**10 Cool SQL Optimisations That do not Depend on the Cost Model**

**FROM:** [**https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/)

[Cost Based Optimisation](https://en.wikipedia.org/wiki/Query_optimization) is the de-facto standard way to optimise SQL queries in most modern databases. It is the reason why it is really really hard to implement a complex, hand-written algorithm in a [3GL (third generation programming language)](https://en.wikipedia.org/wiki/Third-generation_programming_language) such as Java that outperforms a dynamically calculated database execution plan, that has been generated from a modern optimiser. I’ve recently delivered a talk about that topic:

Today, we don’t want to talk about cost based optimisation, i.e. optimisations that depend on a database’s cost model. We’ll look into much simpler optimisations that can be implemented purely based on meta data (e.g. constraints) and the query itself. They’re usually no-brainers for a database to optimise, because the optimisation will *always* lead to a better execution plan, independently of whether there are any indexes, or how much data you have, or how skewed your data distribution is.

So, they’re not no-brainers in the sense whether they’re easy for the optimiser teams to implement, but they’re no-brainers in the sense whether they *should* be done.

These optimisations remove needless, *optional* work (as opposed to [needless, *mandatory* work, which I’ve blogged about before](https://blog.jooq.org/2017/03/08/many-sql-performance-problems-stem-from-unnecessary-mandatory-work/))

**Where do these optimisations apply?**

Most of these optimisations are applied to:

* Fix mistakes in queries
* Allow for reusing complex views without actually executing the entire logic from the view

In the first case, you could claim: “Well, then fix the stupid SQL already”, but then again, who never makes any mistakes, right?

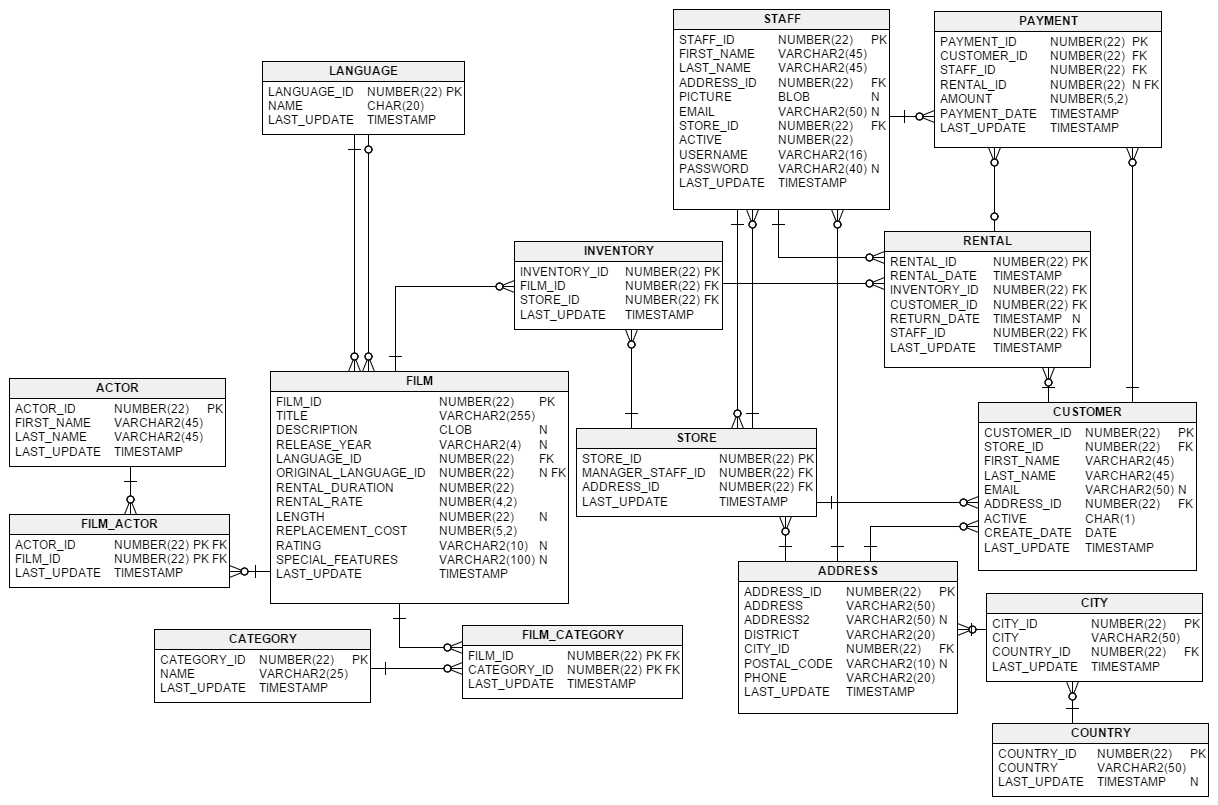
Specifically, the second case is really cool, as these optimisations allow us to build complex libraries of views and table valued functions, which we can reuse in several layers.

**Databases being used**

This post will evaluate 10 SQL optimisations on the 5 most popular RDBMS ([according to the db-engines ranking](http://db-engines.com/en/ranking)):

* Oracle 12.2
* MySQL 8.0.2
* SQL Server 2014
* PostgreSQL 9.6
* DB2 LUW 10.5

In all of this article, I will be using queries against the [Sakila database](https://www.jooq.org/sakila) – as always.

[](https://www.jooq.org/sakila)

These will be the 10 optimisation types:

1. [Transitive Closure](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top1)
2. [Impossible Predicates and Unneeded Table Accesses](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top2)
3. [JOIN Elimination](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top3)
4. [Removing “Silly” Predicates](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top4)
5. [Projections in EXISTS Subqueries](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top5)
6. [Predicate Merging](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top6)
7. [Provably Empty Sets](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top7)
8. [CHECK Constraints](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top8)
9. [Unneeded Self JOIN](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top9)
10. [Predicate Pushdown](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top10)

One final note before you move on: Many of the following examples might be too simple. Some databases (e.g. SQL Server) might not apply a specific optimisation on a query that is “too trivial”. [See also the comments for details](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151697).

[**1. Transitive Closure**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top1)

Let’s start with something simple: [transitive closure](https://en.wikipedia.org/wiki/Transitive_closure). It’s a really trivial concept that applies to a variety of maths operations, e.g. to the equality operator. It can be said that if:

* A = B and…
* B = C

then:

* A = C

Duh, right? But this has some nice implications on SQL optimisers.

Let’s look at an example. Let’s get all films for ACTOR\_ID = 1:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT first\_name, last\_name, film\_id  FROM actor a  JOIN film\_actor fa ON a.actor\_id = fa.actor\_id  WHERE a.actor\_id = 1; |

The result being:

FIRST\_NAME LAST\_NAME FILM\_ID

PENELOPE GUINESS 1

PENELOPE GUINESS 23

PENELOPE GUINESS 25

PENELOPE GUINESS 106

PENELOPE GUINESS 140

PENELOPE GUINESS 166

...

Now, observe the execution plan if we run this query in Oracle:

--------------------------------------------------------------

| Id | Operation | Name | Rows |

--------------------------------------------------------------

| 0 | SELECT STATEMENT | | |

| 1 | NESTED LOOPS | | 19 |

| 2 | TABLE ACCESS BY INDEX ROWID| ACTOR | 1 |

|\* 3 | INDEX UNIQUE SCAN | PK\_ACTOR | 1 |

|\* 4 | INDEX RANGE SCAN | PK\_FILM\_ACTOR | **19** |

--------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

**3 - access("A"."ACTOR\_ID"=1)**

**4 - access("FA"."ACTOR\_ID"=1)**

Specifically the predicate section is really interesting. The predicate ACTOR\_ID = 1 is applied to both the ACTOR and FILM\_ACTOR tables because of transitive closure. If:

* A.ACTOR\_ID = 1 (from the WHERE predicate) and…
* A.ACTOR\_ID = FA.ACTOR\_ID (from the ON predicate)

Then:

* FA.ACTOR\_ID = 1

In other words, the query is rewritten to this:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT first\_name, last\_name, film\_id  FROM actor a  JOIN film\_actor fa ON a.actor\_id = fa.actor\_id  WHERE a.actor\_id = 1  AND fa.actor\_id = 1; |

Or in this particular case, even this, as the A.ACTOR\_ID = 1 predicate ensures a single column from the actor table, so a cross join might do as well (at least that’s what the plan indicates):

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT first\_name, last\_name, film\_id  FROM actor a  JOIN film\_actor fa ON fa.actor\_id = 1  WHERE a.actor\_id = 1; |

This has a few nice effects on more complex queries. In particular, the cardinality estimates will be much more precise this way, as we can pick the estimate based on a concrete, constant predicate value, rather than e.g. the average number of films per actor as in this query (which returns the same result):

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT first\_name, last\_name, film\_id  FROM actor a  JOIN film\_actor fa ON a.actor\_id = fa.actor\_id  WHERE first\_name = 'PENELOPE'  AND last\_name = 'GUINESS' |

The plan being:

----------------------------------------------------------------------------

| Id | Operation | Name | Rows |

----------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | |

| 1 | NESTED LOOPS | | **2** |

|\* 2 | TABLE ACCESS BY INDEX ROWID BATCHED| ACTOR | 1 |

|\* 3 | INDEX RANGE SCAN | IDX\_ACTOR\_LAST\_NAME | 3 |

|\* 4 | INDEX RANGE SCAN | PK\_FILM\_ACTOR | **27** |

----------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - filter("A"."FIRST\_NAME"='PENELOPE')

3 - access("A"."LAST\_NAME"='GUINESS')

4 - access("A"."ACTOR\_ID"="FA"."ACTOR\_ID")

As you can see, the estimate for the number of FILM\_ACTOR rows is too high, and the estimate for the NESTED LOOP result is too low. Here are some interesting numbers:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT count(\*) FROM film\_actor WHERE actor\_id = 1;    SELECT avg(c) FROM (    SELECT count(\*) c FROM film\_actor GROUP BY actor\_id  ); |

Resulting in:

19

27.315

That’s where those estimates come from. If the database *knows* we’re dealing with ACTOR\_ID = 1, it can pick the statistics on the number of films for *that actor*. If it *doesn’t know this* (because our standard statistics don’t correlate FIRST\_NAME / LAST\_NAME with ACTOR\_ID), then we get the average number of films for *any actor*. Simple, insignificant error in this particular case, but when that error propagates in a complex query, it can add up and lead to the wrong choice of JOIN down the line (or up the plan).

So, when you can, always design JOIN and ordinary predicates to profit from transitive closure.

What other databases support this?

