**10 Easy Steps to a Complete Understanding of SQL**

**FROM:** [**http://tech.pro/tutorial/1555/10-easy-steps-to-a-complete-understanding-of-sql?utm\_content=bufferadfd7&utm\_source=buffer&utm\_medium=twitter&utm\_campaign=Buffer**](http://tech.pro/tutorial/1555/10-easy-steps-to-a-complete-understanding-of-sql?utm_content=bufferadfd7&utm_source=buffer&utm_medium=twitter&utm_campaign=Buffer)



Too many programmers think SQL is a bit of a beast. It is one of the few [*declarative languages*](https://en.wikipedia.org/wiki/Declarative_programming) out there, and as such, behaves in an entirely different way from imperative, object-oriented, or even functional languages (although, some say that SQL is also [somewhat functional](http://thoughts.davisjeff.com/2011/09/25/sql-the-successful-cousin-of-haskell/)).

I'm writing SQL every day and embracing SQL with my [Open Source Software jOOQ](http://www.jooq.org). I thus feel compelled to bring the beauty of SQL a bit closer to those of you still struggling with it. The following tutorial is destined for

* readers who have already worked with SQL but never completely understood it
* readers who know SQL well but have never really thought about its syntax
* readers who want to teach SQL to others

This tutorial will focus on SELECT statements only. Other DML statements will be covered in another tutorial. Here are...

**10 Easy Steps to a Complete Understanding of SQL.**

**1. SQL is declarative**

Get this into your head first. Declarative. The only paradigm where you "just" declare the nature of the results that you would like to get. Not *how* your computer shall compute those results. Isn't that wonderful?

SELECT first\_name, last\_name FROM employees WHERE salary > 100000

Easy to understand. You don't care where employee records physically come from. You just want those that have a decent salary.

**What do we learn from this?**

So if this is so simple, what's the problem? The problem is that most of us intuitively think in terms of [*imperative programming*](https://en.wikipedia.org/wiki/Imperative_programming). As in: *"machine, do this, and then do that, but before, run a check and fail if this-and-that"*. This includes storing temporary results in variables, writing loops, iterating, calling functions, etc. etc.

Forget about all that. Think about *how to declare* things. Not about *how to tell the machine* to compute things.

**2. SQL syntax is not "well-ordered"**

A common source of confusion is the simple fact that SQL syntax elements are not ordered in the way they are executed. The lexical ordering is:

* SELECT [ DISTINCT ]
* FROM
* WHERE
* GROUP BY
* HAVING
* UNION
* ORDER BY

For simplicity, not all SQL clauses are listed. This lexical ordering differs fundamentally from the logical order, i.e. from the order of execution:

* FROM
* WHERE
* GROUP BY
* HAVING
* SELECT
* DISTINCT
* UNION
* ORDER BY

There are three things to note:

1. FROM is the first clause, not SELECT. The first thing that happens is loading data from the disk into memory, in order to operate on such data.
2. SELECT is executed after most other clauses. Most importantly, after FROM and GROUP BY. This is important to understand when you think you can reference stuff that you declare in the SELECT clause from the WHERE clause. The following is not possible:
3. SELECT A.x + A.y AS z
4. FROM A

WHERE z = 10 -- z is not available here!

If you wanted to reuse z, you have two options. Either repeat the expression:

SELECT A.x + A.y AS z

FROM A

WHERE (A.x + A.y) = 10

... or you resort to derived tables, common table expressions, or views to avoid code repetition. See examples further down.

1. UNION is placed before ORDER BY in both lexical and logical ordering. Many people think that each UNION subselect can be ordered, but according to the SQL standard and most SQL dialects, that is not true. While some dialects allow for ordering *subqueries* or *derived tables*, there is no guarantee that such ordering will be retained after a UNION operation

Note, not all databases implement things the same way. Rule number 2, for instance, does not apply exactly in the above way to MySQL, PostgreSQL, and SQLite.

**What do we learn from this?**

Always remember both the *lexical order* and the *logical order* of SQL clauses to avoid very common mistakes. If you understand that distinction, it will become very obvious why some things work and others don't.

