 represents false and 1 repesents true.

>>> Any idea how to fix this?

* Create 'people' table with binary variable for friend?
* Create separate tables for friends, enemies, etc…?

Leave it to the database designers.

>>> How a database design can restrict research

* Missing information
* Conflicting information (due to redundancy)
* Not enough levels of a categorical variable
* Binary answer when binary is not appropriate
* Hard to join the tables and connect records
* Hard to search for information in the database
* Many more, keep eyes open…

>>> The second Kahoot

Good luck

>>> Where are we now?

Day 1

1. Introduction
2. The relational model
3. Tables and relationships
4. Connect to a database
5. Basic SQL retrieval

   Lunch!

6. Search conditions
7. Subqueries
8. Basic exercises
9. Joining and join exercises

   Go home and read the notes

Page is hyperlinked: click a topic above to jump to it.

>>> Getting set up with SQL Server

Time for the real deal

Follow the guide (click here)
to connect SSMS or Azure Data Studio

>>> Task 1

In SSMS or Azure Data Studio, begin investigating the different tables and schemas in the PlayPen database: Expand the directory trees to figure out some table names and column names in the Notes schema and Ape schema. Be patient, the directory tree is slow to respond.

Bonus material:
Right click PlayPen, click 'New Query', then run this

```
SELECT *
FROM Information_schema.Tables
WHERE table_type = 'BASE TABLE';
```

>>> Task 2

Expand the Notes.Friends table directory in the directory tree, then expand the 'columns' directory. What do you see? Can you determine the data types of each column? Can you determine whether NULL values are allowed?

When a new table is created, the creator can decide whether to allow NULL values in each column. You can learn about data types and find the data types varchar and int in the T-SQL documentation (which we learn about soon).

>>> Task 3

Right click on the Notes.Friends table in the directory tree and click 'Select Top 1000'. A query is generated that selects the first 1000 rows, and the results are displayed.

Why are there square brackets around the table, schema, and column names in the query? What will happen if you remove the square brackets? Try it.

>>> Task 4

Change the query from the previous task to this:

```
SELECT *
FROM Notes.Friends;
```

Then, execute it. What does it do?

>>> Task 5

Write the following query.

```
SELECT *
FROM Notes.Pets;
```

Then, execute it. What does it do?

>>> Task 6

Write the following query.

```
SELECT *
FROM Notes.Friends, Notes.Pets
WHERE Notes.Friends.friendID = Notes.Pets.friendID;
```

Then, execute it. What does it do?

>>> A note on syntax

SeLeCt*FrOm[NoTeS].
[pEtS]rIpHaRaMbE20160528

---

* Upper/lower-case has no effect
* Spaces usually have no effect
* Square brackets can be omitted
* New lines have no effect
* Alias can be almost anything

So pay attention to style

The concept of an alias is explained on the next slide.

>>> A note on syntax

Aliases give temporary names to tables, and should be used to simplify and shorten your queries.

Without aliases:

```
SELECT *
FROM Notes.Friends, Notes.Pets
WHERE Notes.Friends.friendID = Notes.Pets.friendID;
```

With aliases:

```
SELECT *
FROM Notes.Friends F, Notes.Pets P
WHERE F.friendID = P.friendID;
```

From now on, we will always use aliases.

>>> A note on syntax

Another (optional) way to write aliases

```
SELECT *
FROM Notes.Friends AS F, Notes.Pets AS P
WHERE F.friendID = P.friendID;
```

>>> Where are we now?

Day 1

1. Introduction
2. The relational model
3. Tables and relationships
4. Connect to a database
5. Basic SQL retrieval

   Lunch!

6. Search conditions
7. Subqueries
8. Basic exercises
9. Joining and join exercises

   Go home and read the notes

Page is hyperlinked: click a topic above to jump to it.