**DB2**

Yes

Explain Plan

-----------------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 13

2 | NLJOIN | 27 of 1 | 13

3 | FETCH ACTOR | 1 of 1 (100.00%) | 6

4 | IXSCAN PK\_ACTOR | 1 of 200 ( .50%) | 0

5 | IXSCAN PK\_FILM\_ACTOR | 27 of 5462 ( .49%) | 6

Predicate Information

**4 - START (Q2.ACTOR\_ID = 1)**

**STOP (Q2.ACTOR\_ID = 1)**

**5 - START (1 = Q1.ACTOR\_ID)**

**STOP (1 = Q1.ACTOR\_ID)**

Btw, want cool execution plans like the above on DB2 LUW? Go visit Markus Winand’s script:  
<http://use-the-index-luke.com/s/last_explained>

**MySQL**

Unfortunately, MySQL explain plans are not very useful for such analyses. We don’t really *see* the predicate itself in this output:

ID SELECT TYPE TABLE TYPE REF ROWS

------------------------------------------

1 SIMPLE a const **const** 1

1 SIMPLE fa ref **const** 19

But the fact that the REF column is two times “const” indicates that we’re scanning for a constant value in both tables. Conversely, the plan that queries for FIRST\_NAME / LAST\_NAME looks like this:

ID SELECT TYPE TABLE TYPE REF ROWS

-----------------------------------------------

1 SIMPLE a ref const 3

1 SIMPLE fa ref a.actor\_id 27

And you can see that REF has now switched to a column reference from the JOIN predicate. The cardinality estimate is now almost the same as in Oracle.

So, yes, MySQL supports transitive closure, too.

**PostgreSQL**

Yes

QUERY PLAN

------------------------------------------------------------------------------------

Nested Loop (cost=4.49..40.24 rows=27 width=15)

-> Seq Scan on actor a (cost=0.00..4.50 rows=1 width=17)

**Filter: (actor\_id = 1)**

-> Bitmap Heap Scan on film\_actor fa (cost=4.49..35.47 rows=27 width=4)

Recheck Cond: (actor\_id = 1)

-> Bitmap Index Scan on film\_actor\_pkey (cost=0.00..4.48 rows=27 width=0)

**Index Cond: (actor\_id = 1)**

**SQL Server**

Yes

|--Nested Loops(Inner Join)

|--Nested Loops(Inner Join)

| |--Index Seek **(SEEK:([a].[actor\_id]=(1)))**

| |--RID Lookup

|--Index Seek **(SEEK:([fa].[actor\_id]=(1)))**

**Summary**

All databases can do transitive closure:

|  |  |
| --- | --- |
| **Database** | **Transitive closure** |
| DB2 LUW 10.5 | Yep |
| MySQL 8.0.2 | Yep |
| Oracle 12.2.0.1 | Yep |
| PostgreSQL 9.6 | Yep |
| SQL Server 2014 | Yep |

[Stay tuned, though, for #6](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top6). There are more complex cases of transitive closure, where not all databases get it right.

[**2. Impossible Predicates and Unneeded Table Accesses**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top2)

This optimisation is really silly, but hey, why not. If users write impossible predicates, then why even execute them? Here are some examples:

|  |  |
| --- | --- |
| 1  2  3  4  5 | -- "Obvious"  SELECT \* FROM actor WHERE 1 = 0    -- "Subtle"  SELECT \* FROM actor WHERE NULL = NULL |

The first query should obviously never return any results, but the same is true for the second one, because while NULL IS NULL yields TRUE, always, NULL = NULL evaluates to NULL, which has the same effect as FALSE according to [three-valued logic](https://en.wikipedia.org/wiki/Three-valued_logic).

This doesn’t need much explanation, so let’s immediately jump to see which databases optimise this:

**DB2**

Yes

Explain Plan

-----------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 0

2 | **TBSCAN GENROW** | 0 of 0 | 0

As you can see, the table access to the ACTOR table is completely eliminated from the plan. There’s only a GENROW operation, which generates zero rows. Perfect.

**MySQL**

Yes

ID SELECT TYPE TABLE EXTRAS

-----------------------------------------

1 SIMPLE **Impossible WHERE**

This time, MySQL has been so kind to indicate that the WHERE clause is impossible. Thanks, that’s helpful when analysing – more than the other databases

**Oracle**

Yes

---------------------------------------------------------------

| Id | Operation | Name | Starts | E-Rows | A-Rows |

---------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | | 0 |

|\* 1 | **FILTER** | | 1 | | 0 |

| 2 | TABLE ACCESS FULL| ACTOR | 0 | 200 | 0 |

---------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

**1 - filter(NULL IS NOT NULL)**

Now, observe that the plan still shows the table access to the ACTOR table, and the estimated number of rows is still 200, but there’s a FILTER operation on Id=1, which can never be true. Because Oracle really really doesn’t like the [SQL standard BOOLEAN type](https://community.oracle.com/ideas/2633), they display NULL IS NOT NULL in the plan, rather than simply FALSE. Oh well :)

But seriously, do check for this predicate. I’ve been debugging 1000-line-long execution plan subtrees with super high costs before noticing that the entire subtree was “cut off” by NULL IS NOT NULL. A bit misleading, if you ask me.

**PostgreSQL**

Yes

QUERY PLAN

-------------------------------------------

Result (cost=0.00..0.00 rows=0 width=228)

**One-Time Filter: false**

That’s nicer. No noisy ACTOR access and a nice little FALSE predicate.

**SQL Server**

Yes

|--Constant Scan

SQL Server calls this a “constant scan”, i.e. a scan where nothing happens – just like DB2.

All databases can eliminate impossible predicates:

|  |  |  |
| --- | --- | --- |
| **Database** | **Impossible predicates** | **Unneeded table access** |
| DB2 LUW 10.5 | Yep | Yep |
| MySQL 8.0.2 | Yep | Yep |
| Oracle 12.2.0.1 | Yep | Yep |
| PostgreSQL 9.6 | Yep | Yep |
| SQL Server 2014 | Yep | Yep |

[**3. JOIN Elimination**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top3)

In the previous section, we’ve seen unneeded table access for single table queries. But what happens if one out of several table accesses is unneeded in a JOIN?

[I’ve already blogged about JOIN elimination in a previous blog post](https://blog.jooq.org/2017/09/01/join-elimination-an-essential-optimiser-feature-for-advanced-sql-usage/). SQL engines can determine, based on the way a query is written, and based on the presence of PRIMARY KEYs and FOREIGN KEYs, whether any given JOIN is really required in a query, or whether it could be eliminated without affecting the semantics of the query.

In all of the following three examples, the JOIN is unnecessary:

**to-one INNER JOINs can be removed if there’s a NOT NULL FOREIGN KEY**

Instead of this:

|  |  |
| --- | --- |
| 1  2  3 | SELECT first\_name, last\_name  FROM customer c  JOIN address a ON c.address\_id = a.address\_id |

The database can run this:

|  |  |
| --- | --- |
| 1  2 | SELECT first\_name, last\_name  FROM customer c |

**to-one INNER JOINs can be replaced if there’s a nullable FOREIGN KEY**

The above works if there’s also a NOT NULL constraint on the FOREIGN KEY. If there isn’t, e.g. as in this query:

|  |  |
| --- | --- |
| 1  2  3 | SELECT title  FROM film f  JOIN language l ON f.original\_language\_id = l.language\_id |

The JOIN can still be eliminated, but there needs to be a replacement NOT NULL predicate, as such:

|  |  |
| --- | --- |
| 1  2  3 | SELECT title  FROM film  WHERE original\_language\_id IS NOT NULL |

**to-one OUTER JOINs can be removed if there’s a UNIQUE KEY**

Instead of this:

|  |  |
| --- | --- |
| 1  2  3 | SELECT first\_name, last\_name  FROM customer c  LEFT JOIN address a ON c.address\_id = a.address\_id |

The database can again run this:

|  |  |
| --- | --- |
| 1  2 | SELECT first\_name, last\_name  FROM customer c |

… even if there is no FOREIGN KEY on CUSTOMER.ADDRESS\_ID.

**to-many DISTINCT OUTER JOINs can be removed**

Instead of this:

|  |  |
| --- | --- |
| 1  2  3 | SELECT DISTINCT first\_name, last\_name  FROM actor a  LEFT JOIN film\_actor fa ON a.actor\_id = fa.actor\_id |

The database can run this:

|  |  |
| --- | --- |
| 1  2 | SELECT DISTINCT first\_name, last\_name  FROM actor a |

[All of these examples are explained in detail in the previous article](https://blog.jooq.org/2017/09/01/join-elimination-an-essential-optimiser-feature-for-advanced-sql-usage/), so I’m not going to repeat it, but here’s again the summary of what each database can eliminate:

Here’s a summary of what databases can eliminate:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Database** | **INNER JOIN: to-one** | **INNER JOIN nullable: to-one** | **OUTER JOIN: to-one** | **OUTER JOIN DISTINCT: to-many** |
| DB2 LUW 10.5 | Yep | Yep | Yep | Yep |
| MySQL 8.0.2 | Nope | Nope | Nope | Nope |
| Oracle 12.2.0.1 | Yep | Yep | Yep | Nope |
| PostgreSQL 9.6 | Nope | Nope | Yep | Nope |
| SQL Server 2014 | Yep | Nope | Yep | Yep |

Unfortunately, not all databases can eliminate all joins. DB2 and SQL Server are the clear winners here!

[**4. Removing “Silly” Predicates**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top4)

Equally silly are predicates that are (almost) always true. As you can imagine, if you search for

|  |  |
| --- | --- |
| 1 | SELECT \* FROM actor WHERE 1 = 1; |

Then, databases will not actually evaluate the predicate but ignore it. [This was a recent Stack Overflow question that I’ve answered](https://stackoverflow.com/q/46366684/521799), which actually gave me the idea to write this blog post.

I’ll leave it to you to check this, but what happens if the predicate is just slightly less silly, e.g.:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM film WHERE release\_year = release\_year; |

Do we actually have to compare the value with itself on each row? No, there’s no value where this can be FALSE, right? Right. But we still have to do a check. While the predicate can never be FALSE, it can totally be NULL, [again because of three valued logic](https://en.wikipedia.org/wiki/Three-valued_logic). The RELEASE\_YEAR column is a nullable column, and if RELEASE\_YEAR IS NULL for any given row, then NULL = NULL yields NULL, and the row must be excluded.

So, the query is transformed into this:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM film WHERE release\_year IS NOT NULL; |

Which databases do this?