Of course, it would have been nice if the language was designed in a way that the *lexical order* actually reflected the *logical order*, as it is implemented in Microsoft's [LINQ](http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx).

**3. SQL is about table references**

Because of the difference between *lexical ordering* and *logical ordering*, most beginners are probably tricked into thinking that column values are the first-class citizens in SQL. They are not. The most important things are table references.

The [SQL standard](http://www.andrew.cmu.edu/user/shadow/sql/sql1992.txt) defines the FROM clause as such:

<from clause> ::= FROM <table reference> [ { <comma> <table reference> }... ]

The "output" of the FROM clause is a combined table reference of the combined degree of all table references. Let's digest this, slowly.

FROM a, b

The above produces a combined table reference of the degree of a + the degree of b. If a has 3 columns and b has 5 columns, then the "output table" will have 8 (3 + 5) columns.

The records contained in this combined table reference are those of the cross product / cartesian product of a x b. In other words, each record of a is paired with each record of b. If a has 3 records and b has 5 records, then the above combined table reference will produce 15 records (3 x 5).

This "output" is "fed" / "piped" into the GROUP BY clause (after filtering in the WHERE clause), where it is transformed into a new "output". We'll deal with that later on.

If we're looking at these things from a [*relational algebra*](http://en.wikipedia.org/wiki/Relational_algebra) / [*set theory*](http://en.wikipedia.org/wiki/Set_theory) perspective, a SQL table is a *relation* or a *set of tuples*. And each SQL clause will transform one or several relations in order to produce new relations.

**What do we learn from this?**

Always think in terms of table references to understand how data is "pipelined" through your SQL clauses.

**4. SQL table references can be rather powerful**

A table reference is something rather powerful. A simple example of their power is the JOIN keyword, which is actually not part of the SELECT statement, but part of a "special" table reference. The joined table, as defined in the [SQL standard](http://www.andrew.cmu.edu/user/shadow/sql/sql1992.txt) (simplified):

<table reference> ::=

<table name>

| <derived table>

| <joined table>

If we take again the example from before:

FROM a, b

a can be a joined table as such:

a1 JOIN a2 ON a1.id = a2.id

Expanding this into the previous expression, we'd get:

FROM a1 JOIN a2 ON a1.id = a2.id, b

While it is discouraged to combine the comma-separated list of table references syntax with the joined table syntax, you can most certainly do this. The resulting, combined table reference will now have a degree of a1+a2+b.

Derived tables are even more powerful than joined tables. We'll get to that.

**What do we learn from this?**

Always, always think in terms of table references. Not only is this important to understand how data is "pipelined" through your SQL clauses (see previous section), it will also help you understand how complex table references are constructed.

And, importantly, understand that JOIN is a keyword for constructing joined tables. Not a part of the SELECT statement. Some databases allow for using JOIN in INSERT, UPDATE, DELETE

**5. SQL JOIN tables should be used rather than comma-separated tables**

Before, we've seen this clause:

FROM a, b

Advanced SQL developers will probably tell you that it is discouraged to use the comma-separated list at all, and always fully express your JOINs. This will help you improve readability of your SQL statement, and thus prevent mistakes.

One very common mistake is to forget a JOIN predicate somewhere. Think about the following:

FROM a, b, c, d, e, f, g, h

WHERE a.a1 = b.bx

AND a.a2 = c.c1

AND d.d1 = b.bc

-- etc...

The join table syntax is both

* Safer, as you can place join predicates close to the joined tables, thus preventing mistakes.
* More expressive, as you can distinguish between OUTER JOIN, INNER JOIN, etc.

**What do we learn from this?**

Always use JOIN. Never use comma-separated table references in your FROM clauses.