>>> Time to learn SQL

We will now look closely at some basic clauses

>>> SQL clause: FROM

The FROM clause specifies table(s)
to access in the SELECT statement (and others).

```
FROM MySchema.MyTable MyAlias
```

The above will not run because there is no SELECT. You'll use FROM in almost every query, though.

>>> SQL clause: SELECT

The SELECT clause allows you to choose columns.
You can select all columns with SELECT *

We will look at the execution of this query:

```
SELECT F.firstName, F.favColour
FROM Notes.Friends F;
```

NB: The alias F seems to have been used before it was created! We will learn about the (sometimes confusing) SQL order of execution.

FROM Notes.Friends

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

```
SELECT F.FirstName, F.FavColour
```

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

result

Unnamed	
FirstName	FavColour
X	red
Y	blue
Z	NULL

Let's go back and see that again

>>> Order of execution

Let's go back and see that again

* Syntactic order of execution
* Logical order of execution
* Optimal order of execution

>>> SQL clause: WHERE

The WHERE clause allows you to choose rows,
using a search condition.

We will look at the execution of this query:

```
SELECT F.firstName, F.lastName
FROM Notes.Friends F
WHERE favColour = 'red';
```

FROM Notes.Friends

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

```
WHERE FavColour = 'red'
```

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

```
SELECT FirstName, LastName
```

Unnamed			
ID	FirstName	LastName	FavColour
1	*X*	*A*	red

result

Unnamed	
FirstName	LastName
X	A

>>> Order of execution

1. FROM
2. WHERE
3. SELECT

>>> Where are we now?

Day 1

1. Introduction
2. The relational model
3. Tables and relationships
4. Connect to a database
5. Basic SQL retrieval

   Lunch!

6. Search conditions
7. Subqueries
8. Basic exercises
9. Joining and join exercises

   Go home and read the notes

Page is hyperlinked: click a topic above to jump to it.

>>> Search conditions

Search conditions appear in a WHERE clause.
They make use of comparison operators and logical operators to check which rows match the conditions you specify.

>>> Search conditions

Search conditions appear in a WHERE clause.
They make use of comparison operators and logical operators
to check which rows match the conditions you specify.

Definition
A comparison operator is used to compare two things
and return true, false or NULL.
Click here to see examples in the docs.

>>> Search conditions

Search conditions appear in a WHERE clause.
They make use of comparison operators and logical operators
to check which rows match the conditions you specify.

Definition
Logical operators compare a number of things
and return true, false or NULL.
Click here to see examples in the docs.

>>> Comparison operators

* `WHERE FavColour = 'blue'` (equal)

>>> Comparison operators

* `WHERE FavColour = 'blue'` (equal)
* `WHERE FavColour <> 'blue'` (not equal)
* `WHERE FavColour != 'blue'` (also not equal)

>>> Comparison operators

```
*  WHERE  FavColour = 'blue'    (equal)
*  WHERE  FavColour <> 'blue'   (not equal)
*  WHERE  FavColour != 'blue'   (also not equal)
*  WHERE  Age > 35              (greater than)
*  WHERE  Year <= 1995          (less than or equal)
```

>>> Logical operators

* `WHERE FavColour IN ('blue', 'red', 'green')`

>>> Logical operators

* `WHERE FavColour IN ('blue', 'red', 'green')`
* `WHERE Age BETWEEN 25 AND 35`

>>> Logical operators

* `WHERE FavColour IN ('blue', 'red', 'green')`
* `WHERE Age BETWEEN 25 AND 35`
* `WHERE FirstName LIKE 'B%'`

>>> Logical operators

* `WHERE FavColour IN ('blue', 'red', 'green')`
* `WHERE Age BETWEEN 25 AND 35`
* `WHERE FirstName LIKE 'B%'`
* `WHERE FirstName LIKE '%B'`