**DB2**

Yes

Explain Plan

-------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 49

2 | TBSCAN FILM | 1000 of 1000 (100.00%) | 49

Predicate Information

**2 - SARG Q1.RELEASE\_YEAR IS NOT NULL**

**MySQL**

Very regrettably, again, because MySQL doesn’t display predicates in their execution plans, it’s a bit hard to find out whether MySQL performs this particular optimisation. [We could benchmark things and see if some really big string comparisons are executed or not](https://www.jooq.org/benchmark). Or, we add an index:

|  |  |
| --- | --- |
| 1 | CREATE INDEX i\_release\_year ON film (release\_year); |

And get the plans for these queries instead:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM film WHERE release\_year = release\_year;  SELECT \* FROM film WHERE release\_year IS NOT NULL; |

If the optimisation works, then both queries should produce exactly the same plan. But they don’t in this case:

ID TABLE POSSIBLE\_KEYS ROWS FILTERED EXTRA

------------------------------------------------------

1 film 1000 10.00 Using where

ID TABLE POSSIBLE\_KEYS ROWS FILTERED EXTRA

------------------------------------------------------

1 film i\_release\_year 1000 100.00 Using where

As you can see, the two queries differ substantially in that the POSSIBLE\_KEYS and FILTERED columns yield different values. I’m making an educated guess and say MySQL does not optimise this.

**Oracle**

Yes

----------------------------------------------------

| Id | Operation | Name | Starts | E-Rows |

----------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | |

|\* 1 | TABLE ACCESS FULL| FILM | 1 | 1000 |

----------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

**1 - filter("RELEASE\_YEAR" IS NOT NULL)**

**PostgreSQL**

Disappointingly, no!

QUERY PLAN

--------------------------------------------------------------

Seq Scan on film (cost=0.00..67.50 **rows=5** width=386)

Filter: ((release\_year)::integer = (release\_year)::integer)

The plans and the costs are different. Specifically, observe the cardinality estimate, which is totally off, when this predicate:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM film WHERE release\_year IS NOT NULL; |

… yields much better results

QUERY PLAN

---------------------------------------------------------

Seq Scan on film (cost=0.00..65.00 **rows=1000** width=386)

Filter: (release\_year IS NOT NULL)

Bummer!

**SQL Server**

Surprisingly, also SQL Server doesn’t seem to do this:

|--Table Scan(OBJECT:([film]), WHERE:([release\_year]=[release\_year]))

However, the cardinality estimate is correct when looking at the visual plan, and the costs are all correct as well. From what I’ve seen in the past on SQL Server, though, I’m going to say that in this case, the optimisation is not taking place because SQL server would display the actually executed predicate in the plan (look at the CHECK constraint examples below to see why).

**What about “silly” predicates on NOT NULL columns?**

The above transformation was only needed, because RELEASE\_YEAR is a nullable column. What if we did the same silly query with e.g. FILM\_ID?

|  |  |
| --- | --- |
| 1 | SELECT \* FROM film WHERE film\_id = film\_id |

This is now the same as not putting a predicate at all. Or at least it should be. Is it, though?

**DB2**

Yes!

Explain Plan

-------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 49

2 | TBSCAN FILM | 1000 of 1000 (100.00%) | 49

No predicate is applied at all, and we’re selecting all the films.

**MySQL**

Yes (educated guess, again)

ID TABLE POSSIBLE\_KEYS ROWS FILTERED EXTRA

------------------------------------------------------

1 film 1000 100.00

Observe how now the EXTRA column is empty as if we didn’t have any WHERE clause!

**Oracle**

Yes

----------------------------------------------------

| Id | Operation | Name | Starts | E-Rows |

----------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | |

| 1 | TABLE ACCESS FULL| FILM | 1 | 1000 |

----------------------------------------------------

Again, no predicates are applied.

**PostgreSQL**

Gee, still no!

QUERY PLAN

------------------------------------------------------

Seq Scan on film (cost=0.00..67.50 rows=5 width=386)

Filter: (film\_id = film\_id)

The filter is applied and the cardinality estimate is still 5. Bummer!

**SQL Server**

Also, still no!

|--Table Scan(OBJECT:([film]), WHERE:([film\_id]=[film\_id]))

**Summary**

This appears like a simple optimisation, but it is not applied in all databases, surprisingly not in SQL Server!

|  |  |  |
| --- | --- | --- |
| **Database** | **Silly but needed predicates (NULL semantics)** | **Silly unneeded predicates (no NULL semantics)** |
| DB2 LUW 10.5 | Yep | Yep |
| MySQL 8.0.2 | Nope | Yep |
| Oracle 12.2.0.1 | Yep | Yep |
| PostgreSQL 9.6 | Nope | Nope |
| SQL Server 2014 | Nope | Nope |

[**5. Projections in EXISTS Subqueries**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top5)

Interestingly, this one, I get asked all the time in my [SQL Masterclass](https://www.jooq.org/training) where I advocate that [SELECT \* is mostly bad](https://blog.jooq.org/2017/03/08/many-sql-performance-problems-stem-from-unnecessary-mandatory-work/).

The question then is, is it OK to use SELECT \* in an EXISTS subquery? For instance, if we wanted to find actors who have played in films:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT first\_name, last\_name  FROM actor a  WHERE EXISTS (    SELECT \* -- Is this OK?    FROM film\_actor fa    WHERE a.actor\_id = fa.actor\_id  ) |

And the answer is: Yes it is OK. The asterisk has no impact on the query. How can we “prove” this? Consider the following query:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | -- DB2  SELECT 1 / 0 FROM sysibm.dual    -- Oracle  SELECT 1 / 0 FROM dual    -- PostgreSQL, SQL Server  SELECT 1 / 0    -- MySQL  SELECT pow(-1, 0.5); |

All databases report a division by zero error. Note that interestingly, in MySQL, dividing by zero yields NULL, not an error, so we’re doing something else that’s illegal.

Now, what happens if we do this, instead?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | -- DB2  SELECT CASE WHEN EXISTS (    SELECT 1 / 0 FROM sysibm.dual  ) THEN 1 ELSE 0 END  FROM sysibm.dual    -- Oracle  SELECT CASE WHEN EXISTS (    SELECT 1 / 0 FROM dual  ) THEN 1 ELSE 0 END  FROM dual    -- PostgreSQL  SELECT EXISTS (SELECT 1 / 0)    -- SQL Server  SELECT CASE WHEN EXISTS (    SELECT 1 / 0  ) THEN 1 ELSE 0 END    -- MySQL  SELECT EXISTS (SELECT pow(-1, 0.5)); |

Now, none of the databases fail the query. All of them return TRUE or 1. This means that none of the databases actually evaluated the *projection* (i.e. the SELECT clause) of the EXISTS subquery.

SQL Server, for instance, shows the following plan:

|--Constant Scan(VALUES:((CASE WHEN (1) THEN (1) ELSE (0) END)))

As you can see, the CASE expression was transformed to a constant, the subquery has been eliminated. Other databases still have the subquery in their plan and don’t mention anything about a projection, so let’s again look at the original query’s plan in Oracle:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT first\_name, last\_name  FROM actor a  WHERE EXISTS (    SELECT \*    FROM film\_actor fa    WHERE a.actor\_id = fa.actor\_id  ) |

The plan for the above is:

------------------------------------------------------------------

| Id | Operation | Name | E-Rows |

------------------------------------------------------------------

| 0 | SELECT STATEMENT | | |

|\* 1 | HASH JOIN SEMI | | 200 |

| 2 | TABLE ACCESS FULL | ACTOR | 200 |

**| 3 | INDEX FAST FULL SCAN| IDX\_FK\_FILM\_ACTOR\_ACTOR | 5462 |**

------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - access("A"."ACTOR\_ID"="FA"."ACTOR\_ID")

Column Projection Information (identified by operation id):

-----------------------------------------------------------

1 - (#keys=1) LAST\_NAME, FIRST\_NAME

2 - (rowset=256) A.ACTOR\_ID, FIRST\_NAME, LAST\_NAME

**3 - FA.ACTOR\_ID**

Observe the projection information on the FILM\_ACTOR access in Id=3. In fact, we’re not even accessing the FILM\_ACTOR table, because we don’t have to. The EXISTS predicate can be executed using the foreign key index on the ACTOR\_ID column only, that’s all we need for this query – despite us having written SELECT \*

**Summary**

Luckily, all databases can remove the projection in EXISTS subqueries

|  |  |
| --- | --- |
| **Database** | **EXISTS projection** |
| DB2 LUW 10.5 | Yep |
| MySQL 8.0.2 | Yep |
| Oracle 12.2.0.1 | Yep |
| PostgreSQL 9.6 | Yep |
| SQL Server 2014 | Yep |

[**6. Predicate Merging**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top6)

This one is interesting and has bitten me in the past when I erroneously assumed that a given database could do it.

Consider the following query:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT \*  FROM actor  WHERE actor\_id IN (2, 3, 4)  AND actor\_id IN (1, 2, 3); |

Obviously, the two predicates overlap and can be merged. I would expect the database to transform the above into:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \*  FROM actor  WHERE actor\_id IN (2, 3); |

Looks obvious, right? It is [a more sophisticated case of transitive closure](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top1). Another case would be merging two ranges. When running the following query:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT \*  FROM film  WHERE film\_id BETWEEN 1 AND 100  AND film\_id BETWEEN 99 AND 200 |

We’d hope for the database to rewrite the query to this:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \*  FROM film  WHERE film\_id BETWEEN 99 AND 100 |

The cardinality of the latter predicate should be 2 rows, but the first, combined ranges might not look like it, and the database might choose a full table scan when it should pick the index

Which database can do these optimisations?

**DB2**

*Merging IN predicates*

Yes

Explain Plan

--------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 11

2 | FETCH ACTOR | 2 of 2 (100.00%) | 11

3 | IXSCAN PK\_ACTOR | 2 of 200 ( 1.00%) | 0

Predicate Information

**3 - SARG Q3.ACTOR\_ID IN (2, 3)**

*Merging range predicates*

Yes (but don’t be fooled by the plan!)

Explain Plan

--------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 13

2 | FETCH FILM | 2 of 2 (100.00%) | 13

**3 | IXSCAN PK\_FILM | 2 of 1000 ( .20%) | 6**

Predicate Information

**3 - START (99 <= Q1.FILM\_ID)**

**STOP (Q1.FILM\_ID <= 100)**

SARG (Q1.FILM\_ID <= 200)

SARG (1 <= Q1.FILM\_ID)

As you can see, the predicate was not optimised away entirely. There’s still a filter (SARG) that checks for the overall upper and lower bounds of the combined range, but the important bits are the START and STOP operations, which indicate fast index access. Besides, the cardinality is also correct.