**6. SQL's different JOIN operations**

JOIN operations essentially come with five flavours:

* EQUI JOIN
* SEMI JOIN
* ANTI JOIN
* CROSS JOIN
* DIVISION

These terms are commonly used in [*relational algebra*](https://en.wikipedia.org/wiki/Relational_algebra). SQL uses different terms for the above concepts, if they exist at all. Let's have a closer look:

**EQUI JOIN**

This is the most common JOIN operation. It has two sub-flavours:

* INNER JOIN (or just JOIN)
* OUTER JOIN (further sub-flavoured as LEFT, RIGHT, FULL OUTER JOIN)

The difference is best explained by example:

-- This table reference contains authors and their books.

-- There is one record for each book and its author.

-- authors without books are NOT included

author JOIN book ON author.id = book.author\_id

-- This table reference contains authors and their books

-- There is one record for each book and its author.

-- ... OR there is an "empty" record for authors without books

-- ("empty" meaning that all book columns are NULL)

author LEFT OUTER JOIN book ON author.id = book.author\_id

**SEMI JOIN**

This relational concept can be expressed in two ways in SQL: Using an IN predicate, or using an EXISTS predicate. "Semi" means "half" in latin. This type of join is used to join only "half" of a table reference. What does that mean? Consider again the above joining of author and book. Let's imagine that we don't want author/book combinations, but just those authors who actually also have books. Then we can write:

-- Using IN

FROM author

WHERE author.id IN (SELECT book.author\_id FROM book)

-- Using EXISTS

FROM author

WHERE EXISTS (SELECT 1 FROM book WHERE book.author\_id = author.id)

While there is no general rule as to whether you should prefer IN or EXISTS, these things can be said:

* IN predicates tend to be more readable than EXISTS predicates
* EXISTS predicates tend to be more expressive than IN predicates (i.e. it is easier to express very complex SEMI JOIN)
* There is no formal difference in performance. There may, however, be a [huge performance difference on some databases](http://explainextended.com/2009/09/18/not-in-vs-not-exists-vs-left-join-is-null-mysql/).

Because INNER JOIN also produces only those authors that actually have books, many beginners may think that they can then remove duplicates using DISTINCT. They think they can express a SEMI JOIN like this:

-- Find only those authors who also have books

SELECT DISTINCT first\_name, last\_name

FROM author

JOIN book ON author.id = book.author\_id

This is very bad practice for two reasons:

* It is very slow, as the database has to load a lot of data into memory, just to remove duplicates again.
* It is not entirely correct, even if it produces the correct result in this simple example. But as soon as you JOIN more table references, you will have a very hard time correctly removing duplicates from your results.

Some more information about abuse of DISTINCT can be [seen in this blog post](http://blog.jooq.org/2013/07/30/10-common-mistakes-java-developers-make-when-writing-sql/).

**ANTI JOIN**

This relational concept is just the opposite of a SEMI JOIN. You can produce it simply by adding a NOT keyword to the IN or EXISTS predicates. An example, where we'll select those authors who do not have any books:

-- Using IN

FROM author

WHERE author.id NOT IN (SELECT book.author\_id FROM book)

-- Using EXISTS

FROM author

WHERE NOT EXISTS (SELECT 1 FROM book WHERE book.author\_id = author.id)

The same rules with respect to performance, readability, expressivity apply. However, there is a small caveat with respect to NULLs when using NOT IN, [which is a bit out of scope for this tutorial](http://blog.jooq.org/2012/01/27/sql-incompatibilities-not-in-and-null-values/).

**CROSS JOIN**

This produces a cross product of the two joined table references, combining every record of the first table reference with every record of the second table reference. We have seen before, that this can be achieved with comma-separated table references in the FROM clause. In the rare cases where this is really desired, you can also write a CROSS JOIN explicitly, in most SQL dialects:

-- Combine every author with every book

author CROSS JOIN book

**DIVISION**

The relational division is really a beast of its own breed. In short, if JOIN is multiplication, division is the inverse of JOIN. Relational divisions are very tough to express in SQL. As this is a beginners' tutorial, explaining it is out of scope. For the brave among you, [read on about it here](http://blog.jooq.org/2012/03/30/advanced-sql-relational-division-in-jooq/), [here](http://en.wikipedia.org/wiki/Relational_algebra#Division), [and here](https://www.simple-talk.com/sql/t-sql-programming/divided-we-stand-the-sql-of-relational-division/).