>>> Logical operators

* `WHERE FavColour IN ('blue', 'red', 'green')`
* `WHERE Age BETWEEN 25 AND 35`
* `WHERE FirstName LIKE 'B%'`
* `WHERE FirstName LIKE '%B'`
* `WHERE FirstName LIKE '%b%'`

>>> Logical operators

* `WHERE FavColour IN ('blue', 'red', 'green')`
* `WHERE Age BETWEEN 25 AND 35`
* `WHERE FirstName LIKE 'B%'`
* `WHERE FirstName LIKE '%B'`
* `WHERE FirstName LIKE '%b%'`
* `WHERE FirstName LIKE '%[Bb]%'`

>>> Logical operators

AND				
true	AND	true	=	true
false	AND	true	=	false
true	AND	false	=	false
false	AND	false	=	false

OR				
true	OR	true	=	true
false	OR	true	=	true
true	OR	false	=	true
false	OR	false	=	false

NOT			
NOT	true	=	false
NOT	false	=	true

>>> Group practice

1. `(1 = 1) AND (2 = 1)`
2. `((1 = 1) AND (2 = 1)) OR (1 = 1)`
3. `('red' IN ('green', 'red')) AND ('red' LIKE 'r%')`
4. `NOT ( (1 = 1) AND (2 = 2) AND (3 = 3) )`

>>> Solution

1. `(1 = 1) AND (2 = 1)`
2. `((1 = 1) AND (2 = 1)) OR (1 = 1)`
3. `('red' IN ('green', 'red')) AND ('red' LIKE 'r%')`
4. `NOT (  (1 = 1) AND (2 = 2) AND (3 = 3)  )`

>>> Solution

1. `true AND (2 = 1)`
2. `((1 = 1) AND (2 = 1)) OR (1 = 1)`
3. `('red' IN ('green', 'red')) AND ('red' LIKE 'r%')`
4. `NOT (  (1 = 1) AND (2 = 2) AND (3 = 3)  )`

>>> Solution

1. `true AND false`
2. `((1 = 1) AND (2 = 1)) OR (1 = 1)`
3. `('red' IN ('green', 'red')) AND ('red' LIKE 'r%')`
4. `NOT (  (1 = 1) AND (2 = 2) AND (3 = 3)  )`

>>> Solution

1. `false`
2. `((1 = 1) AND (2 = 1)) OR (1 = 1)`
3. `('red' IN ('green', 'red')) AND ('red' LIKE 'r%')`
4. `NOT (  (1 = 1) AND (2 = 2) AND (3 = 3)  )`

>>> Solution

1. `false`
2. `false OR (1 = 1)`
3. `('red' IN ('green', 'red')) AND ('red' LIKE 'r%')`
4. `NOT (  (1 = 1) AND (2 = 2) AND (3 = 3)  )`

>>> Solution

1. `false`
2. `false OR true`
3. `('red' IN ('green', 'red')) AND ('red' LIKE 'r%')`
4. `NOT (  (1 = 1) AND (2 = 2) AND (3 = 3)  )`

>>> Solution

1. false
2. true
3. `('red' IN ('green', 'red')) AND ('red' LIKE 'r%')`
4. `NOT (  (1 = 1) AND (2 = 2) AND (3 = 3)  )`

>>> Solution

1. false
2. true
3. true AND ('red' LIKE 'r%')
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. `false`
2. `true`
3. `true AND true`
4. `NOT (  (1 = 1) AND (2 = 2) AND (3 = 3)  )`

>>> Solution

1. false
2. true
3. true
4. `NOT (  (1 = 1) AND (2 = 2) AND (3 = 3)  )`

>>> Solution

1. false
2. true
3. true
4. `NOT (  true AND (3 = 3)  )`

>>> Solution

1. false
2. true
3. true
4. NOT (true AND true)

>>> Solution

1. false
2. true
3. true
4. NOT (true)

>>> Solution

1. false
2. true
3. true
4. false

>>> A note on NULL

NULL				
(anything	AND	NULL)	=	NULL
(anything	OR	NULL)	=	NULL
(anything	=	NULL)	=	NULL

We will get practice with NULLs during the exercises.