If you want to be sure, just run this impossible predicate here:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT \*  FROM film  WHERE film\_id BETWEEN 1 AND 2  AND film\_id BETWEEN 199 AND 200; |

To get the correct plan:

Explain Plan

-----------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 0

2 | TBSCAN GENROW | 0 of 0 | 0

Predicate Information

2 - RESID (1 = 0)

**MySQL**

*Merging IN predicates*

Again, unfortunately, MySQL doesn’t display the predicate information very nicely. We get the same plan for both queries:

ID TABLE TYPE KEY ROWS FILTERED EXTRA

------------------------------------------------------

1 actor range PRIMARY 2 100.00 Using where

2x the same cardinalities, 2x “Using where” with no indication what exactly is being done inside of “where”, but given the cardinality, we can assume that the transformation happened correctly. We can look at it differently, let’s try this query:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \* FROM actor  WHERE actor\_id IN (3, 4, 5)  AND actor\_id IN (1, 2, 3); |

Which should be transformed into this one:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM actor  WHERE actor\_id = 3; |

And indeed, it happens:

ID TABLE TYPE KEY ROWS FILTERED EXTRA

------------------------------------------------------

1 actor const PRIMARY 1 100.00

Observe how TYPE=range changed to TYPE=const

So, we can conclude that yes, MySQL implements this optimisation.

*Merging range predicates*

Again, the plan is not helpful at all:

ID TABLE TYPE KEY ROWS FILTERED EXTRA

------------------------------------------------------

1 film range PRIMARY 2 100.00 Using where

But we can again prove that the optimisation is being done by creating an “impossible” predicate as such:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT \*  FROM film  WHERE film\_id BETWEEN 1 AND 2  AND film\_id BETWEEN 199 AND 200 |

In case of which the plan switches to:

ID TABLE EXTRA

-----------------------------------------

1 no matching row in const table

So, again good news for MySQL

**Oracle**

*Merging IN predicates*

Yes

----------------------------------------------------------

| Id | Operation | Name | E-Rows |

----------------------------------------------------------

| 0 | SELECT STATEMENT | | |

| 1 | INLIST ITERATOR | | |

| 2 | TABLE ACCESS BY INDEX ROWID| ACTOR | 2 |

|\* 3 | INDEX UNIQUE SCAN | PK\_ACTOR | 2 |

----------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

**3 - access(("ACTOR\_ID"=2 OR "ACTOR\_ID"=3))**

The predicate being applied only includes the values 2 and 3, so the transformation has worked out correctly.

*Merging range predicates*

Again, yes:

----------------------------------------------------------------

| Id | Operation | Name | E-Rows |

----------------------------------------------------------------

| 0 | SELECT STATEMENT | | |

| 1 | TABLE ACCESS BY INDEX ROWID BATCHED| FILM | 2 |

|\* 2 | INDEX RANGE SCAN | PK\_FILM | 2 |

----------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

**2 - access("FILM\_ID">=99 AND "FILM\_ID"<=100)**

**PostgreSQL**

*Merging IN predicates*

Regrettably, no, this is not optimised!

QUERY PLAN

-----------------------------------------------------------------------------------------------

Seq Scan on actor (cost=0.00..5.50 rows=1 width=25)

Filter: ((actor\_id = ANY ('{2,3,4}'::integer[])) AND (actor\_id = ANY ('{1,2,3}'::integer[])))

Both predicates are still present in the execution plan, and the cardinality estimate is wrong, it should be 2, not 1. If I manually transform the query, I’m getting this plan instead:

QUERY PLAN

-----------------------------------------------------

Seq Scan on actor (cost=0.00..4.50 rows=2 width=25)

Filter: (actor\_id = ANY ('{2,3}'::integer[]))

In particular, we can see the wrong plan if the two predicates do not overlap, in case of which an impossible predicate is formed:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT \*  FROM actor  WHERE actor\_id IN (2, 3, 4)  AND actor\_id IN (7, 8, 9) |

Still, this yields a “wrong” plan:

QUERY PLAN

-----------------------------------------------------------------------------------------------

Seq Scan on actor (cost=0.00..5.50 rows=1 width=25)

Filter: ((actor\_id = ANY ('{2,3,4}'::integer[])) AND (actor\_id = ANY ('{7,8,9}'::integer[])))

Bummer!

*Merging range predicates*

This doesn’t look better

QUERY PLAN

--------------------------------------------------------------------------------------------

Index Scan using film\_pkey on film (cost=0.28..8.30 rows=1 width=386)

Index Cond: ((film\_id >= 1) AND (film\_id <= 100) AND (film\_id >= 99) AND (film\_id <= 200))

Now, it’s hard to say whether this worked or not. Ultimately, we have gotten the correct plan with a reasonable cardinality as before, and it might just work out as on DB2. But what happens if we again create an impossible predicate?

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT \*  FROM film  WHERE film\_id BETWEEN 1 AND 2  AND film\_id BETWEEN 199 AND 200; |

The plan got worse:

QUERY PLAN

-------------------------------------------------------------------------------------------

Index Scan using film\_pkey on film (cost=0.28..8.42 rows=5 width=386)

Index Cond: ((film\_id >= 1) AND (film\_id >= 2) AND (film\_id >= 199) AND (film\_id >= 200))

The cardinality increased instead of it decreasing! And after all, we shouldn’t run this query anyway. No points for PostgreSQL

**SQL Server**

*Merging IN predicates*

Yes, this works:

|--Nested Loops(Inner Join)

|--Index Seek(SEEK:([actor\_id]=(2) OR [actor\_id]=(3)))

|--RID Lookup(OBJECT:([actor]))

*Merging range predicates*

This again looks like the DB2 case:

|--Nested Loops(Inner Join)

|--Index Seek(SEEK:([film\_id] >= (1) AND [film\_id] <= (100)), WHERE:([film\_id]>=(99) AND [film\_id]<=(200)))

|--RID Lookup(OBJECT:([film]))

Unfortunately, observe the distinction between SEEK and WHERE. We want the range [99, 100] in SEEK (as DB2 did) because SEEK is the fast, O(log N) index access, whereas WHERE is linear in O(N) time. Bummer!

This looks like a bug to me, because the impossible predicate yields a more reasonable:

|--Constant Scan

**Summary**

Note that there are many different kinds of predicates that might be merged in one database but not in the other. If in doubt, do check your execution plans!

|  |  |  |
| --- | --- | --- |
| **Database** | **Merging IN** | **Merging ranges** |
| DB2 LUW 10.5 | Yep | Yep |
| MySQL 8.0.2 | Yep | Yep |
| Oracle 12.2.0.1 | Yep | Yep |
| PostgreSQL 9.6 | Nope | Nope |
| SQL Server 2014 | Yep | Nope |

[**7. Provably Empty Sets**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top7)

This one is really cool. We’ve seen [Impossible predicates and unneeded table accesses](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top2) before. What if we do this again, but this time with a JOIN? Can [JOIN elimination](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top3) kick in, too?

We’re trying these queries:

**IS NULL on NOT NULL column**

The predicate in the WHERE clause cannot be TRUE, because we have a NOT NULL constraint on the FILM\_ID column.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT first\_name, last\_name  FROM actor a  JOIN (    SELECT \*    FROM film\_actor    WHERE film\_id IS NULL  ) fa ON a.actor\_id = fa.actor\_id; |

The derived table FA cannot return any rows, because of that NOT NULL constraint on the FA.FILM\_ID column, so it is provably empty. Because an INNER JOIN with an empty table cannot produce any rows either, this should save us from accessing the ACTOR table, so the above query should be rewritten to something like this:

|  |  |
| --- | --- |
| 1  2 | SELECT NULL AS first\_name, NULL AS last\_name  WHERE 1 = 0; |

I.e. the predicate is never evaluated and the JOIN is eliminated.

**INTERSECT NULL and NOT NULL columns**

In principle, this is the same as the previous example, but using a bit more sophisticated syntax:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | SELECT \*  FROM actor a  JOIN (    SELECT actor\_id, film\_id    FROM film\_actor    INTERSECT    SELECT NULL, NULL    FROM dual  ) fa ON a.actor\_id = fa.actor\_id; |

Because of the NOT NULL constraints on both FA.ACTOR\_ID and FA.FILM\_ID, an INTERSECT operation with a (NULL, NULL) tuple should not yield any results, and thus the derived table is provably empty, and thus the INNER JOIN can be eliminated.

Funky, but why not?

**Let’s repeat, with EXISTS**

Finally, let’s repeat the above type of query, but this time with an [SEMI JOIN instead of an INNER JOIN](https://blog.jooq.org/2017/01/12/a-probably-incomplete-comprehensive-guide-to-the-many-different-ways-to-join-tables-in-sql/). First with an [impossible predicate](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top2)…

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SELECT \*  FROM actor a  WHERE a.actor\_id IN (    SELECT actor\_id    FROM film\_actor    WHERE actor\_id IS NULL  ); |

… then again with an intersection.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | SELECT \*  FROM actor a  WHERE a.actor\_id IN (    SELECT actor\_id    FROM film\_actor    INTERSECT    SELECT NULL    FROM sysibm.dual  ) |

Let’s go. Which database can do which optimisation?

**DB2**

*Joining a provably empty set (IS NULL predicate):*

Explain Plan

-----------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 0

2 | TBSCAN GENROW | 0 of 0 | 0

Predicate Information

2 - RESID (1 = 0)

*Joining a provably empty set (INTERSECT):*

Explain Plan

-----------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 0

2 | TBSCAN GENROW | 0 of 0 | 0

Predicate Information

2 - RESID (1 = 0)

*Semi joining a provably empty set (IS NULL predicate):*

Explain Plan

-----------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 0

2 | TBSCAN GENROW | 0 of 0 | 0

Predicate Information

2 - RESID (1 = 0)

*Semi joining a provably empty set (INTERSECT):*

Explain Plan

-----------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 0

2 | TBSCAN GENROW | 0 of 0 | 0

Predicate Information

2 - RESID (1 = 0)

Wow, cool! Looks like a winner!

**MySQL**

*Joining a provably empty set (IS NULL predicate):*

ID TABLE EXTRA

----------------------------

1 Impossible WHERE

Cool! I didn’t expect this!

*Joining a provably empty set (INTERSECT):*

MySQL doesn’t support INTERSECT, regrettably.

*Semi joining a provably empty set (IS NULL predicate):*

ID TABLE EXTRA

----------------------------

1 Impossible WHERE

*Semi joining a provably empty set (INTERSECT):*

MySQL doesn’t support INTERSECT, regrettably.

But still, that’s a great result for MySQL!