**What do we learn from this?**

A lot. Again, let's hammer this into our heads. SQL is about table references. Joined tables are quite sophisticated table references. But there is a difference in relational-speak and SQL-speak. Not all relational join operations are also formal SQL join operations. With a bit of practice and knowledge about relational theory, you will always be able to choose the right type of relational JOIN and be able to translate it to the correct SQL.

**7. SQL's derived tables are like table variables**

Before, we've learned that SQL is a *declarative language*, and as such, variables do not have a place (they do in some SQL dialects, though). But you can write something *like* variables. And those beasts are called derived tables.

A derived table is nothing but a subquery wrapped in parentheses.

-- A derived table

FROM (SELECT \* FROM author)

Note that some SQL dialects require derived tables to have a *correlation name* (also known as alias).

-- A derived table with an alias

FROM (SELECT \* FROM author) a

Derived tables are awesome when you want to circumvent the problems caused by the *logical ordering* of SQL clauses. For instance, if you want to reuse a column expression in both the SELECT and the WHERE clause, just write (Oracle dialect):

-- Get authors' first and last names, and their age in days

SELECT first\_name, last\_name, age

FROM (

SELECT first\_name, last\_name, current\_date - date\_of\_birth age

FROM author

)

-- If the age is greater than 10000 days

WHERE age > 10000

Note that some databases, and the SQL:1999 standard have taken derived tables to the next level, introducing *common table expressions*. This will allow you to reuse the same *derived table* several times within a single SQL SELECT statement. The above query would then translate to the (almost) equivalent:

WITH a AS (

SELECT first\_name, last\_name, current\_date - date\_of\_birth age

FROM author

)

SELECT \*

FROM a

WHERE age > 10000

Obviously, you could also externalise "a" into a standalone view for even broader reuse of common SQL subselects. [Read more about views here](http://en.wikipedia.org/wiki/View_%28SQL%29).

**What do we learn from this?**

Again, again, again. SQL is mostly about table references, not columns. Make use of them. Don't be afraid of writing derived tables or other complex table references.

**8. SQL GROUP BY transforms previous table references**

Let's reconsider our previous FROM clause:

FROM a, b

And now, let's apply a GROUP BY clause to the above combined table reference

GROUP BY A.x, A.y, B.z

The above produces a new table reference with only three remaining columns (!). Let's digest this again. If you apply GROUP BY, then you reduce the number of available columns in all subsequent logical clauses - including SELECT. This is the syntactical reason why you can only reference columns from the GROUP BY clause in the SELECT clause.

* Note that other columns may still be available as arguments of aggregate functions:
* SELECT A.x, A.y, SUM(A.z)
* FROM A

GROUP BY A.x, A.y

* Note that [MySQL, unfortunately, doesn't adhere to this standard](http://blog.jooq.org/2012/08/05/mysql-bad-idea-384/), causing nothing but confusion. Don't fall for MySQL's tricks. GROUP BY transforms table references. You can thus only reference columns also referenced in the GROUP BY clause.

**What do we learn from this?**

GROUP BY, again, operates on table references, transforming them into a new form.

**9. SQL SELECT is called projection in relational algebra**

I personally like the term "projection", as it is used in relational algebra. Once you've generated your table reference, filtered it, transformed it, you can step to projecting it to another form. The SELECT clause is like a projector. A table function making use of a *row value expression* to transform each record from the previously constructed table reference into the final outcome.

Within the SELECT clause, you can finally operate on columns, creating complex column expressions as parts of the record / row.

There are a lot of special rules with respect to the nature of available expressions, functions, etc. Most importantly, you should remember these:

1. You can only use column references that can be produced from the "output" table reference
2. If you have a GROUP BY clause, you may only reference columns from that clause, or aggregate functions.
3. You can use window functions instead of aggregate functions, when you don't have a GROUP BY clause.
4. If you don't have a GROUP BY clause, you must not combine aggregate functions with non-aggregate functions.
5. There are some rules with respect to wrapping regular functions in aggregate functions and vice-versa.
6. There are ...

