SQL Basics

for

myEvolv Reporting

Table of Contents

Relational Databases & Data Normalization	3
myEvolv's Structure	5
Primary & Foreign Keys	6
SQL Syntax	
Table & Column Notation	7
SELECT & FROM	7
JOIN	8
AS	10
WHERE	10
Bits and Pieces	
IN	11
LIKE	11
ORDER BY	11
Date Format	12
DATEDIFF	12
DATEADD & GETDATE	12

Relational Databases and Data Normalization

When a database is described as relational, it has been designed to conform to a set of practices called the rules of normalization. A normalized database is one that follows the rules of normalization.

For example, in myEvolv, we have clients who are enrolled in various programs. Each client and program has a number and a name. You could organize this information as shown in Table 1.

Table 1: Sample Client Information

ClntNo	**ClntName**	**ProgNo**	**ProgName**
101	Abigail	10	Residential
102	Bob	20	Foster Care
103	Carolyn	10	Residential
104	Doug	20	Foster Care
105	Evelyn	10	Residential

If you structure your data this way and make certain changes to it, you'll have problems. For example, deleting all the clients in the Residential program will eliminate the program itself. If you change the name of the Foster Care program to "Therapeutic Foster Care," you would need to change the record of each client in that program.

Using the principles of relational databases, the Client and Program data can be restructured into two separate tables (CLNT and PROG), as shown in Tables 2 and 3.

Table 2: A Sample Relational PROG Table

ProgNo	**ProgName**
10	Residential
20	Foster Care

Table 3: A Sample Relational CLNT Table

ClntNo	**CName**	**ProgNo**
101	Abigail	10
102	Bob	20
103	Carolyn	10
104	Doug	20
105	Evelyn	10

By using this structure, you can examine the CLNT table to find out that Doug is enrolled in program 20. Then you can check the PROG table to find out that program 20 is Foster Care. You might think that Table 1 looks more efficient. However, retrieving the information you need in a number of different ways is much easier with the two-table structure. Joining the information in the two tables for more efficient retrieval is exactly the problem that relational databases were designed to solve.

myEvolv's Structure

myEvolv has been designed as an event-based system. All of the things that users add to an individual's record are recorded as types of events. What this means for the database is that the Event Log table is one of the most important tables in the database and will play into most of the queries that you will write.

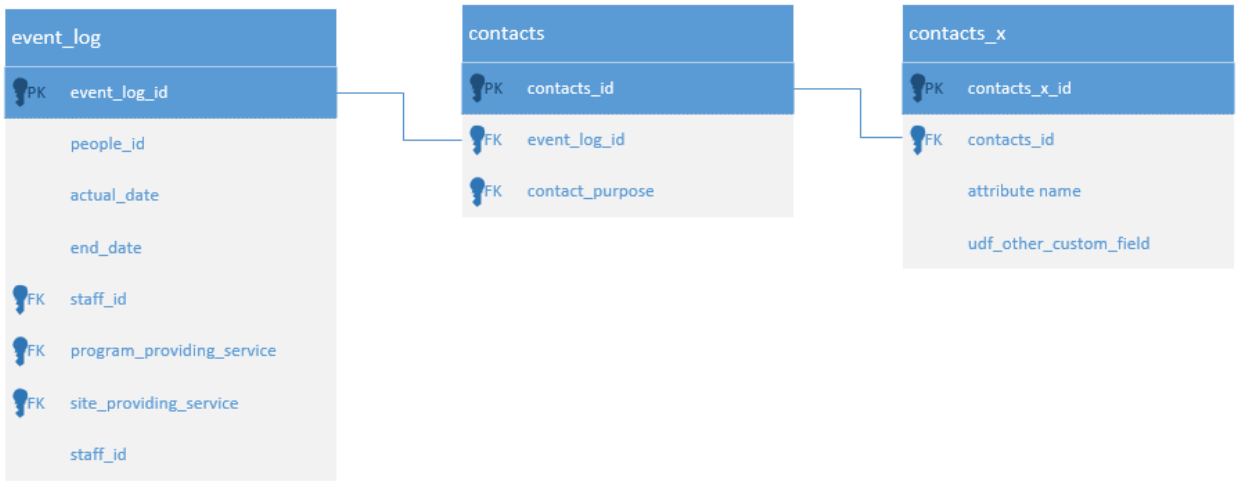
If you run a basic query on the event log table

```
SELECT * FROM event_log
```

You will notice that many of the columns in the table hold random strings of letters and numbers and the rest are largely dates. The random-string columns are holding foreign key references to other tables and there are often a lot of tables to be referenced, even for basic events.

