2021-07-29T18:00:00+02:00 | Michal Stín

SaaS KPI Metrics, Part 1: Using SQL to Calculate Customer Lifetime Value (LTV) and Related Metrics

Working with data is an essential skill for marketers in today's data-driven world. As a marketer myself, I decided to create a series of articles about calculating key performance indicator (KPI) metrics for SaaS (Software as a Service) companies. We'll start by calculating customer lifetime value (LTV) using SQL.

Data is one of today's most valuable resources. If you work in a SaaS company, you are flooded with a huge amount of data every day. Without the support of a full Business Intelligence department, it is often difficult for a marketer to profit from such data.

Even if you are lucky enough to be able to rely on your IT colleagues, it is liberating to be able to analyze key metrics on your own. Excel and Google Sheets are great analytical tools, but you'll quickly come to the point where you need to dig deeper and look for your data directly in the database. And that's when you need to be able to write SQL queries.

As a marketer without a degree in computer science, I know that gaining this kind of knowledge without a formal technical education may seem difficult. But don't worry – you don't have to know IT to learn SQL. Thanks to the online courses on LearnSQL.com, all you'll need to do is find some time in your busy calendar and enroll in the SQL Fundamentals course. You can learn SQL for sure, even if you haven't had any previous experience with databases.

Learn the SQL basics by doing interactive courses from our SQL Fundamentals track!

Since you are reading this article, you probably already know that this blog is worth following. It has plenty of articles to help you broaden your skills and achieve your goals.

I did exactly that. A few weeks ago, I started to learn SQL; now not only do I use it on a daily basis as a marketer, I also wrote this article. I didn't think it would all go so quickly, but I am an example that it can be done. So no excuses – let's get to work!

Computing LTV and Related SaaS KPI Metrics Using SQL

My initial goal was to compute **the LTV of our customers based on GEO segments.** I ended up with a pretty complex query that also computed some other interesting KPIs you may find useful for your SaaS business.

Although you're welcome to adapt my approach to your needs, it's usually not possible to provide a copy/paste query for every SaaS business. Each business has different data architecture and limitations. However, I will explain how to use SQL to calculate each metric and bring them all together to get the LTV value. Specifically, I will describe how we compute these six SaaS KPIs at UptimeRobot:

Total Current Customers
Current Average Monthly Recurring Revenue (AVG MRR)
Customer Lifetime value (LTV)
Current Monthly Recurring Revenue (MRR)
All Time Average Revenue per User (ARPU)
Average Customer Lifetime

Why Calculate Customer Lifetime Value?

I am a really big fan of Stephen Covey's end goal mindset, so I will start from the end. Knowing the lifetime value of your customers is the key to success for most of your marketing efforts and decisions. You'll want to see LTV for segments like:

Source of customer acquisition
The day, month, or year of customer acquisition

Products (lifetime value of customers using different products)
Payment type (PayPal, credit card, invoices)
Payment period (monthly vs. annual)
Geographical location

As a new CMO, I was wondering **if there are significant differences in revenue made by our customers across countries.** By answering this question, we could better prioritize our activities and hopefully identify new opportunities for harvesting low-hanging fruits.

It would be also great to know the LTV based on the source of customer acquisition, but – due to current technical limitations in web analytics – I don't see any method of developing this report based on reliable data.

No matter which segment you choose, first you will need to compute the overall LTV of your customer base. To get that, you'll first need to compute some partial KPIs. Then you can segment. However, you will need to know where you are heading in the beginning so you will know how to prepare your data.

LearnSQL.com is an online platform designed to help you master SQL. LearnSQL.com allows you to choose from a full learning track, mini-tracks to sharpen targeted skills, and individual courses. You can also select the SQL dialect (Standard SQL, Microsoft SQL Server, or PostgreSQL) that best suits your needs.

Customer Lifetime Value (LTV) Formula

There are several methods to compute customer lifetime value. This one suits my needs best:

*LTV = ARPU * Avg. Lifetime*

I will be using data we store in the `payments` table. I will compute the LTV with the data from this one table, but you might need to work with multiple tables. For that, you'll need to understand SQL JOINs. If you're not familiar with them, I recommend looking at the SQL JOINs course.

I will be using JOIN in my computation of LTV. This Illustrated Guide to the SQL Self Join helped me understand this powerful concept. If you want more, you can also read SQL JOIN Types Explained.

