

>>> Introduction to SQL
>>> Featuring MySQL and T-SQL

Daniel Fryer [†]
Feb, 2022

[†]daniel@vfryer.com

```
>>> Daily schedule
```

| Timetable | | | | |
|-----------|---|---------|------------------|----------|
| 9:00am | - | 10:30am | lecture 1 | (1.5 hr) |
| 10:30am | - | 11:00am | morning tea | (30 min) |
| 11:00am | - | 12:30pm | lecture 2 | (1.5 hr) |
| 12:30pm | - | 1:30pm | lunch | (1 hr) |
| 1:30pm | - | 3:00pm | guided exercises | (1.5 hr) |
| 3:00pm | - | 5:00pm | one-on-one help | (2 hr) |

>>> Overview

Day 1

1. Introduction
2. Intro to relational model
3. Tables and relationships
4. Programming in SQL
5. Basic SQL
6. Intro to subqueries
7. Joining in SQL

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

>>> How to pronounce SQL

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

We will be boldly using two dialects of (ISO/ANSI) SQL:

- * T-SQL (Microsoft, proprietary)
- * MySQL (Oracle, open-source)

```
>>> No ice breaker... yet
```

Breakout rooms during the exercises

```
>>> Show of hands
```

Past experience

>>> The Kahoots!

Definition

A Kahoot is a fun group quiz that we'll do here and there throughout the course. Join in to test your skills.

And now for a practice Kahoot...

>>> Where are we now?

Day 1

1. Introduction
2. Intro to relational model
3. Tables and relationships
4. Programming in SQL
5. Basic SQL
6. Intro to subqueries
7. Joining in SQL

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


```
>>> Let the learning begin
```

A Relational Database Management System (RDBMS).

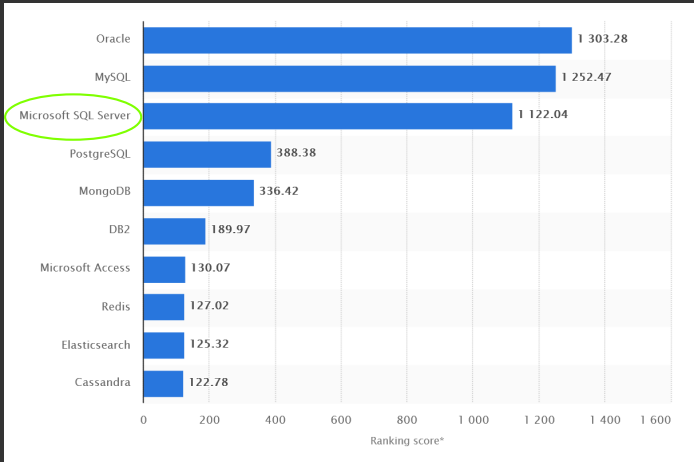
Definition

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](https://www.statista.com)

>>> The SQL bridge

Data Analyst

Responsibilities

- Curate the data
- Visualize and report data

Tools

- Excel
- SQL
- Tableau

Skills

- Analytics
- Communication & Visualization

Business Facing

- Sometimes

Salary

\$65,000



Data Scientist

Responsibilities

- Source data
- Analyze data
- Run experiments

Tools

- Python
- R
- SQL

Business Facing

- Yes

Build models

- Recommend solutions
- Storytell

Skills

- Analytics
- Communication
- Story-tell
- Model building
- Math
- Coding

Salary

\$120,000



Data Engineer

Responsibilities

- Build data pipelines and warehouse
- Manage scalability of data products

Tools

- Java
- C++
- Kubernetes
- Hadoop
- Spark
- Python

Skills

- Coding
- Model implementation

Business Facing

- Not Really

Salary

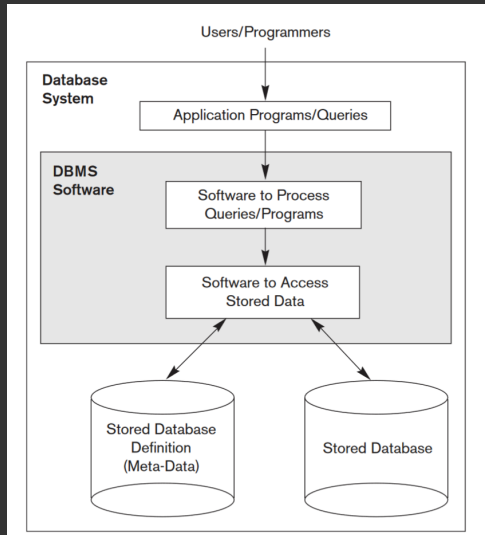
\$110,000



*Matha*Magicians

>>> DBMS

A layer of abstraction between human and machine



>>> RDBMS

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).