Well, there are lots of complex rules. They could fill yet another tutorial. For instance, the reason why you cannot combine aggregate functions with non-aggregate functions in the projection of a SELECT statement without GROUP BY clause (rule number 4) is this:

1. It doesn't make sense. Intuitively.
2. If intuition doesn't help (it hardly does, with a SQL beginner), then syntax rules do. SQL:1999 introduced GROUPING SETS, and SQL:2003 introduced empty grouping sets: GROUP BY (). Whenever an aggregate function is present, and there is no explicit GROUP BY clause, an implicit, empty GROUPING SET is applied (rule number 2). Hence, the original rules about *logical ordering* aren't exactly true anymore, and the projection (SELECT) influences the outcome of a logically preceding, yet lexically succeeding clause (GROUP BY).

Confused? Yes. Me too. Let's get back to simpler things.

**What do we learn from this?**

The SELECT clause may be one of the most complex clauses in SQL, even if it appears so simple. All other clauses just "pipe" table references from one to another. The SELECT clause messes up the beauty of these table references, by completely transforming them, applying some rules to them retroactively.

In order to understand SQL, it is important to understand everything *else* first, before trying to tackle SELECT. Even if SELECT is the first clause in lexical ordering, it should be the last.

**10. SQL DISTINCT, UNION, ORDER BY, and OFFSET are simple again**

After the complicated SELECT, we can get back to simple things again:

* Set operations (DISTINCT and UNION)
* Ordering operations (ORDER BY, OFFSET .. FETCH)

**Set operations**

Set operations operate on "sets", which are actually nothing other than... tables. Well, almost. Conceptually, they're easy to understand.

* DISTINCT removes duplicates *after* the projection.
* UNION concatenates two subselects and removes duplicates
* UNION ALL concatenates two subselects retaining duplicates
* EXCEPT removes records from the first subselect that are also contained in the second subselect (and then removes duplicates)
* INTERSECT retains only records contained in both subselects (and then removes duplicates)

All of this removing duplicates is usually non-sense. Most often, you should just use UNION ALL, when you want to concatenate subselects.

**Ordering operations**

Ordering is not a relational feature. It is a SQL-only feature. It is applied at the very end of both *lexical ordering* and *logical ordering* of your SQL statement. Using ORDER BY and OFFSET .. FETCH is the only way to guarantee that records can be accessed by index in a reliable way. All other ordering is always arbitrary and random, even if it may appear to be reproducible.

OFFSET .. FETCH is only one syntax variant. Other variants include MySQL's and PostgreSQL's LIMIT .. OFFSET, or SQL Server's and Sybase's TOP .. START AT. A good overview of various ways to implement OFFSET .. FETCH [can be seen here](http://www.jooq.org/doc/3.1/manual/sql-building/sql-statements/select-statement/limit-clause/).

**Let's get to work**

As with every language, SQL takes a lot of practice to master. The above 10 simple steps will help you make more sense of the every day SQL that you're writing. On the other hand, it is also good to learn from common mistakes. The following two articles list lots of common mistakes Java (and other) developers make when writing SQL:

* [10 Common Mistakes Java Developers Make when Writing SQL](http://blog.jooq.org/2013/07/30/10-common-mistakes-java-developers-make-when-writing-sql/)
* [10 More Common Mistakes Java Developers Make when Writing SQL](http://blog.jooq.org/2013/08/12/10-more-common-mistakes-java-developers-make-when-writing-sql/)

[Tech.Pro](http://tech.pro/)

|  |  |  |
| --- | --- | --- |
| Unlike this Like this post |  |  |

**Related Posts**

* [[](http://tech.pro/blog/1616/the-premature-return-to-sql)](http://tech.pro/blog/1616/the-premature-return-to-sql)

**[The Premature Return to SQL](http://tech.pro/blog/1616/the-premature-return-to-sql)**

* [[](http://tech.pro/blog/1689/does-java-8-still-need-linq-or-is-it-better-than-linq)](http://tech.pro/blog/1689/does-java-8-still-need-linq-or-is-it-better-than-linq)