We store records about every payment any user makes in the `payments` table. To find the LTV, I'll be using these fields:

- `userID` – Unique identifier for every user.
- `paymentDateTime`
- `paymentStatus` – Where 0 = failed, 1 = success, and 2 = refunded.
- `paymentPeriod` – In months: 1 for monthly payments, 12 for annual payments.
- `paymentAmount`
- `paymentBillingCountry`

Take a look at some example records showing the latest payments:

```
SELECT userID, paymentDateTime, paymentStatus, paymentPeriod, paymentBillingCountry
FROM payments
ORDER BY paymentDateTime DESC
limit 50
```

Message | Result 1 | Profile | Status

userID	paymentDateTime	paymentStatus	paymentPeriod	paymentBillingCountry
71203	2021-05-04 16:34:41	1	12	USA
1294846	2021-05-04 16:33:52	1	1	GBR
728329	2021-05-04 16:28:43	1	12	USA
480245	2021-05-04 16:27:49	1	1	USA
1168395	2021-05-04 16:23:51	1	1	(NULL)
248622	2021-05-04 16:22:08	1	12	DNK
632815	2021-05-04 16:17:42	1	12	USA
1246735	2021-05-04 16:13:17	1	0	USA
1180142	2021-05-04 16:09:23	1	1	ESP
4226	2021-05-04 16:09:02	1	12	USA
1180142	2021-05-04 16:05:59	0	1	ESP
1218427	2021-05-04 16:03:49	1	1	USA
838874	2021-05-04 16:02:12	1	12	BRA
214038	2021-05-04 16:02:01	1	12	GBR
646134	2021-05-04 15:54:38	1	12	USA
574216	2021-05-04 15:50:53	1	12	CHE
1075617	2021-05-04 15:50:15	1	1	GBR
411491	2021-05-04 15:49:16	1	12	ISR
1286169	2021-05-04 15:48:46	1	1	USA
222469	2021-05-04 15:44:40	1	1	CHL
431346	2021-05-04 15:41:22	1	1	GBR
188661	2021-05-04 15:40:33	1	1	BRA
119907	2021-05-04 15:36:47	1	1	POL
1090991	2021-05-04 15:36:07	1	1	NLD
205170	2021-05-04 15:35:45	1	12	DNK
1295849	2021-05-04 15:35:00	1	1	CAN

If I order this data by `userID`, you can see that one user can have multiple records. Each row represents a unique payment the user made. Some of the payments have a `paymentStatus = 0`, which means the payment failed.

Some records have a `NULL` value in `paymentBillingCountry`. To be honest, this should not happen; I found it out as I was writing this article and I have to ask our developers to fix it. That's a real-life example for you. You always find something unpredictable.

```sql
SELECT userID, paymentDateTime, paymentStatus, paymentPeriod, paymentAmount, paymentBillingCountry
FROM payments
ORDER BY userID DESC
limit 2000
```

Message | Result 1 | Profile | Status

userID	paymentDateTime	paymentStatus	paymentPeriod	paymentAmount	paymentBillingCountry
1295849	2021-05-04 15:35:00	1	1	8	CAN
1295813	2021-05-04 15:16:03	1	12	84	MEX
1295781	2021-05-04 14:52:50	1	1	8	TUR
1295285	2021-05-04 06:00:42	1	1	8	USA
1295045	2021-05-04 00:38:27	1	1	9.99	(NULL)
1295024	2021-05-03 23:08:17	1	1	8	CAN
1294962	2021-05-04 02:49:25	1	12	84	AUS
1294939	2021-05-03 21:22:18	1	1	9.99	(NULL)
1294939	2021-05-03 21:15:17	0	1	8	USA
1294939	2021-05-03 21:12:56	0	12	84	USA
1294939	2021-05-03 21:11:19	0	1	8	USA
1294888	2021-05-03 20:35:29	1	1	8	USA
1294846	2021-05-04 16:33:52	1	1	8	GBR
1294841	2021-05-03 20:25:11	1	1	8	USA
1294705	2021-05-03 17:25:05	1	1	9.99	(NULL)
1294688	2021-05-03 17:08:43	1	1	9.6	FRA
1294660	2021-05-03 16:45:29	1	1	8	CHL
1294645	2021-05-03 20:07:50	1	12	84	CAN
1294566	2021-05-03 15:29:54	1	12	100.8	FRA
1294458	2021-05-03 14:19:49	1	1	8	USA
1294450	2021-05-03 14:00:07	1	12	84	USA
1294444	2021-05-03 13:46:22	1	12	84	CAN
1294441	2021-05-03 15:37:50	1	12	252	USA
1294409	2021-05-03 13:32:08	1	12	84	USA
1294217	2021-05-03 09:43:25	1	12	84	FRA
1293926	2021-05-03 02:25:54	1	1	8	USA

Computing Monthly Recurring Revenue and Average Revenue Per User

Let's start computing the LTV part by part. First, we find the **ARPU (average revenue per user)** by computing the **MRR (monthly recurring revenue) of our current customers** and dividing it by the **number of total current customers** by country.