>>> RDBMS

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**.

Intuition

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 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | 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 loooooong way.


```
>>> The anatomy of a table
```

| Friends | | | |
|----------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |

```
>>> The anatomy of a table
```

Table name

| Friends | | | |
|----------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |

```
>>> The anatomy of a table
```

Row
(record)

| Friends | | | |
|----------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |

```
>>> The anatomy of a table
```

| Friends | | | |
|----------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |

Column (attribute)

>>> The anatomy of a table

Column names (attribute names)

| Friends | | | |
|----------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |

>>> The anatomy of a table

| Primary key | | Friends | |
|-------------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |

>>> The anatomy of a table

| Primary key | | Friends | |
|-------------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | 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

```
Friends(FriendID, FirstName, LastName, FavColour)
                varchar      varchar      varchar
```

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. Intro to relational model
3. Tables and relationships
4. Programming in SQL
5. Basic SQL
6. Intro to subqueries
7. Joining in SQL

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

>>> 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 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | 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 → delete pets
- * Ambiguity. Is information related to pets or owners?


```
>>> So what do we do instead?
```

Suspense.

Kahoot time! The relational model.

```
>>> What if we do this instead?
```

| Friends | | | |
|----------|-----------|-----|---------|
| FriendID | FirstName | ... | PetName |
| 1 | X | ... | NULL |
| 2 | Y | ... | Chikin |
| 3 | Z | ... | Cauchy |
| 3 | Z | ... | Gauss |

```
>>> What if we do this instead?
```

| Friends | | | |
|----------|-----------|-----|---------|
| FriendID | FirstName | ... | PetName |
| 1 | X | ... | NULL |
| 2 | Y | ... | Chikin |
| 3 | Z | ... | Cauchy |
| 3 | Z | ... | Gauss |

This causes data redundancy

```
>>> 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

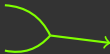
| Pets | | | |
|-------|---------|-----|----------|
| PetID | PetName | ... | FriendID |
| 1 | Chikin | ... | 2 |
| 2 | Cauchy | ... | 3 |
| 3 | Gauss | ... | 3 |



| 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 → 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

```
>>> Challenge
```

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 | <i>X</i> | <i>A</i> | red | 2 |
| 2 | <i>Y</i> | <i>B</i> | blue | NULL |
| 3 | <i>Z</i> | <i>C</i> | 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.

>>> Referential integrity

When there is a foreign key entry that is not NULL, the primary key entry that it 'points at' must exist.

>>> Referential integrity

When there is a foreign key entry that is not NULL, the primary key entry that it 'points at' must exist.

Guarantees that a foreign key is not 'meaningless'.

```
>>> Referential integrity
```

Guarantees that a foreign key is not 'meaningless'.

| Friends | | | | |
|----------|-----------|----------|-----------|--------------|
| FriendID | FirstName | LastName | FavColour | DefeatedByID |
| 1 | <i>X</i> | <i>A</i> | red | 4 |
| 2 | <i>Y</i> | <i>B</i> | blue | NULL |
| 3 | <i>Z</i> | <i>C</i> | NULL | 2 |

```
>>> Identifying a primary / foreign key pair?
```

A foreign key is any column (or collection of columns) where each record is **guaranteed** to equal one, and only one, primary key entry in the other table.

Problem: What happens if the database is sloppy, and there aren't any foreign keys??