>>> Where are we now?

Day 1

1. Introduction
2. The relational model
3. Tables and relationships
4. Connect to a database
5. Basic SQL retrieval

   Lunch!

6. Search conditions
7. Subqueries
8. Basic exercises
9. Joining and join exercises

   Go home and read the notes

Page is hyperlinked: click a topic above to jump to it.

>>> Basic subquery

Subqueries (next slide) are powerful with search conditions.
In fact, some logical operators only work with subqueries:

* EXISTS
* ALL
* ANY

Subqueries are also known as nested queries.

>>> Class practice

Can anyone figure out what this does?
Note: the subquery is executed first.

```
SELECT *
FROM Notes.Friends F
WHERE F.friendID IN (SELECT P.friendID
                     FROM Notes.Pets P);
```

>>> Solution

```
1. SELECT P.friendID FROM Notes.Pets P
```

>>> Solution

1. `SELECT P.friendID FROM Notes.Pets P`
   Retrieves a table of all the FriendIDs in Notes.Pets.

>>> Solution

1. `SELECT P.friendID FROM Notes.Pets P`
   Retrieves a table of all the FriendIDs in Notes.Pets.

2. Let's refer to the output of Step 1 as RESULT.

>>> Solution

1. `SELECT P.friendID FROM Notes.Pets P`
   Retrieves a table of all the FriendIDs in Notes.Pets.
2. Let's refer to the output of Step 1 as RESULT.
3. `SELECT * FROM Notes.Friends F WHERE F.FriendID IN RESULT`

>>> Solution

1. `SELECT P.friendID FROM Notes.Pets P`
   Retrieves a table of all the FriendIDs in Notes.Pets.

2. Let's refer to the output of Step 1 as RESULT.

3. `SELECT * FROM Notes.Friends F WHERE F.FriendID IN RESULT`
   Retrieves only the rows of Notes.Friends
   whose FriendID is in RESULT.

>>> Solution

1. `SELECT P.friendID FROM Notes.Pets P`
   Retrieves a table of all the FriendIDs in Notes.Pets.

2. Let's refer to the output of Step 1 as RESULT.

3. `SELECT * FROM Notes.Friends F WHERE F.FriendID IN RESULT`
   Retrieves only the rows of Notes.Friends
   whose FriendID is in RESULT.

It retrieved the details of all friends who have pets.

>>> Correlated subquery

This query achieves the same thing as the previous one. Do you notice anything strange about it? Can you figure out how it works?

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.friendID
              FROM Notes.Pets P
              WHERE P.friendID = F.friendID);
```

>>> Correlated subquery

This query achieves the same thing as the previous one. Do you notice anything strange about it? Can you figure out how it works?

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.friendID
              FROM Notes.Pets P
              WHERE P.friendID = F.friendID);
```

The alias here makes the subquery correlated.

>>> Correlated subquery

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.friendID
              FROM Notes.Pets P
              WHERE P.friendID = F.friendID);
```

We can think of this query's execution as follows:

>>> Correlated subquery

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.friendID
              FROM Notes.Pets P
              WHERE P.friendID = F.friendID);
```

We can think of this query's execution as follows:

1. First, grab a list of the 3 FriendID entries from Notes.Friends, giving FriendID_list = {1,2,3}.

>>> Correlated subquery

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.friendID
              FROM Notes.Pets P
              WHERE P.friendID = F.friendID);
```

We can think of this query's execution as follows:

1. First, grab a list of the 3 FriendID entries from Notes.Friends, giving FriendID_list = {1,2,3}.
2. For each element of FriendID_list, execute the subquery once, giving 3 results RESULT_list = {{},{2},{3,3}}.

>>> Correlated subquery

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.friendID
              FROM Notes.Pets P
              WHERE P.friendID = F.friendID);
```

We can think of this query's execution as follows:

1. First, grab a list of the 3 FriendID entries from Notes.Friends, giving FriendID_list = {1,2,3}.
2. For each element of FriendID_list, execute the subquery once, giving 3 results RESULT_list = {{},{2},{3,3}}.