Here is the query for this job:

```
(SELECT paymentBillingCountry, count(DISTINCT userID) Total_Current_Customers,
ROUND(SUM(paymentAmount / paymentPeriod)) AS CurrentMRR
FROM payments
WHERE paymentDateTime < CURDATE() AND DATE_ADD(paymentDateTime, INTERVAL
paymentPeriod MONTH) >= CURDATE() AND paymentStatus = 1 AND paymentPeriod > 0 AND
userID IS NOT NULL
GROUP BY paymentBillingCountry
ORDER BY CurrentMRR DESC) AS arpu
```

We select `paymentBillingCountry` since we want to **group the data by country**, then with `count(DISTINCT userID)` we count the number of `Total_Current_Customers`. The `DISTINCT` statement ensures that we count every customer just once, even if they've made multiple payments.

Lastly, we count the `CurrentMRR` generated by all users from the given country as the `SUM` of the payment amount divided by the payment period (to prevent including whole payments in case of annual payments).

```
ROUND(SUM(paymentAmount / paymentPeriod)
```

We use the ROUND() function to make the output easier to digest (nicer numbers).

All the magic happens in the `WHERE` clause. Here, we filter just the payments of current (active) customers. We use the `DATE_ADD` function for this. By adding the `paymentPeriod` to the `paymentDateTime`, we can check if the customer's expiration date is after the current date.

```
DATE_ADD(paymentDateTime, INTERVAL paymentPeriod MONTH) >= CURDATE()
```

If this condition is TRUE, it means the customer is active and we want to include their last payment into the computation of `CurrentMRR`.

To get the desired result, we need to use `GROUP BY paymentBillingCountry`. If you are not familiar with GROUP BY, you can learn more about it here.

There are also some other conditions in the `WHERE` clause:

```
AND paymentStatus = 1 AND paymentPeriod > 0 AND userID IS NOT NULL
```

These are mostly specific to our business case. We want to include only the `paymentAmount` of successful payments (`paymentStatus = 1`) and exclude payments for specific non-recurring products (`paymentPeriod > 0`) as well as payments without an assigned `userID (userID IS NOT NULL)`.

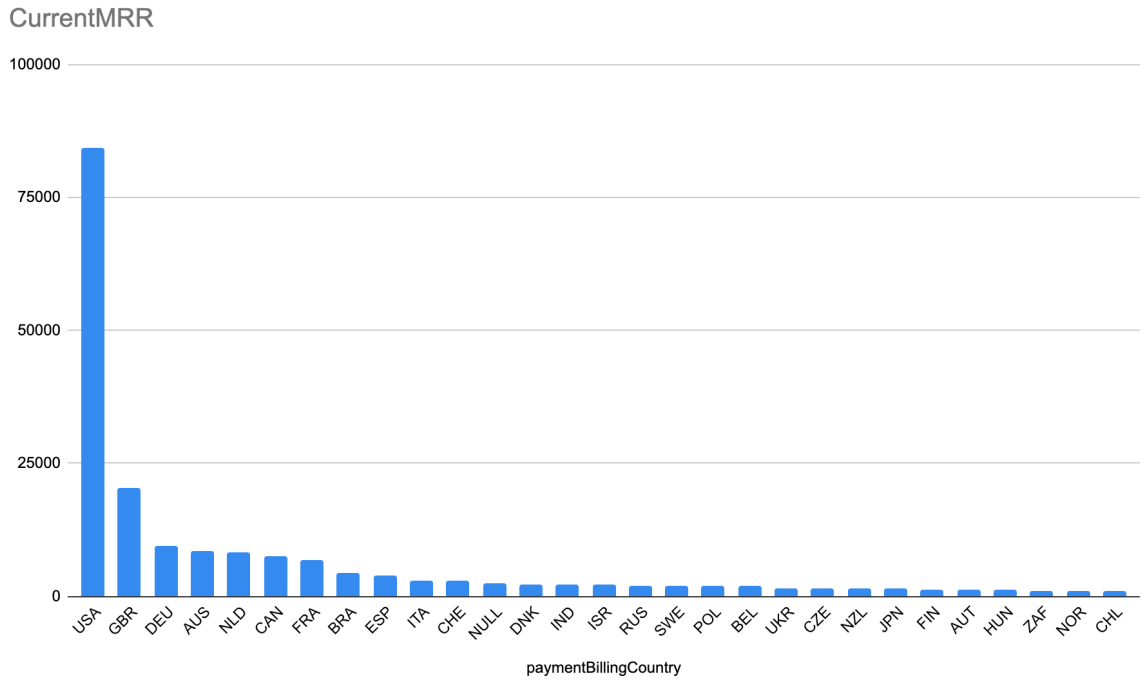
In the last line of code, we order the result based on the `CurrentMRR` (`ORDER BY CurrentMRR DESC`).