**Oracle**

*Joining a provably empty set (IS NULL predicate):*

---------------------------------------------------------------------------

| Id | Operation | Name | Starts | E-Rows | A-Rows |

---------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | | 0 |

**|\* 1 | FILTER | | 1 | | 0 |**

|\* 2 | HASH JOIN | | 0 | 5462 | 0 |

| 3 | TABLE ACCESS FULL | ACTOR | 0 | 200 | 0 |

| 4 | INDEX FAST FULL SCAN| PK\_FILM\_ACTOR | 0 | 5462 | 0 |

---------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

**1 - filter(NULL IS NOT NULL)**

2 - access("A"."ACTOR\_ID"="FILM\_ACTOR"."ACTOR\_ID")

Again, a very confusing execution plan in Oracle, but the NULL IS NOT NULL filter is there, and it happens before all the other operations, which are not executed.

*Joining a provably empty set (INTERSECT):*

---------------------------------------------------------------------------------

| Id | Operation | Name | Starts | E-Rows | A-Rows |

---------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | | 0 |

| 1 | NESTED LOOPS | | 1 | 1 | 0 |

| 2 | NESTED LOOPS | | 1 | 1 | 0 |

| 3 | VIEW | | 1 | 1 | 0 |

| 4 | INTERSECTION | | 1 | | 0 |

| 5 | SORT UNIQUE | | 1 | 5462 | 5463 |

| 6 | INDEX FAST FULL SCAN | PK\_FILM\_ACTOR | 1 | 5462 | 5463 |

| 7 | FAST DUAL | | 1 | 1 | 1 |

|\* 8 | INDEX UNIQUE SCAN | PK\_ACTOR | 0 | 1 | 0 |

| 9 | TABLE ACCESS BY INDEX ROWID| ACTOR | 0 | 1 | 0 |

---------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

8 - access("A"."ACTOR\_ID"="FA"."ACTOR\_ID")

Interesting. This plan will indeed access the entire FILM\_ACTOR primary key. It can save accesses to the ACTOR table and primary key index, because it does the derived table first (which yields no rows), but still those Ids=5 and 6 should not be there. Bummer!

*Semi joining a provably empty set (IS NULL predicate):*

This works again:

-------------------------------------------------------------------------------------

| Id | Operation | Name | Starts | E-Rows | A-Rows |

-------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | | 0 |

|\* 1 | FILTER | | 1 | | 0 |

|\* 2 | HASH JOIN SEMI | | 0 | 200 | 0 |

| 3 | TABLE ACCESS FULL | ACTOR | 0 | 200 | 0 |

| 4 | INDEX FAST FULL SCAN| IDX\_FK\_FILM\_ACTOR\_ACTOR | 0 | 5462 | 0 |

-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter(NULL IS NOT NULL)

2 - access("A"."ACTOR\_ID"="ACTOR\_ID")

… with the same confusing plan that keeps around the unexecuted subtree.

*Semi joining a provably empty set (INTERSECT):*

Again, no optimisation here:

-------------------------------------------------------------------------------------------

| Id | Operation | Name | Starts | E-Rows | A-Rows |

-------------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | | 0 |

| 1 | NESTED LOOPS | | 1 | 1 | 0 |

| 2 | NESTED LOOPS | | 1 | 1 | 0 |

| 3 | VIEW | VW\_NSO\_1 | 1 | 1 | 0 |

| 4 | INTERSECTION | | 1 | | 0 |

| 5 | SORT UNIQUE | | 1 | 5462 | 200 |

| 6 | INDEX FAST FULL SCAN | IDX\_FK\_FILM\_ACTOR\_ACTOR | 1 | 5462 | 5463 |

| 7 | FAST DUAL | | 1 | 1 | 1 |

|\* 8 | INDEX UNIQUE SCAN | PK\_ACTOR | 0 | 1 | 0 |

| 9 | TABLE ACCESS BY INDEX ROWID| ACTOR | 0 | 1 | 0 |

-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

8 - access("A"."ACTOR\_ID"="ACTOR\_ID")

Not so good!

**PostgreSQL**

Disappointingly, PostgreSQL doesn’t fare well in this experiment!

*Joining a provably empty set (IS NULL predicate):*

Nope:

QUERY PLAN

--------------------------------------------------------------------------------------------

Hash Join (cost=8.31..13.07 rows=1 width=13)

Hash Cond: (a.actor\_id = film\_actor.actor\_id)

-> Seq Scan on actor a (cost=0.00..4.00 rows=200 width=17)

-> Hash (cost=8.30..8.30 rows=1 width=2)

-> Index Scan using idx\_fk\_film\_id on film\_actor (cost=0.28..8.30 rows=1 width=2)

Index Cond: (film\_id IS NULL)

*Joining a provably empty set (INTERSECT):*

Even worse:

QUERY PLAN

---------------------------------------------------------------------------------------------------

Hash Join (cost=166.60..171.36 rows=1 width=29)

Hash Cond: (a.actor\_id = fa.actor\_id)

-> Seq Scan on actor a (cost=0.00..4.00 rows=200 width=25)

-> Hash (cost=166.59..166.59 rows=1 width=4)

-> Subquery Scan on fa (cost=0.00..166.59 rows=1 width=4)

-> HashSetOp Intersect (cost=0.00..166.58 rows=1 width=8)

-> Append (cost=0.00..139.26 rows=5463 width=8)

-> Subquery Scan on "\*SELECT\* 2" (cost=0.00..0.02 rows=1 width=8)

-> Result (cost=0.00..0.01 rows=1 width=4)

-> Subquery Scan on "\*SELECT\* 1" (cost=0.00..139.24 rows=5462 width=8)

-> Seq Scan on film\_actor (cost=0.00..84.62 rows=5462 width=4)

*Semi joining a provably empty set (IS NULL predicate):*

Same as inner join:

QUERY PLAN

-------------------------------------------------------------------------------------------------

Hash Semi Join (cost=6.06..10.60 rows=1 width=25)

Hash Cond: (a.actor\_id = film\_actor.actor\_id)

-> Seq Scan on actor a (cost=0.00..4.00 rows=200 width=25)

-> Hash (cost=6.05..6.05 rows=1 width=2)

-> Index Only Scan using film\_actor\_pkey on film\_actor (cost=0.28..6.05 rows=1 width=2)

Index Cond: (actor\_id IS NULL)

*Semi joining a provably empty set (INTERSECT):*

Unsurprisingly:

QUERY PLAN

--------------------------------------------------------------------------------------------------

Hash Semi Join (cost=152.94..157.48 rows=1 width=25)

Hash Cond: (a.actor\_id = "ANY\_subquery".actor\_id)

-> Seq Scan on actor a (cost=0.00..4.00 rows=200 width=25)

-> Hash (cost=152.93..152.93 rows=1 width=2)

-> Subquery Scan on "ANY\_subquery" (cost=0.00..152.93 rows=1 width=2)

-> HashSetOp Intersect (cost=0.00..152.92 rows=1 width=6)

-> Append (cost=0.00..139.26 rows=5463 width=6)

-> Subquery Scan on "\*SELECT\* 2" (cost=0.00..0.02 rows=1 width=6)

-> Result (cost=0.00..0.01 rows=1 width=2)

-> Subquery Scan on "\*SELECT\* 1" (cost=0.00..139.24 rows=5462 width=6)

-> Seq Scan on film\_actor (cost=0.00..84.62 rows=5462 width=2)

**SQL Server**

SQL Server shines, like DB2:

*Joining a provably empty set (IS NULL predicate):*

|--Constant Scan

*Joining a provably empty set (INTERSECT):*

|--Constant Scan

*Semi joining a provably empty set (IS NULL predicate):*

|--Constant Scan

*Semi joining a provably empty set (INTERSECT):*

|--Constant Scan

**Summary**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Database** | **JOIN / NULL** | **JOIN / INTERSECT** | **SEMI JOIN / NULL** | **SEMI JOIN / INTERSECT** |
| DB2 LUW 10.5 | Yep | Yep | Yep | Yep |
| MySQL 8.0.2 | Yep | Not supported | Yep | Not supported |
| Oracle 12.2.0.1 | Yep | Nope | Yep | Nope |
| PostgreSQL 9.6 | Nope | Nope | Nope | Nope |
| SQL Server 2014 | Yep | Yep | Yep | Yep |

On a side note, this could be done in thousands of other ways. Feel free to comment with your own ideas on how to create *“provably empty sets”* to see if this is optimised by any of the databases.

[**8. CHECK Constraints**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top8)

Oh, this is cool! Our Sakila database has a CHECK constraint on the FILM.RATING table:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE film (    ..    RATING varchar(10) DEFAULT 'G',    ..    CONSTRAINT check\_special\_rating      CHECK (rating IN ('G','PG','PG-13','R','NC-17')),    ..  ); |

Seriously, use CHECK constraints for data integrity. The cost of adding them is super low – much less than other constraints like PRIMARY, UNIQUE, and FOREIGN KEY constraints, as the do not profit from an index to enforce them, so you get them almost for “free”.

But there’s also an interesting optimisation aspect here! Check out these queries:

**Impossible predicate**

We’ve seen [impossible predicates](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top2) before, even with NOT NULL constraints (which are special types of CHECK constraints, in fact), but this is even more powerful:

|  |  |
| --- | --- |
| 1  2  3 | SELECT \*  FROM film  WHERE rating = 'N/A'; |

There can be no such film, because the CHECK constraint prevents its insertion (or update). This should again be transformed into a NOOP. Now, what about this?

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE INDEX idx\_film\_rating ON film (rating);    SELECT count(\*)  FROM film  WHERE rating NOT IN ('G','PG','PG-13','R'); |

With the above index, we should probably simply run a quick index scan to count all the films of rating = ‘NC-17’, because that’s the only remaining rating. So the query should be rewritten to this:

|  |  |
| --- | --- |
| 1  2  3 | SELECT count(\*)  FROM film  WHERE rating = 'NC-17'; |

It should be, regardless of the index, because comparing the column with a single value is faster than comparing it with 4 values.

So, which database can do these things?

**DB2**

*Impossible predicate (rating = ‘N/A’)*

Cool!

Explain Plan

-----------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 0

2 | TBSCAN GENROW | 0 of 0 | 0

Predicate Information

2 - RESID (1 = 0)

*Inverse predicate (rating = ‘NC-17’)*

Nope…

Explain Plan

------------------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 34

2 | GRPBY (COMPLETE) | 1 of 210 ( .48%) | 34

**3 | IXSCAN IDX\_FILM\_RATING | 210 of 1000 ( 21.00%) | 34**

Predicate Information

**3 - SARG NOT(Q1.RATING IN ('G', 'PG', 'PG-13', 'R'))**

While the index is used on ID=3 and while the cardinalities are correct, it is scanned entirely, as we do not have a range predicate but a “SARG” predicate. [For more details, see Markus Winand’s overview here](http://use-the-index-luke.com/sql/explain-plan/db2/operations).