```
>>> An example to contemplate
```

```
Houses(Bedrooms, Bathrooms, LandSize, PostCode)  
Suburbs(PostCode, SuburbName).
```


>>> An example to contemplate

```
Houses(Bedrooms, Bathrooms, LandSize, PostCode)  
Suburbs(PostCode, SuburbName).
```

1. Is every PostCode entry in Suburbs unique?
2. Is every PostCode in Houses also in Suburbs?

Does it really matter if we can't tell?

>>> An example to contemplate

```
Houses(Bedrooms, Bathrooms, LandSize, PostCode)  
Suburbs(PostCode, SuburbName).
```

1. Is every PostCode entry in Suburbs unique?
2. Is every PostCode in Houses also in Suburbs?

Does it really matter if we can't tell?

- * From 1: can't be sure which suburb a house is in.
- * From 1: joining can lead to unexpected duplicates.
- * From 2: can't find any matching suburb.

>>> 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

>>> Will see again during the exercises

- * 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?

- * 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 → 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
```

How do we delete a friend without deleting their passport?

>>> Solution

How do we delete a friend without deleting their passport?

| 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

How do we delete a friend without deleting their passport?

| 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?

How do we delete a friend without deleting their passport?

| 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?

How do we delete a friend without deleting their passport?

| 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 avoid 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 represents `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 damage 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...

>>> Where are we now?

Day 1

1. Introduction
2. Intro to relational model
3. Tables and relationships
4. Programming in SQL
5. Basic SQL
6. Intro to subqueries
7. Joining in SQL

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

>>> Walk-through of SQL Server

Time for the real deal

If you've completed set-up, that's great!
Otherwise, we can troubleshoot it this afternoon.

If you're on macOS, don't worry!
I'll be giving a demo of Sequel Ace later.

>>> Demonstration

In Azure Data Studio, I'll do the following:

- * Connect to 'localhost'.
- * Open a new query tab.
- * Change between databases.
- * Figure out what tables are in a database.
- * Explain what a schema is.
- * Figure out what columns are in a table.
- * Figure out what the data types are.
- * Figure out what the primary/foreign keys are.
- * Figure out if NULL values are allowed.

```
>>> Bonus demo
```

Sneak preview of SQL code

- * The `USE` clause.
- * Retrieve `Friends`.
- * Retrieve `Pets`.
- * Join `Friends` with `Pets`.
- * Aliases.
- * Quoting identifiers.

>>> 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 JOIN Notes.Pets  
ON Notes.Friends.friendID = Notes.Pets.friendID;
```

With aliases:

```
SELECT *  
FROM Notes.Friends F JOIN Notes.Pets P  
ON 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 JOIN Notes.Pets AS P  
ON F.friendID = P.friendID;
```

>>> Where are we now?

Day 1

1. Introduction
2. Intro to relational model
3. Tables and relationships
4. Programming in SQL
5. Basic SQL
6. Intro to subqueries
7. Joining in SQL

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


```
>>> 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: 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.

Remember: MySQL doesn't have schemas (but don't Google it)

```
>>> 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;
```

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


```
>>> SELECT execution
```

```
FROM Notes.Friends
```

| Friends | | | |
|----------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |

```
>>> SELECT execution
```

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

| Friends | | | |
|----------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |

```
>>> SELECT execution
```

result

| Unnamed | |
|-----------|-----------|
| FirstName | FavColour |
| X | red |
| Y | blue |
| Z | NULL |

```
>>> Order of execution
```

But did you see that order of execution?

```
>>> Order of execution
```

But did you see that order of execution?

- * 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';
```

```
>>> WHERE execution
```

```
FROM Notes.Friends
```

| Friends | | | |
|----------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |

WHERE FavColour = 'red'

| Friends | | | |
|----------|-----------|----------|-----------|
| FriendID | FirstName | LastName | FavColour |
| 1 | <i>X</i> | <i>A</i> | red |
| 2 | <i>Y</i> | <i>B</i> | blue |
| 3 | <i>Z</i> | <i>C</i> | NULL |


```
SELECT FirstName, LastName
```

| Unnamed | | | |
|---------|-----------|----------|-----------|
| ID | FirstName | LastName | FavColour |
| 1 | X | A | red |

result

| Unnamed | |
|-----------|----------|
| FirstName | LastName |
| <i>X</i> | A |

```
>>> Order of execution
```

1. FROM
2. WHERE
3. SELECT

```
>>> Order of execution
```

Order of execution is irrelevant, Danny!