You may recall that when you are using the form designer in myEvolv that you select a form family to use. Each form family refers to a table in the myEvolv database that holds the data related to those types of forms. For example, if you are working in the Diagnosis form family, you are able to add fields from the Event Log table and the Diagnosis table. If you are working in the Activities – People form family, you are able to add fields from the Event Log table and the Contacts table.

If you add a user-defined field or New Database Field, you are creating it in a related table. So if you are in the Activities – People form family, your user-defined field is created in the Contacts X table.



You will see this pattern repeated for each form family and you will use it to access your user-defined fields.

Primary Keys and Foreign Keys

Tables are linked through referencing keys that match between two or more tables. Keys are typically integers or strings and are stored in a table's primary key column and any foreign key columns. myEvolv uses a GUID (Globally Unique Identifier) for its keys.

```
5ce1fc17-63a4-4975-aa11-0c0e9cc873b8
```

A **primary key** is assigned to each record/row in a table and it uniquely identifies that record in the table. For the most part, in a myEvolv table, the primary key column will be named after the table itself. For example, the Event Log table's primary key column is `event_log_id` and the People table's primary key is `people_id`.

A **foreign key** is the reference to another table's primary key stored in a table. A table may have no foreign key columns in it or it may have many. The column will have the primary key of the referenced table stored in it and in myEvolv, the foreign key column is *typically* named after the table it references. For example, the Event Log table has a foreign key column of `people_id` that references the primary key column of the People table.

The Event Log table has (at least) two commonly used foreign key columns that do not match the names of their respective tables. `program_providing_service` and `site_providing_service` refer to the Program Info and Group Profile tables respectively.

In order to determine the table reference for a foreign key, it may be necessary to consult the Data Dictionary in myEvolv. Setup > User Tools > Data Dictionary > All Tables

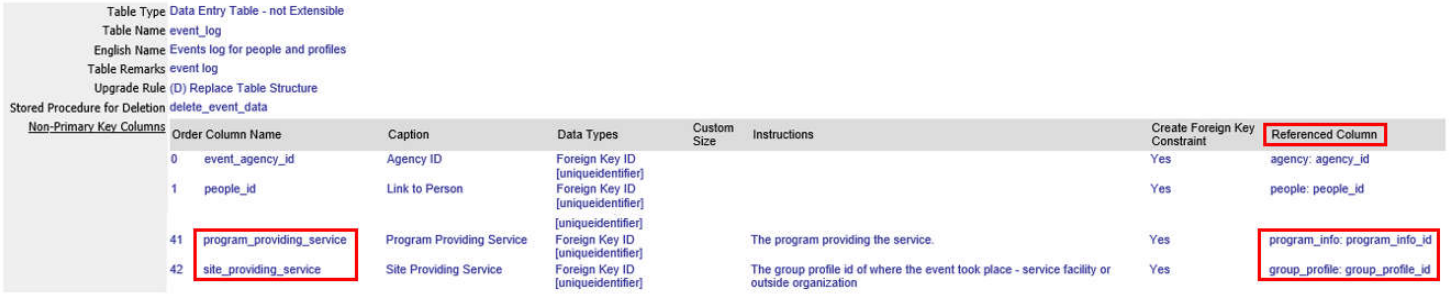


Table Type: Data Entry Table - not Extensible
Table Name: event_log
English Name: Events log for people and profiles
Table Remarks: event log
Upgrade Rule: (D) Replace Table Structure
Stored Procedure for Deletion: delete_event_data