We can also show this by manually inverting the predicate to get:

Explain Plan

------------------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 7

2 | GRPBY (COMPLETE) | 1 of 210 ( .48%) | 7

3 | IXSCAN IDX\_FILM\_RATING | 210 of 1000 ( 21.00%) | 7

Predicate Information

**3 - START (Q1.RATING = 'NC-17')**

**STOP (Q1.RATING = 'NC-17')**

Now, we’re getting the desired range predicate

**MySQL**

MySQL supports the CHECK constraint syntax but doesn’t enforce it for whatever reason. Try this:

|  |  |
| --- | --- |
| 1  2  3 | CREATE TABLE x (a INT CHECK (a != 0));  INSERT INTO x VALUES (0);  SELECT \* FROM x; |

You’ll get:

A

-

0

Zero points for MySQL (really, why not just support CHECK constraints?)

**Oracle**

*Impossible predicate (rating = ‘N/A’)*

--------------------------------------------------------------

| Id | Operation | Name | Starts | E-Rows | A-Rows |

--------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | | 0 |

**|\* 1 | FILTER | | 1 | | 0 |**

|\* 2 | TABLE ACCESS FULL| FILM | 0 | 89 | 0 |

--------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

**1 - filter(NULL IS NOT NULL)**

2 - filter("RATING"='N/A')

Again, the super confusing NULL IS NOT NULL filter that cuts off the FULL TABLE SCAN, which might as well be removed entirely from the plan. But at least it works!

*Inverse predicate (rating = ‘NC-17’)*

Ooops:

----------------------------------------------------------------------------

| Id | Operation | Name | Starts | E-Rows | A-Rows |

----------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | | 1 |

| 1 | SORT AGGREGATE | | 1 | 1 | 1 |

|\* 2 | INDEX FAST FULL SCAN| IDX\_FILM\_RATING | 1 | 415 | 210 |

----------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - filter((RATING'PG-13' AND RATING'R' AND RATING'PG' AND RATING'G'))

The predicate could not be inversed, we get a much off cardinality estimate, we get an INDEX FAST FULL SCAN, instead of an INDEX RANGE SCAN, and a filter predicate rather than an access predicate. Here’s what we should have gotten, e.g. when manually inverting the predicate:

------------------------------------------------------------------------

| Id | Operation | Name | Starts | E-Rows | A-Rows |

------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | | 1 |

| 1 | SORT AGGREGATE | | 1 | 1 | 1 |

|\* 2 | INDEX RANGE SCAN| IDX\_FILM\_RATING | 1 | 210 | 210 |

------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("RATING"='NC-17')

Bummer!

**PostgreSQL**

Note that the [Sakila database](https://www.jooq.org/sakila) in its PostgreSQL version uses an ENUM type instead of a CHECK constraint on the RATING column. I’ve duplicated the table to use a CHECK constraint instead.

*Impossible predicate (rating = ‘N/A’)*

Doesn’t work:

QUERY PLAN

------------------------------------------------------

Seq Scan on film2 (cost=0.00..67.50 rows=1 width=385)

Filter: ((rating)::text = 'N/A'::text)

*Inverse predicate (rating = ‘NC-17’)*

Also nope:

QUERY PLAN

------------------------------------------------------------------

Aggregate (cost=70.53..70.54 rows=1 width=8)

-> Seq Scan on film2 (cost=0.00..70.00 rows=210 width=0)

Filter: ((rating)::text ALL ('{G,PG,PG-13,R}'::text[]))

Too bad!

**NOTE:** As was kindly pointed out by [David Rowley in the comments](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151812), this feature can be opted in by specifying:

SET constraint\_exclusion TO on;

**SQL Server**

*Impossible predicate (rating = ‘N/A’)*

Yes!

|--Constant Scan

*Inverse predicate (rating = ‘NC-17’)*

Also yes!

|--Compute Scalar

|--Stream Aggregate

|--Index Seek(OBJECT:([idx\_film\_rating]), SEEK:([rating]='NC-17'))

**Summary**

|  |  |  |
| --- | --- | --- |
| **Database** | **Impossible predicate** | **Inverse predicate** |
| DB2 LUW 10.5 | Yep | Nope |
| MySQL 8.0.2 | Not supported | Not supported |
| Oracle 12.2.0.1 | Yep | Nope |
| PostgreSQL 9.6 | Nope | Nope |
| SQL Server 2014 | Yep | Yep |

[**9. Unneeded Self JOIN**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top9)

When your queries get more complex, it might well happen that you’re going to self JOIN a table based on its primary key. Trust me, this is common practice when you build complex views and JOIN them to each other, so a database noticing this is a crucial step in optimising complex SQL. I won’t show a complex example, but a simple one, e.g.

|  |  |
| --- | --- |
| 1  2  3 | SELECT a1.first\_name, a1.last\_name  FROM actor a1  JOIN actor a2 ON a1.actor\_id = a2.actor\_id; |

This could be considered a special case of [JOIN elimination](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top3) as we don’t really need the JOIN of A2, we can do everything with A1 only. [Now, INNER JOIN elimination normally works in the presence of a FOREIGN KEY only](https://blog.jooq.org/2017/09/01/join-elimination-an-essential-optimiser-feature-for-advanced-sql-usage/), which we don’t have here. But because of the PRIMARY KEY on ACTOR\_ID, we can prove that in fact A1 = A2. In a way, this is [transitive closure all over again](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top1).

We can take this one step further and use columns from both A1 and A2:

|  |  |
| --- | --- |
| 1  2  3 | SELECT a1.first\_name, a2.last\_name  FROM actor a1  JOIN actor a2 ON a1.actor\_id = a2.actor\_id; |

In the classic JOIN elimination case, we could no longer eliminate the JOIN because we’re projecting from both tables. But since we’ve already proven that A1 = A2, we can use them interchangeably, so the expectation is for this query to be transformed into:

|  |  |
| --- | --- |
| 1  2 | SELECT first\_name, last\_name  FROM actor; |

Who can do this?

**DB2**

*Projecting from A1 only*

Yes:

Explain Plan

------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 20

2 | TBSCAN ACTOR | 200 of 200 (100.00%) | 20

*Projecting from A1 and A2*

… and yes:

Explain Plan

------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 20

2 | TBSCAN ACTOR | 200 of 200 (100.00%) | 20

**MySQL**

*Projecting from A1 only*

Nope

ID TABLE REF EXTRA

-----------------------------------

1 a1

1 a2 a1.actor\_id Using index

*Projecting from A1 and A2*

… and nope

ID TABLE REF EXTRA

-----------------------------------

1 a1

1 a2 a1.actor\_id

That’s disappointing…

**Oracle**

*Projecting from A1 only*

Yes

--------------------------------------------

| Id | Operation | Name | E-Rows |

--------------------------------------------

| 0 | SELECT STATEMENT | | |

| 1 | TABLE ACCESS FULL| ACTOR | 200 |

--------------------------------------------

*Projecting from A1 and A2*

And yes

--------------------------------------------

| Id | Operation | Name | E-Rows |

--------------------------------------------

| 0 | SELECT STATEMENT | | |

| 1 | TABLE ACCESS FULL| ACTOR | 200 |

--------------------------------------------

**PostgreSQL**

*Projecting from A1 only*

Nope:

QUERY PLAN

--------------------------------------------------------------------

Hash Join (cost=6.50..13.25 rows=200 width=13)

Hash Cond: (a1.actor\_id = a2.actor\_id)

-> Seq Scan on actor a1 (cost=0.00..4.00 rows=200 width=17)

-> Hash (cost=4.00..4.00 rows=200 width=4)

-> Seq Scan on actor a2 (cost=0.00..4.00 rows=200 width=4)

*Projecting from A1 and A2*

And nope:

QUERY PLAN

---------------------------------------------------------------------

Hash Join (cost=6.50..13.25 rows=200 width=13)

Hash Cond: (a1.actor\_id = a2.actor\_id)

-> Seq Scan on actor a1 (cost=0.00..4.00 rows=200 width=10)

-> Hash (cost=4.00..4.00 rows=200 width=11)

-> Seq Scan on actor a2 (cost=0.00..4.00 rows=200 width=11)

**SQL Server**

*Projecting from A1 only*

Surprisingly, no! (But remember, this is SQL Server 2014, maybe this got fixed in a more recent version. I should definitely upgrade!)

|--Merge Join(Inner Join, MERGE:([a2].[actor\_id])=([a1].[actor\_id]))

|--Index Scan(OBJECT:([a2]))

|--Sort(ORDER BY:([a1].[actor\_id] ASC))

|--Table Scan(OBJECT:([a1]))

*Projecting from A1 and A2*

Also no, and even with a different, worse plan:

|--Hash Match(Inner Join, HASH:([a1].[actor\_id])=([a2].[actor\_id]))

|--Table Scan(OBJECT:([sakila].[dbo].[actor] AS [a1]))

|--Table Scan(OBJECT:([sakila].[dbo].[actor] AS [a2]))

**Summary**

I would have frankly expected this to work on all databases, but I was proven very wrong, which is a shame. Along with [JOIN elimination](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top3), this is one of the most crucial optimisations to enable building huge SQL queries from reusable parts, such as views and table valued functions. Unfortunately, this is not supported 3/5 of the most popular databases.

|  |  |  |
| --- | --- | --- |
| **Database** | **Self-join elimination, single table projection** | **Self-join elimination, complete projection** |
| DB2 LUW 10.5 | Yep | Yep |
| MySQL 8.0.2 | Nope | Nope |
| Oracle 12.2.0.1 | Yep | Yep |
| PostgreSQL 9.6 | Nope | Nope |
| SQL Server 2014 | Nope | Nope |

[**10. Predicate Pushdown**](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#top10)

This optimisation doesn’t belong here 100%, because it is not entirely true to assume this transformation is not cost based. But since I cannot think of a single obvious reason why an optimiser should not push down predicates into derived tables, I’m listing this here along with the other, non-cost-based optimisations.

Consider this query:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT \*  FROM (    SELECT \*    FROM actor  ) a  WHERE a.actor\_id = 1; |

The derived table has absolutely no value in this query and it should be eliminated as well, by unnesting it. But let’s ignore that for a moment.

We’d expect the database to perform this query instead:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT \*  FROM (    SELECT \*    FROM actor    WHERE actor\_id = 1  ) a; |

And then again, possibly, eliminate the outer query.