The whole query is in parentheses, followed by `AS arpu`. If we want to run this query by itself, we will need to remove this. It's there because we will need it later, when we will be joining it with another query.

paymentBillingCountry	Total_Current_Customers	CurrentMRR
USA	8483	84222
GBR	2024	20393
DEU	1139	9541
AUS	936	8599
NLD	851	8183
CAN	896	7583
FRA	655	6926
BRA	466	4437
ESP	317	3872
ITA	318	2903
CHE	324	2863
(NULL)	225	2450
DNK	246	2292
IND	268	2283
ISR	164	2155
RUS	196	2031
SWE	230	2006
POL	233	1905
BEL	190	1866
UKR	109	1596
CZE	175	1537
NZL	179	1508
JPN	131	1394
FIN	99	1262

Are you missing the ARPU? That's correct. We will compute it later for each country in the master query by dividing `CurrentMRR` by `Total_Current_Customers`. This gives us the average monthly recurring revenue (average MRR) generated by current (active) customers for each country.

When you get this result, you can just copy/paste it into the Google Sheets or Excel and make a nice visualization of this analysis. I am sure your stakeholders will love it. Even for you, it will be much more insightful.



Computing Average Customer Lifetime

Once we have the ARPU by country ready, we need to get the **average customer lifetime.** This metric is **the average length of time a customer is active/paying.** I will describe two solutions:

> Computing the **difference between the first and the last payment** the customer made.
>
> **The average of all the payments** the customer made during the payment period.

It's up to you (and the data you have available) which solution you choose. I began with the first one, but later I realized it is not 100% precise. It does not take into account the situations when users stop paying for a while and then later renew their subscription.

Since we store the number of months the customer has pre-paid the payment in the `paymentPeriod` field, I was able to come up with a second solution. Don't worry if you do not have this data available. I was comparing it and I can tell you that there were really minor differences. If you are analyzing a big enough data set, you should be safe.

Let's look at the first solution:

```
(SELECT paymentBillingCountry, AVG(LT.LifeTime) AS LT
FROM

(SELECT MinP.userID, MinP.paymentBillingCountry,
((DATEDIFF(MAX(MinP.paymentDateTime), MIN(MinP.paymentDateTime))/ 30.5) +
LP.paymentPeriod) AS LifeTime
    FROM payments MinP
    LEFT JOIN

(SELECT p1.userID, p1.paymentBillingCountry, DATE_FORMAT(p1.paymentDateTime,
'%d/%m/%Y') AS paymentDateTime, p1.paymentPeriod
FROM payments p1
LEFT JOIN payments p2 ON (p1.userID = p2.userID AND p1.paymentDateTime <
p2.paymentDateTime AND p2.paymentType = 2)
WHERE p2.paymentDateTime IS NULL AND p1.paymentDateTime IS NOT NULL AND
p1.paymentPeriod > 0
GROUP BY p1.userID
ORDER BY p1.paymentID DESC) LP

ON MinP.userID = LP.userID

GROUP BY MinP.userID) LT

GROUP BY paymentBillingCountry
ORDER BY paymentBillingCountry) avgLT
```

I will start explaining from the `/*last payment with Payment Period*/` part. The task seems to be pretty simple, but it is not a trivial task. I need to add the `paymentPeriod` of the last payment customer made to the difference between the last and first payment they made.