Non-Primary Key Columns

Order	Column Name	Caption	Data Types	Custom Size	Instructions	Create Foreign Key Constraint	Referenced Column
0	event_agency_id	Agency ID	Foreign Key ID [uniqueidentifier]			Yes	agency: agency_id
1	people_id	Link to Person	Foreign Key ID [uniqueidentifier]			Yes	people: people_id
41	program_providing_service	Program Providing Service	Foreign Key ID [uniqueidentifier]		The program providing the service.	Yes	program_info: program_info_id
42	site_providing_service	Site Providing Service	Foreign Key ID [uniqueidentifier]		The group profile id of where the event took place - service facility or outside organization	Yes	group_profile: group_profile_id

SQL Syntax

Table & Column Notation

You refer to specific columns in the database using dot (.) notation where you specify the table and column:

people

people_id	last_name	first_name	dob
a95a615d-6e58-4760-b77d-570475e5cd58	Smith	John	1969-05-04
b64d9db6-4aa9-4fdb-981f-0ab0584ebbd6	Jones	Jessica	1982-09-09
a8ceed63-55e4-4166-8816-066da5d46f0e	Williams	Benito	1950-06-15
f0570b52-e634-4dd2-81bc-7376bc735f6d	Collins	Joan	1952-07-19
40076362-fcdd-4b56-8a05-02a366b1e05e	Perez	Wendy	1998-01-12

```
[table].[column]
people.last_name
```

SELECT & FROM

The most basic queries you will write in SQL are simply listing the columns that you would like to **SELECT** **FROM** a specific table.

```
SELECT
    people.last_name,
    people.first_name,
    people.dob
FROM people
```

If you would like all of the columns from a table, you can use the asterisk (*) as a wild card.

```
SELECT
    people.*
FROM people
```

These types of queries are limited to getting information from a single table because you cannot stack the **FROM** clause with additional tables like you could with the **SELECT** clause. To add additional tables to our queries, we must use `JOIN`.

JOIN

`JOIN` clauses allow you to bring more tables into the query and also to do some filtering on the results based on which type of `JOIN` you employ. There are 4 different basic `JOINS` that you might use. The most common are the `INNER JOIN` and the `LEFT JOIN`.

Here are the different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table

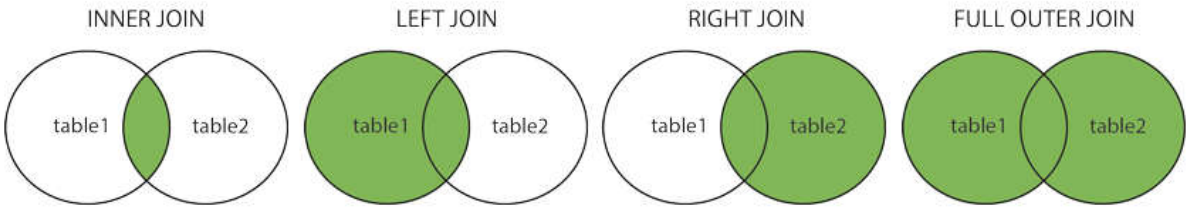


Image from w3schools.com

The `LEFT` and `RIGHT` tables in the descriptions above refer to the first table listed (`LEFT`) and the other table being joined to it (`RIGHT`)

It is important to understand when to use which `JOIN` in your query since they act as a filter and you do not want to inadvertently exclude rows in your results. For example, let's say that you want a list of all clients and their addresses to use for a mailing. To do this, you must join the address table to the people table.

If you use an `INNER JOIN`, the query will only grab records where there is a matching `people_id` in both tables.

```
SELECT
people.people_id,
people.first_name,
people.last_name,
address.address_id,
address.people_id,
address.address,
address.city,
address.zip_code
FROM people
JOIN address ON address.people_id = people.people_id
```

Your results would look something like this:

people_id	last_name	first_name	address_id	people_id	street	city	zip_code
7b91e6fb-6a9b-46c8-86d8-2e17fe5d399a	Williams	Seth	cc137c55-a935-4278-815d-7218e506a529	7b91e6fb-6a9b-46c8-86d8-2e17fe5d399a	123 Main St.	Utica	13501
513a6935-6591-410c-932f-b38335cfa0b4	Smith	Jane	af9b8f9b-7403-4b2f-aff5-08737bf53dd0	513a6935-6591-410c-932f-b38335cfa0b4	456 Banks Ave.	Homer	13077
e632a787-0798-4916-8940-d1aa3d58ab96	Martinez	Mark	e8cedc73-4a7b-4ccf-b0e2-2f92fef1b568	e632a787-0798-4916-8940-d1aa3d58ab96	9 Fleet St.	Ithaca	14850
c246c6b8-dd32-415f-9b26-8f066a16c397	Johnson	Antonio	b8ef7698-b322-4990-94e7-524d36f26506	c246c6b8-dd32-415f-9b26-8f066a16c397	42 Albany St.	Ithaca	14850
74583807-40a2-4897-8c48-4e051f6f12a2	Manning	Maria	589bb5e2-a7ee-419c-a26a-2adb4394aa95	74583807-40a2-4897-8c48-4e051f6f12a2	222 State Route 13	Cortland	13045