A more sophisticated example would be when using UNION:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | SELECT \*  FROM (    SELECT first\_name, last\_name, 'actor' type    FROM actor    UNION ALL    SELECT first\_name, last\_name, 'customer' type    FROM customer  ) people  WHERE people.last\_name = 'DAVIS'; |

The result of this query is:

FIRST\_NAME LAST\_NAME TYPE

----------------------------

JENNIFER DAVIS actor

SUSAN DAVIS actor

SUSAN DAVIS actor

JENNIFER DAVIS customer

Now, we’d love the database optimiser to run this statement instead:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | SELECT \*  FROM (    SELECT first\_name, last\_name, 'actor' type    FROM actor    WHERE last\_name = 'DAVIS'    UNION ALL    SELECT first\_name, last\_name, 'customer' type    FROM customer    WHERE last\_name = 'DAVIS'  ) people; |

I.e. pushing down the predicate into the derived table, and from there on into the two UNION ALL subqueries, because after all, we have indexes on both ACTOR.LAST\_NAME and CUSTOMER.LAST\_NAME columns.

Again, this transformation might be motivated based on costs in most databases, but I still think it’s a no-brainer to do anyway, because it’s almost always better to reduce the number of processed tuples as early as possible in any algorithm. If you know a case where this transformation is a bad idea, please comment! I’d be very curious.

So, which databases can do this? (And please, this is so basic, yet important, let the answer be: all)

**DB2**

*Simple derived table*

Yes

Explain Plan

--------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 6

2 | FETCH ACTOR | 1 of 1 (100.00%) | 6

3 | IXSCAN PK\_ACTOR | 1 of 200 ( .50%) | 0

Predicate Information

3 - START (Q1.ACTOR\_ID = 1)

STOP (Q1.ACTOR\_ID = 1)

*UNION derived table*

Yes, again:

Explain Plan

-----------------------------------------------------------------

ID | Operation | Rows | Cost

1 | RETURN | | 20

2 | UNION | 2 of 1 | 20

3 | FETCH CUSTOMER | 1 of 1 (100.00%) | 13

4 | IXSCAN IDX\_CUSTOMER\_LAST\_NAME | 1 of 599 ( .17%) | 6

5 | FETCH ACTOR | 1 of 1 (100.00%) | 6

6 | IXSCAN IDX\_ACTOR\_LAST\_NAME | 1 of 200 ( .50%) | 0

Predicate Information

4 - START (Q1.LAST\_NAME = 'DAVIS')

STOP (Q1.LAST\_NAME = 'DAVIS')

6 - START (Q3.LAST\_NAME = 'DAVIS')

STOP (Q3.LAST\_NAME = 'DAVIS')

Also, in both cases, the derived table (view) was removed from the plan as it is not really necessary.

**MySQL**

*Simple derived table*

Yes

ID TABLE TYPE KEY REF EXTRA

---------------------------------------

1 actor const PRIMARY const

The usual PRIMARY KEY access by a constant value is applied.

*UNION derived table*

Oops, nope

ID SELECT\_TYPE TABLE TYPE KEY REF ROWS EXTRA

------------------------------------------------------------------

1 PRIMARY ref const 10

2 DERIVED actor ALL 200

3 UNION customer ALL 599

The manual transformation would yield:

ID SELECT\_TYPE TABLE TYPE KEY REF ROWS EXTRA

--------------------------------------------------------------------------

1 PRIMARY ALL 5

2 DERIVED actor ref idx\_actor\_last\_name const 3

3 UNION customer ref idx\_last\_name const 1

That’s really a problem if you want to nest complex queries in MySQL!

**Oracle**

*Simple derived table*

Yes, works

---------------------------------------------------------------------------

| Id | Operation | Name | Starts | E-Rows | A-Rows |

---------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 1 | | 1 |

| 1 | TABLE ACCESS BY INDEX ROWID| ACTOR | 1 | 1 | 1 |

|\* 2 | INDEX UNIQUE SCAN | PK\_ACTOR | 1 | 1 | 1 |

---------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

2 - access("ACTOR"."ACTOR\_ID"=1)

The derived table has been unnested, too.

*UNION derived table*

Works as well:

---------------------------------------------------------------------------------

| Id | Operation | Name | E-Rows |

---------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | |

| 1 | VIEW | | 4 |

| 2 | UNION-ALL | | |

| 3 | TABLE ACCESS BY INDEX ROWID BATCHED| ACTOR | 3 |

|\* 4 | INDEX RANGE SCAN | IDX\_ACTOR\_LAST\_NAME | 3 |

| 5 | TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMER | 1 |

|\* 6 | INDEX RANGE SCAN | IDX\_CUSTOMER\_LAST\_NAME | 1 |

---------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

4 - access("LAST\_NAME"='DAVIS')

6 - access("LAST\_NAME"='DAVIS')

However, without unnesting the derived table. The Id=1 “VIEW” indicates that it’s still there. This isn’t a problem in this case, just perhaps a bit cosmetic overhead.

**PostgreSQL**

*Simple derived table*

Yes, it works:

QUERY PLAN

----------------------------------------------------

Seq Scan on actor (cost=0.00..4.50 rows=1 width=25)

Filter: (actor\_id = 1)

Note, interestingly, PostgreSQL sometimes doesn’t even use the PRIMARY KEY for a single row lookup but scans the entire table. In this case, 200 rows x 25 bytes per row (“width”) fits in a single block, so why bother reading the index anyway, generating more I/O for this small table access?

*UNION derived table*

Yes, this works as well:

QUERY PLAN

-----------------------------------------------------------------------------------

Append (cost=0.00..12.83 rows=4 width=45)

-> Seq Scan on actor (cost=0.00..4.50 rows=3 width=45)

Filter: ((last\_name)::text = 'DAVIS'::text)

-> Index Scan using idx\_last\_name on customer (cost=0.28..8.29 rows=1 width=45)

Index Cond: ((last\_name)::text = 'DAVIS'::text)

Again, the index on ACTOR.LAST\_NAME is not used, but the one on CUSTOMER.LAST\_NAME is, as the CUSTOMER table is quite larger.

**SQL Server**

*Simple derived table*

Yep, works

|--Nested Loops(Inner Join)

|--Index Seek(SEEK:([actor\_id]=(1)))

|--RID Lookup(OBJECT:([actor]))

*UNION derived table*

Works as well.

|--Concatenation

|--Compute Scalar(DEFINE:([Expr1003]='actor'))

| |--Nested Loops(Inner Join)

| |--Index Seek(SEEK:([actor].[last\_name]='DAVIS'))

| |--RID Lookup(OBJECT:([actor]))

|--Compute Scalar(DEFINE:([Expr1007]='customer'))

|--Nested Loops(Inner Join)

|--Index Seek(SEEK:([customer].[last\_name]='DAVIS'))

|--RID Lookup(OBJECT:([customer]))

**Summary**

My hopes were scattered. MySQL 8.0.2 doesn’t support this simple optimisation completely yet. All others do, however:

|  |  |  |
| --- | --- | --- |
| **Database** | **Simple derived table pushdown** | **UNION derived table pushdown** |
| DB2 LUW 10.5 | Yep | Yep |
| MySQL 8.0.2 | Yep | Nope |
| Oracle 12.2.0.1 | Yep | Yep |
| PostgreSQL 9.6 | Yep | Yep |
| SQL Server 2014 | Yep | Yep |

**Conclusion**

The list presented here is far from complete. There are many more of these simple SQL transformations that are (or should be) a no-brainer for a database to implement, even before the cost-based optimiser kicks in. They remove what I call **unnecessary, *optional* work** ([as opposed to **unnecessary, *mandatory* work**](https://blog.jooq.org/2017/03/08/many-sql-performance-problems-stem-from-unnecessary-mandatory-work/)). They are essential tools for:

* Preventing silly mistakes from affecting SQL performance. Everyone makes mistakes, and as projects grow larger and SQL queries grow more complex, these mistakes might accumulate, yet hopefully, without effect
* Enabling the reuse of complex building blocks, such as views and table-valued functions, which can be inlined into parent SQL queries, transformed, and parts removed or rewritten

These features are essential for the second part. Without them, it is very difficult to build 4000 LOC SQL queries that still perform decently, based on a library of reusable SQL components.

Unfortunately for users of PostgreSQL and MySQL, these two popular Open Source databases are still much behind their commercial counterparts DB2, Oracle, and SQL Server – where DB2 fared best in this article, Oracle and SQL Server being roughly on par.

SQL is a wonderful language, because it is declarative and any statement can be rewritten to something simpler or more sophisticated, which performs much better than what the author has written. If you have liked this article, you may also like:

* [How to Write Efficient TOP N Queries in SQL](https://blog.jooq.org/2017/09/22/how-to-write-efficient-top-n-queries-in-sql/)
* [JOIN Elimination: An Essential Optimiser Feature for Advanced SQL Usage](https://blog.jooq.org/2017/09/01/join-elimination-an-essential-optimiser-feature-for-advanced-sql-usage/)
* [10 SQL Tricks That You Didn’t Think Were Possible](https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/)
* [10 Common Mistakes Java Developers Make when Writing SQL](https://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](https://blog.jooq.org/2013/08/12/10-more-common-mistakes-java-developers-make-when-writing-sql/)
* [Yet Another 10 Common Mistakes Java Developers Make When Writing SQL](https://blog.jooq.org/2014/05/26/yet-another-10-common-mistakes-java-developer-make-when-writing-sql-you-wont-believe-the-last-one/)
* [Many SQL Performance Problems Stem from “Unnecessary, Mandatory Work”](https://blog.jooq.org/2017/03/08/many-sql-performance-problems-stem-from-unnecessary-mandatory-work/)
* [196Click to share on Facebook (Opens in new window)196](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?share=facebook&nb=1)
* [Click to share on LinkedIn (Opens in new window)](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?share=linkedin&nb=1)
* [Click to share on Twitter (Opens in new window)](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?share=twitter&nb=1)
* [Click to share on Reddit (Opens in new window)](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?share=reddit&nb=1)
* [More](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/)
* [Click to email this to a friend (Opens in new window)](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?share=email&nb=1)
* [Click to print (Opens in new window)](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#print)

**Post navigation**

[Previous](https://blog.jooq.org/2017/09/22/how-to-write-efficient-top-n-queries-in-sql/)

[Next](https://blog.jooq.org/2017/10/11/benchmarking-jdk-string-replace-vs-apache-commons-stringutils-replace/)

**25 thoughts on “10 Cool SQL Optimisations That do not Depend on the Cost Model”**

1. Regarding INNER JOIN elimination, isn’t it a requirement that the foreign key be NOT NULL? If the foreign key columns are NULLable then the INNER JOIN works as a restriction and the two queries are not equivalent, aren’t they? I think this should be pointed out.