**[Does Java 8 Still Need LINQ? Or is it Better than LINQ?](http://tech.pro/blog/1689/does-java-8-still-need-linq-or-is-it-better-than-linq)**

* [[](http://tech.pro/tutorial/1631/5-claims-about-sql-explained)](http://tech.pro/tutorial/1631/5-claims-about-sql-explained)

**[5 Claims About SQL, Explained](http://tech.pro/tutorial/1631/5-claims-about-sql-explained)**

* [[](http://tech.pro/tutorial/1609/null-in-sql-explaining-its-behavior)](http://tech.pro/tutorial/1609/null-in-sql-explaining-its-behavior)

**[NULL in SQL: explaining its behavior](http://tech.pro/tutorial/1609/null-in-sql-explaining-its-behavior)**

* [[](http://tech.pro/tutorial/1142/building-faceted-search-with-postgresql)](http://tech.pro/tutorial/1142/building-faceted-search-with-postgresql)

**[Building Faceted Search With PostgreSQL](http://tech.pro/tutorial/1142/building-faceted-search-with-postgresql)**

Top of Form

**Login**







Login

Bottom of Form

Top of Form

**Register**











I agree to [Tech.Pro's Terms](http://tech.pro/terms-and-conditions) Register

Bottom of Form

**Comments (8)**

* [Pete Danes**Pete Danes**](http://tech.pro/petedanes) posted 2 months ago

Wow. This is one of the clearest explanations of SQL functioning that I have ever read. Especially #2, the part about SQL's execution order. After decades in the business, I finally understand WHY it's not possible to use an alias in the WHERE clause. Many thanks, sir, this humble and unworthy grasshopper grovels at your virtual feet.

I wonder how much teeth-gnashing and hair-pulling would have been averted over the years if the orginal SQL designers had thought to lay out their new language in the proper order. That is, if we today wrote our queries as: FROM MyTable WHERE column1 = 'abc' GROUP BY column2 HAVING column3 = 'def' SELECT column1 as C1, column2 as C2 ...

And I wonder, will any manufacturer of SQL products ever have the courage to propose and start using such a syntax? Or at least offer it as an alternative? I'd certainly be inclined to look carefully at such a product on the strength of such an offering alone.

And finally, I think I'm going to start writing my from-scratch queries that way, that is, in that order. I'll have to skip around a bit with the cursor to get the result correct, but it turns out that it even helps the Intellisense code offer useful things, if you create the query text in that order (in T-SQL, anyway). You have to put in the bare SELECT keyword, so that it understands what you're trying to write, but from there you can proceed in the 'sensible' order.

i Reply

* + [Lukas Eder**Lukas Eder**](http://tech.pro/lukaseder) 2 months ago

Glad you liked it! N.B: the term "execution order" from this article has been criticised, as the execution order is really implementation-dependent. I should have really distinguished between "lexical" and "logical" / "semantic" order.

I wonder how much teeth-gnashing and hair-pulling would have been averted over the years [...]

This is what Microsoft LINQ has attempted to do. Still, with UNIONS and ORDER BY / OFFSET .. FETCH being semantically ordered "behind" SELECT, the world wouldn't yet be perfect.

* [Amanda Maddox**Amanda Maddox**](http://tech.pro/amandamaddox) posted 2 months ago

Good article. The comments below may help people teach others about SQL.

Point 2.2 – When we teach SQL it is good to provide examples of how the data can be pulled even though “z” does not exist.

SELECT my\_table.x + my\_table.y AS z

FROM my\_table

WHERE (my\_table.x + my\_table.y) = 10

Point 4. "While it is discouraged to combine the comma-separated list of table references syntax with the joined table syntax, you can most certainly do this." Combining comma-separated list with joined table syntax is STRONGLY discouraged. If you are using this as a guide to teach others SQL, I would train your students NEVER to combine this syntax. Point 5 is your reason why not to do this.

Point 6. “There may, however, be a huge performance difference on some databases.” I would recommend every student read the referenced article. This can cause performance issues on enterprise level systems.