If you use a `LEFT JOIN`, the query will return all rows from the people table and fill in address table columns where there is a match on the `people_id`. Where there is no match, the values will be `NULL`.

```
SELECT
people.people_id,
people.first_name,
people.last_name,
address.address_id,
address.people_id,
address.address,
address.city,
address.zip_code
FROM people
LEFT JOIN address ON address.people_id = people.people_id
```

Your results would look something like this:

people_id	last_name	first_name	address_id	people_id	street	city	zip_code
7b91e6fb-6a9b-46c8-86d8-2e17fe5d399a	Williams	Seth	cc137c55-a935-4278-815d-7218e506a529	7b91e6fb-6a9b-46c8-86d8-2e17fe5d399a	123 Main St.	Utica	13501
513a6935-6591-410c-932f-b38335cfa0b4	Smith	Jane	af9b8f9b-7403-4b2f-aff5-08737bf53dd0	513a6935-6591-410c-932f-b38335cfa0b4	456 Banks Ave.	Homer	13077
eb8c64f3-95c2-4fde-b874-c249f9493e59	Cook	Colin					
e632a787-0798-4916-8940-d1aa3d58ab96	Martinez	Mark	e8cedc73-4a7b-4ccf-b0e2-2f92fef1b568	e632a787-0798-4916-8940-d1aa3d58ab96	9 Fleet St.	Ithaca	14850
43b0d7f3-ad7d-43af-8b6a-21cca5fd008b	Adams	Sarah					
f611a781-f029-44d4-8b7e-43ffa9270680	Washington	Lee					
c246c6b8-dd32-415f-9b26-8f066a16c397	Johnson	Antonio	b8ef7698-b322-4990-94e7-524d36f26506	c246c6b8-dd32-415f-9b26-8f066a16c397	42 Albany St.	Ithaca	14850
74583807-40a2-4897-8c48-4e051f6f12a2	Manning	Maria	589bb5e2-a7ee-419c-a26a-2adb4394aa95	74583807-40a2-4897-8c48-4e051f6f12a2	222 State Route 13	Cortland	13045

Notice that in this result, we have Colin Cook, Sarah Adams and Lee Washington included in the results whereas they were not in the `INNER JOIN` results. That is because the `INNER JOIN` filtered these clients out for not having an address record. The `LEFT JOIN` insured that all people from the people table are in your result set, regardless of whether they have an address in the system.

When using a `JOIN` clause in your query, it is accompanied with one or more `ON` conditions which defines which columns should be matched but can also be used to do further filtering. For example, you might use

```
JOIN address ON address.people_id = people.people_id AND address.zip_code <> '14850'
```

This `JOIN` clause will get all people with a matching address record but leave out anyone with a 14850 zip code. It can be useful or even sometimes necessary to do this filtering on your `JOIN`s, but SQL also includes the `WHERE` clause to further filter your results.

AS

In the example above, both tables have a `people_id` column and we selected both in our query. Some report writing programs will handle duplicate column names automatically but others will require using an alias for the column name so that they can be distinguished. You may also need to give whole tables an alias when you use the same table multiple times in one query. To give a column or table an alias, you use the `AS` keyword:

```
1   SELECT
2   people.people_id AS people_people_id,
3   people.first_name,
4   people.last_name,
5   address.address_id,
6   address.people_id AS address_people_id,
7   address.address,
8   address.city,
9   address.zip_code
10  FROM people
11  LEFT JOIN address ON address.people_id = people.people_id
```

It can also be useful to use aliases even when they aren't necessary just to shorten table names when you will be typing them out a lot. For example, you might want to rename the

`budget_capitation_totals_mirror_header_summary_view`

to just

`budg_cap`

when writing your queries.