I was dealing with a similar problem and found the solution on Stack Overflow. It's a problem of getting records with max value for each group of grouped SQL results. This is the solution that worked for me:

The correct solution is:

336

```
SELECT o.*
FROM `Persons` o                    # 'o' from 'oldest person in group'
  LEFT JOIN `Persons` b             # 'b' from 'bigger age'
      ON o.Group = b.Group AND o.Age < b.Age
WHERE b.Age is NULL                 # bigger age not found
```

How it works:

It matches each row from `o` with all the rows from `b` having the same value in column `Group` and a bigger value in column `Age`. Any row from `o` not having the maximum value of its group in column `Age` will match one or more rows from `b`.

The `LEFT JOIN` makes it match the oldest person in group (including the persons that are alone in their group) with a row full of `NULL`s from `b` ('no biggest age in the group').
Using `INNER JOIN` makes these rows not matching and they are ignored.

The `WHERE` clause keeps only the rows having `NULL`s in the fields extracted from `b`. They are the oldest persons from each group.

Source: StackOverflow.com

To be honest, I did not understand it until I read the great explanation of self-JOINs that I mentioned earlier. Thanks to this article, I was able to visualize this problem and the solution.

I12

	A	B	C	D	E	F	G	H
1		Person.o				Person.b		
2	**Person**	**Group**	**Age**		**Person**	**Group**	**Age**	
3	~~Bob~~	~~1~~	~~32~~		~~Bob~~	~~1~~	~~32~~	
4	Bob	1	32		Jill	1	34	
5	Bob	1	32		Shawn	1	42	
6	~~Jill~~	~~1~~	~~34~~		~~Bob~~	~~1~~	~~32~~	
7	~~Jill~~	~~1~~	~~34~~		~~Jill~~	~~1~~	~~34~~	
8	Jill	1	34		Shawn	1	42	
9	~~Shawn~~	~~1~~	~~42~~		~~Bob~~	~~1~~	~~32~~	
10	~~Shawn~~	~~1~~	~~42~~		~~Jill~~	~~1~~	~~34~~	
11	~~Shawn~~	~~1~~	~~42~~		~~Shawn~~	~~1~~	~~42~~	
12								
13								
14								
15	Jake	2	29		Jake	2	29	
16	Paul	2	36		Paul	2	36	
17	Laura	2	39		Laura	2	39	
18								
19								
20								

I get the customer's **last payment date** alongside the **payment period for this payment** with the following part of the code.

```
(SELECT p1.userID, p1.paymentBillingCountry, DATE_FORMAT(p1.paymentDateTime,
'%d/%m/%Y') AS paymentDateTime, p1.paymentPeriod
FROM payments p1
LEFT JOIN payments p2 ON (p1.userID = p2.userID AND p1.paymentDateTime <
p2.paymentDateTime AND p2.paymentType = 2)
WHERE p2.paymentDateTime IS NULL AND p1.paymentDateTime IS NOT NULL AND
p1.paymentPeriod > 0
GROUP BY p1.userID
ORDER BY p1.paymentID DESC) LP
```

The LP at the end is the alias of this new temporary table and it means LastPayment. This new table is joined with the original payment table:

```
(SELECT p1.userID, p1.paymentBillingCountry, DATE_FORMAT(p1.paymentDateTime,
'%d/%m/%Y') AS paymentDateTime, p1.paymentPeriod
FROM payments p1
LEFT JOIN payments p2 ON (p1.userID = p2.userID AND p1.paymentDateTime <
p2.paymentDateTime AND p2.paymentType = 2)
WHERE p2.paymentDateTime IS NULL AND p1.paymentDateTime IS NOT NULL AND
p1.paymentPeriod > 0
GROUP BY p1.userID
ORDER BY p1.paymentID DESC) LP

ON MinP.userID = LP.userID

GROUP BY MinP.userID) LT
```

In this part, I compute the difference between the last and first payment the customer made. I get results in the day format, so I divide it by the average number of days in a month (30.5) and add the `paymentPeriod`.

```
((DATEDIFF(MAX(MinP.paymentDateTime), MIN(MinP.paymentDateTime))/ 30.5) +
LP.paymentPeriod)
```