*Luca Veronese* [, September 28, 2017 at 14:06](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151622)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151622#respond)

* + You’re right, I had this on my TODO list for a future blog post. In fact, the INNER JOIN is transformed into a NOT NULL predicate. Will edit both posts.

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 28, 2017 at 14:22](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151623)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151623#respond)

* + Huh, interesting. In fact, SQL Server 2014 cannot eliminate an INNER JOIN on a nullable FOREIGN KEY. Bummer! Thanks again for your hint!

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 28, 2017 at 14:29](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151624)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151624#respond)

* + - Glad you found it useful

*lveronese65* [, September 28, 2017 at 14:43](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151626)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151626#respond)

* + - SQL Server leaves out some optimizations if a query’s cost is very low. Without the actual execution plan used it’s hard to say, but I suspect the queries here are far too simple to get that optimization. I’ve seen the join elimination occur in much larger queries on a much older version of SQL Server, so I would assume it can happen elsewhere. The available rules are in a somewhat cryptic form in sys.dm\_exec\_query\_transformation\_stats, so one could always try to decipher that to figure out which re-write rules are possible

[*peschkaj*](http://gravatar.com/peschkaj) [, September 30, 2017 at 16:19](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151672)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151672#respond)

* + - * That’s interesting, do you have a link to some documentation / empirical study that backs this claim? I mean, join elimination is a thing that *should* happen before a query is costed, because it can be applied purely based on metadata (constraints) introspection. I would find it surprising that it would apply only after costing the query. In fact, the cases where it works are also very simple.

Having said so, I don’t know SQL Server very well, so you may well be right.

[*lukaseder*](https://lukaseder.wordpress.com/) [, September 30, 2017 at 19:26](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151674)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151674#respond)

* + - * + My best recommendation is to read “Microsoft SQL Server Internals”. The chapter about query optimization is written by the architect who designed the optimizer. But, in short, if plans are “trivial” (only one way to run them), no optimization is performed. You can see this in the actual execution plan properties. If queries are non-trivial but below a certain threshold, a second set of optimizations is considered, and then if queries are above the threshold, all optimizations are included.

This blog post from a former co-worker of mine will show you how to figure out of a plan is trivial <https://www.brentozar.com/archive/2017/06/query-plans-trivial-optimization-vs-simple-parameterization/>

*peschkaj* [, September 30, 2017 at 20:48](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151675)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151675#respond)

Excellent, thanks a lot for the interesting link!

[*lukaseder*](https://lukaseder.wordpress.com/) [, October 2, 2017 at 11:15](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151697)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151697#respond)

1. For #8 PostgreSQL. This is disabled by default and can be enabled with: SET constraint\_exclusion TO on;

[*David Rowley*](http://gravatar.com/davidrowley1) [, October 7, 2017 at 12:21](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151812)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151812#respond)

* + Oh, that’s very interesting! Thanks for sharing – I’ll update the post immediately.

[*lukaseder*](https://lukaseder.wordpress.com/) [, October 9, 2017 at 13:20](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151843)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151843#respond)

1. Hello,  
   Your post was reported to PostgreSQL’s developpers :  
   <https://www.postgresql.org/message-id/CAMjNa7cC4X9YR-vAJS-jSYCajhRDvJQnN7m2sLH1wLh-_Z2bsw@mail.gmail.com>

Which resulted to this first comit :  
<https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=8ec5429e2f422f4d570d4909507db0d4ca83bbac>

I hope to see other optimizations :)

[*Adrien N*](https://twitter.com/Adrien_nayrat) [, October 9, 2017 at 20:57](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151851)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151851#respond)

* + Yeah, I’ve seen the discussion. Very cool. I’m a bit disappointed by Tom Lane’s disapproval of predicate merging (#6) and self join reduction (#9):  
    <https://www.postgresql.org/message-id/7105.1507342794%40sss.pgh.pa.us>

I’ve found these tools to be invaluable for designing very complex SQL queries based on heavily nested views where the same table is joined to itself by design on several levels of nesting… But I guess I should have shown a concrete example. I’ll do so in a future blog post.

[*lukaseder*](https://lukaseder.wordpress.com/) [, October 10, 2017 at 09:33](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151863)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151863#respond)

* + - Hey Luke, I was the one who posted it to the mailing list to start some discussion. I’m not one of the hackers, but I follow the discussion on there. Why not chime in on that thread? Robert Hass just responded with something quite a bit more encouraging.

[*Adam Brusselback*](http://gosimple.me/) [, October 12, 2017 at 00:11](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151900)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151900#respond)

* + - * Yeah, I could. How do I even reply to a specific email on that list?

Indeed, I like Robert Hass’s way of putting it. He’s much more customer oriented, I suspect :) In the end, I don’t really have a stake in this as I’m not going to be using PostgreSQL on client systems any time soon. My clients have requirements that only the top 3 commercial DBs (DB2, SQL Server, or Oracle) can solve right now, and they won’t migrate off those databases. I just want to point out these features / limitations to people using jOOQ to help them make the right choices / tradeoffs in terms of costs / benefits when choosing a particular RDBMS.

[*lukaseder*](https://lukaseder.wordpress.com/) [, October 12, 2017 at 09:59](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-151908)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=151908#respond)

* + - The optimization possibilities is one point – second and pretty important is planner speed – usually more features means less speed. Tom Lane interest is ensuring good enough planner speed and planner complexity.

[*okbobcz*](http://gravatar.com/okbobcz) [, December 8, 2017 at 17:54](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-153650)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=153650#respond)

* + - * I am well aware of this opinion in the PostgreSQL community and among its maintainers. But given the fact that the commercial competitors still drastically outdo PostgreSQL in a variety of performance related areas, I will dare claim that the PostgreSQL folks simply haven’t figured it out yet (without asserting that this is a simple task to do, of course).

[*lukaseder*](https://lukaseder.wordpress.com/) [, December 11, 2017 at 09:18](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-153683)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=153683#respond)

* + - * + I don’t think so every corner case should be soved – For any database server – commercial or free we have to rewrite some queries. PostgreSQL has not implicit plan cache – due less code complexity, and then the planner speed is maybe more important. On the end – the migration issues are not related usually to these missing optimization, but the another details are important – Oracle equality NULL and empty string. PostgreSQL developers has not any benefit from getting bigger business. They searching the compromise between performance and code stability, and maintainability. I agree so some part other commercial databases has better implemented – but usually its is corner cases, and on end – commercial software is significantly much more complex – try to compare speed of postgresql instalation and MSSQL or Oracle – it is seconds versus minutes.

PostgreSQL development continues – but community priority are issue interesting for 90% users. Robert is from EDB – he have to see a marketing issues too and his vision is little bit different than community vision.

Your article is interesting – I don’t remember any similar article in last years – and maybe some optimization can be implemented in PostgreSQL without negative impacts.

*okbobcz* [, December 12, 2017 at 16:44](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-153710)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=153710#respond)

That’s your opinion and experience. It doesn’t match mine at all. While I appreciate PostgreSQL is nicer for developers, Oracle is orders of magnitude better for operations. For the record, operations couldn’t care less about silly quirks like NULL vs empty string or speedy installations (while I do agree they’re tedious for developers). And remember, operations is where the money is, so where do you want to offer most value as a vendor?

In any case, from what I’ve seen on the mailing list, about half of my items are already being addressed by PostgreSQL, so there’s that :-)

[*lukaseder*](https://lukaseder.wordpress.com/) [, December 12, 2017 at 16:59](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-153711)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=153711#respond)

1. Hi,  
   I think it makes sense to also consider. In many db comparisons, MariaDB is not taken int account, which IMHO makes people to still think that MariaDB is more or less the same as MySQL which is not the case.

We’re using Hibernate with joined inheritance, so it makes a big difference to us whether subclasses/tables are always joined in particular queries or not (=3. JOIN Elimination). While checking the execution plans of MySQL 5.5 vs MariaDB 10, I’ve realized that the join elimination capabilities of MariaDB are much better, compared to MySQL. I actually wanted to spend some more time on this topic so see how later MySQL versions perform, but thanks to this post, I don’t have to do that anymore :)).

Best regards,  
Niko

*Niko Wittenbeck* [, October 23, 2017 at 14:45](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-152372)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=152372#respond)

* + Thanks for your comment.

There are many different databases out there that would be worth looking into. Some are really sophisticated for their time series analysis capabilities, for instance. I wish I had more time to look into all of them. In the meantime, I’m mostly focusing on the top 5 databases in this ranking: <https://db-engines.com/en/ranking>

MariaDB is certainly an interesting alternative to MySQL, but I have to draw the line somewhere… In any case, now that you’ve supplied this comment, the comparison is more complete :)

Cheers,  
Lukas

[*lukaseder*](https://lukaseder.wordpress.com/) [, October 23, 2017 at 16:27](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-152387)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=152387#respond)

1. Hi,  
   Thanks for a thorough and interesting blog post! I have published a blog post addressing your complaints about MySQL EXPLAIN not providing enough information to determine what is going on. The blog post shows that this information is available in other variants of EXPLAIN. See  
   <https://oysteing.blogspot.co.uk/2017/11/going-beyond-tabular-explain.html>

[*Øystein Grøvlen*](http://oysteing.blogspot.com/) [, November 24, 2017 at 15:34](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-153370)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=153370#respond)

* + Thanks for your message. Having a lower-level data format (e.g. in XML / JSON), which is processable by tools is excellent, of course. But for human consumption, the tabular format is really the most important one. Compare the utility of MySQL’s explain output with that of Oracle, for instance (which is the best in my opinion).

[*lukaseder*](https://lukaseder.wordpress.com/) [, November 28, 2017 at 11:42](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-153468)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=153468#respond)

1. In the video you mentioned that a stored procedure is a micro service. I tried looking up if other people say that. Could you expound on that comment?

[*jon49*](http://gravatar.com/jon49) [, March 17, 2018 at 02:52](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-159527)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=159527#respond)

* + Gee, err… it was a joke

[*lukaseder*](https://lukaseder.wordpress.com/) [, March 17, 2018 at 17:57](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-159578)

[Reply](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/?replytocom=159578#respond)

* + - Haha, you got me!

[*jon49*](http://thisisafiller.wordpress.com/) [, March 17, 2018 at 19:37](https://blog.jooq.org/2017/09/28/10-cool-sql-optimisations-that-do-not-depend-on-the-cost-model/#comment-159606)