WHERE

After your `JOIN`s, you can use a `WHERE` clause in order to filter your result set down to just the records you want see. As an alternate to the `JOIN` clause above where we filtered out the 14850 zip codes, we could have just tacked the following `WHERE` clause to the end to achieve the same effect:

```
WHERE address.zip_code <> '14850'
```

You are only allowed one `WHERE` clause per query so you have to stack your conditions using `AND`, `OR` and `NOT` in order to get more complex.

```
WHERE address.zip_code <> '14850' AND people.last_name <> 'Johnson'
```

The main challenge with `WHERE` clauses comes from understanding how to use `AND`, `OR` and `NOT` in combination with one another.

X AND Y OR Z
is not the same as
X AND (Y OR Z)
is not the same as
(X AND Y) OR Z

Bits and Pieces

Here are some other helpful pieces of syntax that you might find helpful in writing queries

IN

If you want to filter for a list of values rather than one and you don't want to write a long chain of `AND`s, you can use the `IN` operator and provide a list of values to match against:

```
WHERE program_info.program_name IN ('Residential', 'Foster Care', 'Substance Abuse')
```

LIKE

If you know some portion of the value for something, you can use the `LIKE` keyword and the `%` and _ characters in your clause as wild cards to find partial matches.

`%` - The percent sign represents zero, one, or multiple characters

_ - The underscore represents a single character

WHERE program_info.program_name LIKE 'a%'	Finds any values that start with "a"
WHERE program_info.program_name LIKE '%a'	Finds any values that end with "a"
WHERE program_info.program_name LIKE '%or%'	Finds any values that have "or" in any position
WHERE program_info.program_name LIKE '_r%'	Finds any values that have "r" in the second position
WHERE program_info.program_name LIKE 'a_%_%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE program_info.program_name LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

ORDER BY

The `ORDER BY` keyword allows you to sort your results by one or more columns. The default is to sort in ascending order but you can explicitly determine the order with the `ASC` and `DESC` keywords. `ORDER BY` clauses must go after the `WHERE` clause.

```
ORDER BY people.last_name ASC
```

Date Format

myEvolv uses a SQL Server for its database, so dates should be provided in a *yyyy-mm-dd* format.

```
SELECT
people.last_name
people.first_name,
people.dob
FROM people
WHERE people.dob > '2018-01-01'
```

This query will find all people who were born since January 1, 2018

DATEDIFF

SQL includes some functions that can be useful when you are trying to fine tune the records you are looking for. The `DATEDIFF` function returns the number of the specified interval between two dates. It can count days, months, years, etc. For example, you may want tocheck contemporaneousness of documentation by finding services that were entered into myEvolv (`event_log.date_entered`) more than 3 days after they were provided (`event_log.actual_date`):

```
SELECT
people.last_name,
people.first_name,
def.event_name,
event_log.actual_date,
event_log.date_entered,
staff.staff_name
FROM event_log
JOIN people ON people.people_id = event_log.people_id
JOIN event_definition AS def ON def.event_definition_id = event_log.event_definition_id
JOIN staff_view AS staff ON staff.staff_id = event_log.staff_id
WHERE DATEDIFF(day, event_log.actual_date, event_log.date_entered) > 3
```

Syntax:

DATEDIFF(*interval, date1, date2*)

DATEADD & GETDATE

The `DATEADD` function adds increments of time to the specified date. This can be useful when you are trying to do things like find services provided in the last 30 days without hard coding today's date into the query. Note also the `GETDATE` function, which can be used instead of a date and will always equal the date that the query is being run.

```
SELECT
people.last_name,
people.first_name,
def.event_name,
event_log.actual_date,
event_log.date_entered,
staff.staff_name
FROM event_log
JOIN people ON people.people_id = event_log.people_id
JOIN event_definition AS def ON def.event_definition_id = event_log.event_definition_id
JOIN staff_view AS staff ON staff.staff_id = event_log.staff_id
WHERE event_log.actual_date >= DATEADD(day, -30, GETDATE())
```

Syntax:

DATEADD(*interval, number, date*)