This gives me the lifetime for each customer and their `paymentBillingCountry`:

```sql
SELECT MinP.userID, MinP.paymentBillingCountry,((DATEDIFF(MAX(MinP.paymentDateTime), MIN(MinP.paymentDateTime))/ 30.5) + LP.paymentPeriod) AS LifeTime
    FROM payments MinP
    LEFT JOIN

/*last payment with Payment Period*/
(select p1.userID, p1.paymentBillingCountry, DATE_FORMAT(p1.paymentDateTime, '%d/%m/%Y') as paymentDateTime, p1.paymentPeriod
from payments p1
left join payments p2 on (p1.userID = p2.userID and p1.paymentDateTime < p2.paymentDateTime AND p2.paymentType = 2)
where p2.paymentDateTime IS NULL AND p1.paymentDateTime IS NOT NULL AND p1.paymentPeriod > 0
group by p1.userID
order by p1.paymentID DESC) LP

on MinP.userID = LP.userID

GROUP BY MinP.userID
```

Message | Result 1 | Profile | Status

userID	paymentBillingCountry	LifeTime
187834	ESP	25.4590
65218	CZE	77.2459
11733	GBR	74.3115
135439	POL	4.9344
61314	BEL	77.2459
172045	GBR	85.5738
64561	SWE	12.0000
99267	NLD	81.4754
148563	DNK	80.4918
77680	FRA	80.9508
189160	AUS	77.3279
161052	AUS	2.1803
189327	DEU	23.9672
38396	ITA	84.2295
162955	TUR	68.9180

But I need the average by country. I achieve this one level higher in my nested query by selecting this ...

```
(SELECT paymentBillingCountry, AVG(LT.LifeTime) AS LT
```

... and grouping it by country:

```
GROUP BY paymentBillingCountry
```

I also order it by country:

```
ORDER BY paymentBillingCountry
```

And when I run the whole query, I get the desired result: the average customer lifetime by country.

paymentBillingCountry	LT
ABW	29.98360000
AFG	20.17760000
AGO	18.40165000
ALB	17.38852000
AND	8.58032000
ANT	23.88632667
ARE	14.45509292
ARG	19.83641957
ARM	16.41685714
AUS	24.57155614
AUT	24.35185622
AZE	28.56966250
BDI	2.49180000
BEL	27.60581080
BEN	15.98360000
BGD	17.78454286
BGR	22.10769535
BHR	20.15162500
BHS	27.73225000
BIH	23.41452143
BLR	16.12078947
BLZ	40.76230000

This method would be much simpler if we did not have to consider the last payment period in the computation. Fortunately, the second solution doesn't:

```
(SELECT ULT.paymentBillingCountry, AVG(ULT.UserLifeTime) AvgLifeTimeByCountry
FROM
    (SELECT userID, SUM(paymentPeriod) UserLifeTime, paymentBillingCountry
    FROM payments
    GROUP BY userID
    ORDER BY userID) ULT
GROUP BY paymentBillingCountry) ULT1
```

We start with counting the SUM of the payment period for all payments the customer made. I call this metric `UserLifetime`. To get the desired result, we need to group it by `userID`:

```
(SELECT userID, SUM(paymentPeriod) UserLifeTime, paymentBillingCountry
FROM payments
GROUP BY userID
ORDER BY userID) ULT
```

We use this new table in the `FROM` clause to compute the average lifetime of customers by country.

```
/*AVG LifeTime by Country based on SUM of PaymentPeriod)*/

(SELECT ULT.paymentBillingCountry, AVG(ULT.UserLifeTime) AvgLifeTimeByCountry
FROM
    (SELECT userID, SUM(paymentPeriod) UserLifeTime, paymentBillingCountry
    FROM payments
    GROUP BY userID
    ORDER BY userID) ULT
GROUP BY paymentBillingCountry) ULT1
```

And here is the result:

paymentBillingCountry	AvgLifeTimeByCountry
ABW	30.0000
AFG	17.8333
AGO	14.7500
ALB	28.5000
AND	11.0000
ANT	36.2143
ARE	15.0252
ARG	19.1745
ARM	11.2500
AUS	21.1110
AUT	22.0449
AZE	24.0000
BDI	2.5000
BEL	23.2896
BEN	11.0000
BGD	16.8261
BGR	20.2059
BHR	19.3750
BHS	25.6667
BIH	23.6429
BLR	15.6053
BLZ	33.2000
BMU	21.0000

You can see that there are not any big differences for most countries. But in Albania (ALB), the difference between the two methods is quite significant (17.38 vs. 28.5). This is caused by the small number of

customers from this country (5). As I said, you should be safe if you have enough data. If we look at the USA, where we have most of our customers (8,483), the difference in lifetime between the two methods is almost nothing (21.63 vs. 21.16). It makes sense that the first method gives a slightly higher result, since it doesn't account for customers that stop and then renew their subscription.