Point 7. Students may not understand “externalise 'a' into a view”. Maybe the author could define what a “view” is for students.

Point 8. Students may not understand “aggregate functions”. Maybe the author could provide an example SQL to clarify.

Select SUM(A.x) as TotalX, A.y, B.z

From A, B

GROUP BY A.x, A.y, B.z

Students may also be interested in learning about CASE statements, which can be very powerful in the “projection” of data. http://www.tizag.com/sqlTutorial/sqlcase.php

Point 10. I have to agree with Yawar Amin about the “union all” recommendation. Students may never need to use “union all”. "union" in most cases will work.

i Reply

* + [Lukas Eder**Lukas Eder**](http://tech.pro/lukaseder) 2 months ago

Point 2.2 – When we teach SQL it is good to provide examples of how the data can be pulled even though “z” does not exist.

Point 7. Students may not understand “externalise 'a' into a view”.

Point 8. Students may not understand “aggregate functions”.

Good points. Fixed.

Students may also be interested in learning about CASE statements, which can be very powerful in the “projection” of data.

True, but at that point, I wanted to get to an end with the article...

Point 10. I have to agree with Yawar Amin about the “union all” recommendation.

I'm really curious about real-world experience backing these claims. I understand the theory behind this rationale, but I think it is really impractical in actual RDBMS implementations, where UNION ALL is *always* faster (or rather: *never* slower) than UNION. I have tuned a few 1000-lines-of-code SQL monsters with triple-nested UNIONs to see that UNION ALL eases so much pain for Oracle 11g, removing costly sort operations that have to be applied early.

I don't fully understand the SQL transformation rules, but it appeared to me that Oracle has significant problems when applying predicate push-down operations, when subqueries contain unions...

Also, removing *unwanted* duplicates is a clear sign of code smell, indicating that the entire query is wrong. Teaching students to ignore that is what we pay for terribly, later on in the industry.

* [Yawar Amin**Yawar Amin**](http://tech.pro/yawaramin) posted 2 months ago

Great article, I just have a small issue with the union all recommendation: Date (2011, p. 119) advises *against* specifying all as it's almost always unnecessary.

Date, C. J. (2011). SQL and relational theory: how to write accurate SQL code. Sebastopol, CA: O’Reilly. Retrieved from http://proquest.safaribooksonline.com/9781449319724

i Reply

* + [Lukas Eder**Lukas Eder**](http://tech.pro/lukaseder) 2 months ago

Date (2011, p. 119) advises against specifying all as it's almost always unnecessary.

Huh! Well, the only reason *in favour* of UNION is the actual need to remove duplicates. Other than that (i.e. if duplicates are OK, or if duplicates are impossible), UNION is always slower than UNION ALL. What's the reason your author gives for their recommendation?

* + [Yawar Amin**Yawar Amin**](http://tech.pro/yawaramin) 2 months ago

@Lukas: Date makes the argument that we should always specify distinct wherever necessary, and never specify all. This eliminates all possibility of duplicate rows in the results, without us having to think about or understand the specifics of the tables we're querying. And we want to eliminate all duplicates because there are no such things as duplicates in the relational model. Date's goal with the book I mentioned is to guide people to following the relational model as closely as possible in their SQL queries. See e.g. p. 78.

* + [Lukas Eder**Lukas Eder**](http://tech.pro/lukaseder) 2 months ago

@Yawar:

Date makes the argument that we should always specify distinct wherever necessary, and never specify all.

Eeek!

without us having to think about or understand the specifics of the tables we're querying.

Yikes :-) Not having to think? Very interesting claim.

Date's goal with the book I mentioned is to guide people to following the relational model as closely as possible in their SQL queries.

OK, I understand his rationale, and being C.J. Date, he certainly knows a couple of things about the world he created with E.F. Codd and other creators of relational theory.

But it's utterly impractical in a real-world scenario. In other words: *"How to kill your database"*. I'll certainly read up on his claims and write a follow-up post on that topic. Thanks for the info!