```
>>> Order of execution
```

Order of execution is irrelevant, Danny!

Wrong you are.

Aliases can be created in the `SELECT` clause too!

```
SELECT F.FirstName AS Nombre, F.FavColour AS ColorFavorito  
FROM Notes.Friends F  
WHERE ColorFavorito = 'red';
```

Let's try executing the above. What will happen?

```
>>> Why do you keep saying 'clause'?
```

SQL is like speaking ... or cooking.

- * **Clauses** are components of **statements**.
- * The statements we're learning are called **queries**.
- * A statement is somewhat comparable to a 'sentence'.
- * Better to think of them as ingredients in a recipe?

>>> Chopping and changing

- * We've seen how to 'chop' (with `SELECT` and `WHERE`).
- * We've seen how to 'change' (with table/column aliases).

Can we also change the entries?

>>> Chopping and changing

- * We've seen how to 'chop' (with `SELECT` and `WHERE`).
- * We've seen how to 'change' (with table/column aliases).

Can we also change the entries?

Change entries with the `CASE WHEN` expression.

```
SELECT *, CASE WHEN FavColour = 'red' THEN 'rojo'
            WHEN FavColour = 'blue' THEN 'azul'
            ELSE FavColour END AS ColorFavorito
FROM Notes.Friends;
```

Let's execute the above. What will it do?


```
>>> Ordering
```

We can also reorder the results!

```
SELECT *  
FROM Notes.Friends  
ORDER BY FriendID DESC;
```

Let's execute it to experiment.

```
>>> Lexicographic ordering
```

What happens if we order by a character string?

| Numbers | |
|---------|-----------|
| Num | NumString |
| 111 | '111' |
| 31 | '31' |
| 32 | '32' |
| 211 | '211' |

```
SELECT *  
FROM Notes.Numbers  
ORDER BY NumString;
```

Let's execute it to find out.

```
>>> Transforming entries
```

Functions that transform entries are often called **scalar functions**. Perhaps the most important is:

```
SELECT *  
FROM Notes.Numbers  
ORDER BY CAST( NumString AS INT );
```

Let's execute it.

```
>>> Other scalar functions
```

The three categories we will look at are

- * Mathematical functions
- * String functions
- * Date and time functions

Any many more ([click here](#)).

```
>>> Three mathematical functions
```

| Function | Description |
|----------|------------------------|
| SQRT | Square root |
| ROUND | Rounding |
| RAND | Generate random number |

I will go through some examples...

```
>>> Two string functions
```

| Function | Description |
|-----------|---------------------|
| CONCAT | Concatenate columns |
| SUBSTRING | Extract characters |

I will go through some examples...

```
>>> Three date/time functions
```

| Function | Description |
|----------|--------------------------------|
| DAY | Extract the day (of the month) |
| MONTH | Extract the month |
| YEAR | Extract the year |

I will go through some examples...

>>> Where are we now?

Day 1

1. Introduction
2. Intro to relational model
3. Tables and relationships
4. Programming in SQL
5. Basic SQL
6. Intro to subqueries
7. **Joining in SQL**

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 JOIN Notes.Pets P  
ON 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 JOIN Notes.Pets P  
ON F.friendID = P.friendID;
```

Another way to write the same query: **implicit syntax**

```
SELECT F.firstName, P.petName  
FROM Notes.Friends F, Notes.Pets P  
WHERE 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 | |

```
>>> SQL query: JOIN
```

```
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 | ... |

```
>>> SQL query: JOIN
```

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;
```

>>> SQL query: LEFT JOIN. 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 | ... |

```
>>> SQL operator: LEFT JOIN
```

If we did a 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 | ... |

```
>>> SQL query: LEFT JOIN
```

result

| Unnamed | |
|---------|-----------|
| PetName | FirstName |
| NULL | X |
| Chikin | Y |
| Cauchy | Z |
| Gauss | Z |


```
>>> SQL query: RIGHT JOIN
```

Question for the class:

What does RIGHT JOIN do?

>>> Exercises

Do exercises at the ends of Chapters 1 and 2.

[Click here to find the textbook.](#)