Pets			
PetID	PetName	PetDOB	FriendID
1	Chikin	24/09/2016	2
2	Cauchy	01/03/2012	3
3	Gauss	01/03/2012	3

>>> Correlated subquery

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.friendID
              FROM Notes.Pets P
              WHERE P.friendID = F.friendID);
```

We can think of this query's execution as follows:

1. First, grab a list of the 3 FriendID entries from Notes.Friends, giving FriendID_list = {1,2,3}.
2. For each element of FriendID_list, execute the subquery once, giving 3 results RESULT_list = {{},{2},{3,3}}.
3. For each element of RESULT_list, the search condition EXISTS (RESULT) returns true if RESULT is not empty.

>>> Correlated subquery

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.friendID
              FROM Notes.Pets P
              WHERE P.friendID = F.friendID);
```

We can think of this query's execution as follows:

1. First, grab a list of the 3 FriendID entries from Notes.Friends, giving FriendID_list = {1,2,3}.
2. For each element of FriendID_list, execute the subquery once, giving 3 results RESULT_list = {{},{2},{3,3}}.
3. For each element of RESULT_list, the search condition EXISTS (RESULT) returns true if RESULT is not empty.
4. The final table contains each row of Notes.Friends that has a FriendID entry that returned true.

>>> Correlated subquery

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.friendID
              FROM Notes.Pets P
              WHERE P.friendID = F.friendID);
```

We can think of this query's execution as follows:

1. First, grab a list of the 3 FriendID entries from Notes.Friends, giving FriendID_list = {1,2,3}.
2. For each element of FriendID_list, execute the subquery once, giving 3 results RESULT_list = {{},{2},{3,3}}.
3. For each element of RESULT_list, the search condition EXISTS (RESULT) returns true if RESULT is not empty.
4. The final table contains each row of Notes.Friends that has a FriendID entry that returned true.

We will look at nested queries more later, after aggregation.

>>> The third Kahoot

Enjoy

>>> Where are we now?

Day 1

1. Introduction
2. The relational model
3. Tables and relationships
4. Connect to a database
5. Basic SQL retrieval

   Lunch!

6. Search conditions
7. Subqueries
8. Basic exercises
9. Joining and join exercises

   Go home and read the notes

Page is hyperlinked: click a topic above to jump to it.

>>> Exercises

Do Exercises Section 5.2.2

Click here to find the textbook.

>>> Where are we now?

Day 1

1. Introduction
2. The relational model
3. Tables and relationships
4. Connect to a database
5. Basic SQL retrieval

   Lunch!

6. Search conditions
7. Subqueries
8. Basic exercises
9. Joining and join exercises

   Go home and read the notes

Page is hyperlinked: click a topic above to jump to it.

>>> SQL query: JOIN

A JOIN (also known as an INNER JOIN) pairs the records from one table with the records from another table, using a primary/foreign key pair.

We will look at the execution of this query:

```
SELECT F.firstName, P.petName
FROM Notes.Friends F, Notes.Pets P
WHERE F.friendID = P.friendID;
```

>>> SQL query: JOIN

We will look at the execution of this query:

```
SELECT F.firstName, P.petName
FROM Notes.Friends F, Notes.Pets P
WHERE F.friendID = P.friendID;
```

Another way to write the same query: explicit JOIN

```
SELECT F.firstName, P.petName
FROM Notes.Friends F JOIN Notes.Pets P
ON F.friendID = P.friendID;
```

>>> SQL query: JOIN

Yet another way to write the same query:

```
SELECT F.firstName, P.petName
FROM Notes.Friends F INNER JOIN Notes.Friends P
ON F.friendID = P.friendID
```

>>> SQL query: JOIN

Note that JOIN is an operator that is inside the FROM clause.