You've already learned the basics of SQL, and you ask, "What's next?" Here's the answer: SQL Fundamentals track!

Putting It All Together: Calculating LTV with SQL

Now that we have prepared the following by country ...

`CurrentMRR`

`Total_Current_Customers`

`Average Customer Lifetime`

... we just need to join everything in one master query:

```
SELECT arpu.paymentBillingCountry, arpu.Total_Current_Customers,
(arpu.CurrentMRR/arpu.Total_Current_Customers) AS CurrentAvgMRR_ARPU,
ULT1.AvgLifeTimeByCountry AS LifeTime, (ULT1.AvgLifeTimeByCountry *
(arpu.CurrentMRR/arpu.Total_Current_Customers)) AS LTV
FROM

(SELECT paymentBillingCountry, COUNT(DISTINCT userID) Total_Current_Customers,
ROUND(SUM(paymentAmount / paymentPeriod)) AS CurrentMRR
FROM payments
WHERE paymentDateTime < CURDATE() AND DATE_ADD(paymentDateTime, INTERVAL
paymentPeriod MONTH) >= CURDATE() AND paymentStatus = 1 AND paymentPeriod > 0 AND
userID IS NOT NULL
GROUP BY paymentBillingCountry
ORDER BY CurrentMRR DESC) AS arpu

LEFT JOIN

(SELECT ULT.paymentBillingCountry, AVG(ULT.UserLifeTime) AvgLifeTimeByCountry
FROM
    (SELECT userID, SUM(paymentPeriod) UserLifeTime, paymentBillingCountry
    FROM payments
    GROUP BY userID
    ORDER BY userID) ULT
GROUP BY paymentBillingCountry) ULT1
```

```
ON ULT1.paymentBillingCountry = arpu.paymentBillingCountry
```

And when I run it, I get this result:

paymentBillingCountry	Total_Current_Customers	CurrentAvgMRR_ARPU	LifeTime	LTV
USA	8511	10.0087	21.1700	211.8841
GBR	2032	9.9040	22.8355	226.1636
DEU	1149	8.3272	20.4992	170.7018
AUS	927	9.1791	21.1110	193.7794
NLD	856	9.6460	23.9784	231.2963
CAN	905	8.5326	23.0634	196.7907
FRA	655	10.6260	22.4214	238.2488
BRA	468	9.5641	17.8158	170.3921
ESP	318	12.0723	21.2661	256.7313
(NULL)	220	14.1409	(NULL)	(NULL)
ITA	324	9.2377	23.3117	215.3454
CHE	332	8.9608	23.0222	206.2983
DNK	247	9.3927	25.5046	239.5574
IND	268	8.4328	20.9002	176.2480
ISR	168	12.7798	19.7262	252.0961
POL	235	8.5702	18.8945	161.9299
SWE	233	8.3777	23.4734	196.6527
RUS	191	10.1466	15.8796	161.1239
BEL	192	9.8125	23.3284	228.9099
CZE	179	8.7933	20.5963	181.1094
UKR	106	14.8302	15.4427	229.0182
NZL	179	8.4637	22.7231	192.3212
JPN	133	10.6165	23.2234	246.5522
FIN	100	12.7100	22.8486	290.4057
AUT	164	7.3598	22.0449	162.2451

I can see where most of our customers come from and how the Current MRR, Lifetime and LTV values differ across countries.

Discover how you can use SQL in real business situations. Try our SQL Reporting track and learn queries you'll actually use.

Ready to Write Your Own KPI SQL Queries?

And here you have a recipe for computing customer lifetime value and some accompanying SaaS KPIs. I hope you've enjoyed my experiences and how I've used SQL in these queries. Or maybe you have an idea how to write these queries better? Let me know in the comments.

Stay tuned for the next article from our SaaS KPIs metrics series. I've already started working on it.

In the meantime, if you are interested in data analysis, I recommend LearnSQL.com's SQL Reporting track. This is a great set of courses and interactive exercises. In it, you'll learn to create SQL reports, to perform trend analysis using SQL, and how to analyze the customer lifecycle.

Remember, learning SQL and mastering databases just pays off!