```
FROM Friends F JOIN Pets P ON F.FriendID = P.FriendID
```

Pets			
PetID	PetName	…	FriendID
1	Chikin		2
2	Cauchy		3
3	Gauss		3

Friends		
FriendID	FirstName	…
1	X	
2	Y	
3	Z	

```
SELECT F.FirstName, P.PetName
```

Unnamed						
PetID	PetName	…	FriendID	FriendID	FirstName	…
1	Chikin	…	2	2	Y	…
2	Cauchy	…	3	3	Z	…
3	Gauss	…	3	3	Z	…

result

Unnamed	
PetName	FirstName
Chikin	Y
Cauchy	Z
Gauss	Z

>>> Order of execution

JOIN is technically an operator, not a clause.

1. FROM (and JOIN)
2. WHERE
3. SELECT

>>> Group practice

Table1		
A	B	C
1	Ignorance	is
2	War	is
3	Freedom	is
4	Friendship	is

Table2		
D	E	A
slavery.	3	1
weakness.	4	2
strength.	1	3
peace.	2	4

* `SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.A`
* `SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.E`

>>> Solutions

* `SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.A`

B	C	D
Ignorance	is	slavery.
War	is	weakness.
Freedom	is	strength.
Friendship	is	peace.

>>> Solutions

* `SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.A`

B	C	D
Ignorance	is	slavery.
War	is	weakness.
Freedom	is	strength.
Friendship	is	peace.

* `SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.E`

B	C	D
Ignorance	is	strength.
War	is	peace.
Freedom	is	slavery.
Friendship	is	weakness.

>>> SQL query: LEFT JOIN

The join query below (that we looked at earlier)
excludes any friends that have no pets (and vice versa).

```
SELECT F.firstName, P.petName
FROM Notes.Friends F JOIN Notes.Pets P
ON F.friendID = P.friendID;
```

>>> SQL query: LEFT JOIN

The join query below (that we looked at earlier)
excludes any friends that have no pets (and vice versa).

```
SELECT F.firstName, P.petName
FROM Notes.Friends F JOIN Notes.Pets P
ON F.friendID = P.friendID;
```

LEFT JOIN keeps every row from the table on the left.

```
SELECT F.firstName, P.petName
FROM Notes.Friends F LEFT JOIN Notes.Pets P
ON F.friendID = P.friendID;
```

result

Unnamed	
PetName	FirstName
NULL	X
Chikin	Y
Cauchy	Z
Gauss	Z

>>> One more LEFT JOIN example. Remember this?

```
FROM Friends F JOIN Pets P ON F.FriendID = P.FriendID
```

Pets			
PetID	PetName	…	FriendID
1	Chikin		2
2	Cauchy		3
3	Gauss		3

Friends		
FriendID	FirstName	…
1	X	
2	Y	
3	Z	

>>> The result was...

Unnamed						
PetID	PetName	…	FriendID	FriendID	FirstName	…
1	Chikin	…	2	2	Y	…
2	Cauchy	…	3	3	Z	…
3	Gauss	…	3	3	Z	…

If we did an LEFT JOIN instead we would get:

```
FROM Friends F LEFT JOIN Pets P ON F.FriendID = P.FriendID
```

Unnamed						
PetID	PetName	…	FriendID	FriendID	FirstName	…
NULL	NULL	…	NULL	1	X	…
1	Chikin	…	2	2	Y	…
2	Cauchy	…	3	3	Z	…
3	Gauss	…	3	3	Z	…

Question for the class:

What does RIGHT JOIN do?

Question for the class:

What does FULL OUTER JOIN do?

Do Exercises Section 5.2.3

Click here to find the textbook.

>>> End of day 1

School's out