

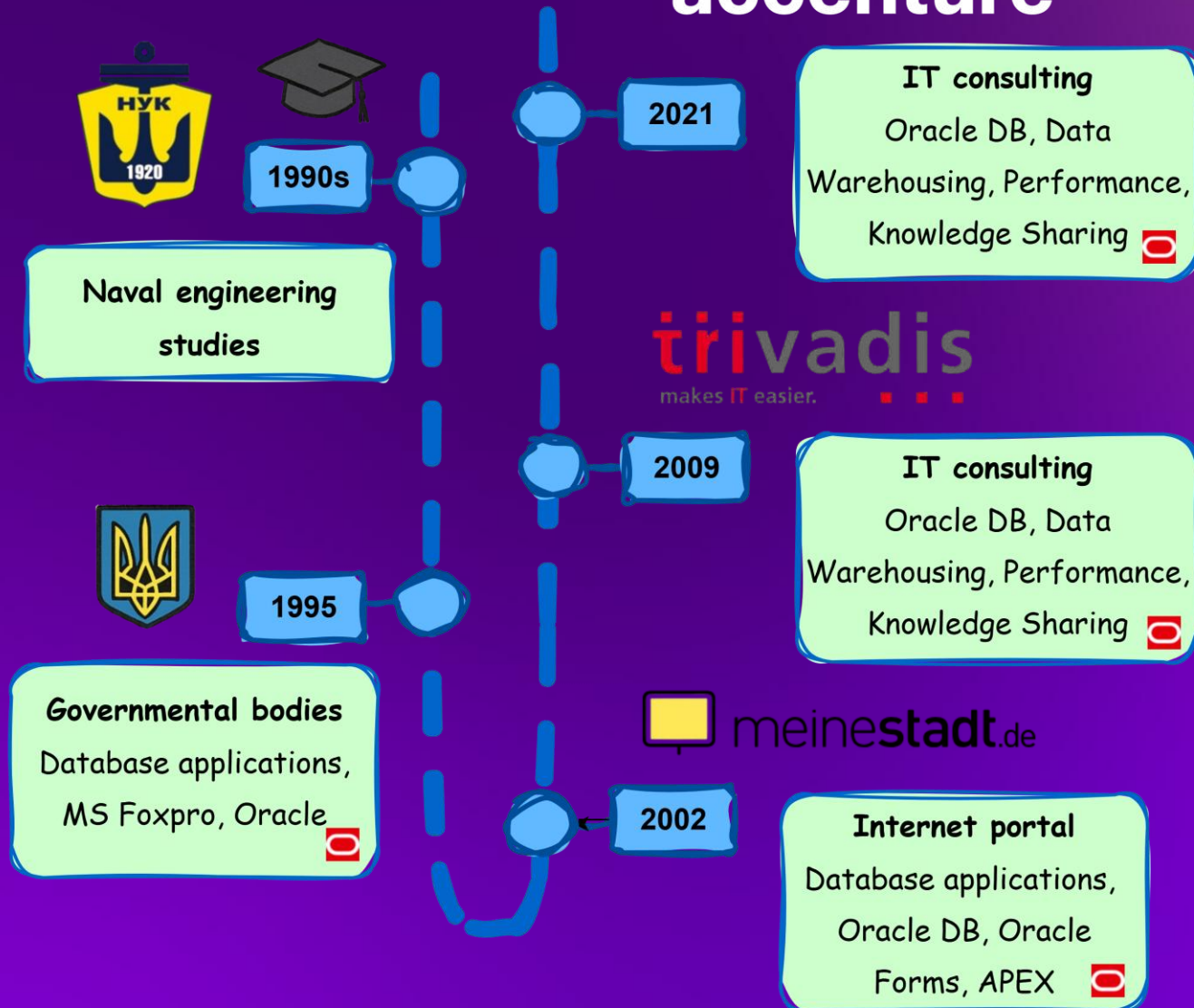
IN? EXISTS? JOIN?

Don't Ask the Internet.
Ask the Optimizer.

Andrej Pashchenko

Copyright © 2025 Accenture. All rights reserved.

>
accenture



Oracle ACE Pro



blog.sqlora.com



linktr.ee/andrejpashchenko



Problems with Internet Search

 Stack Overflow
2 Antworten · vor 12 Jahren

Oracle IN vs Exists difference? - sql

IN picks the list of matching values. EXISTS returns the boolean values like true or false. Exists is faster than in.

2 Antworten · Top-Antwort: simply put, EXISTS is usually used for checking whether rows that meet a c...

How to use Select **Exists** in **Oracle**? - Stack Overflow 5 Antworten 15. Mai 2013

How does **EXISTS** work in **Oracle**, and how does it ... 2 Antworten 16. Jan. 2012

Weitere Ergebnisse von stackoverflow.com

 AskTom
<https://asktom.oracle.com> · ords · Diese Seite übersetzen

IN & EXISTS - Ask TOM

Lets say the result of the subquery is small -- then **IN** is typically more appropriate. If both the subquery and the outer table are huge -- either might work as ...

 GeeksforGeeks
<https://www.geeksforgeeks.org> · ... · Diese Seite übersetzen

Difference between EXISTS and IN in PL/SQL

11.11.2024 — **EXISTS** is generally more efficient for large data sets and handles NULLs gracefully, making it ideal for related record checks. Understanding ...

 Oracle Forums
<https://forums.oracle.com> · post · Diese Seite übersetzen

Which is more faster - IN or EXISTS?

22.03.2011 — **Exist** is more faster than **IN** because IN doesn't use indexes at the time of fetching but Exist uses Index at the time of fetching.

Query using **EXISTS** operator is slower (or hanging) than the ... 20. Juni 2023

Convert IN to **EXISTS**. - Oracle Forums 22. Okt. 2021

Weitere Ergebnisse von forums.oracle.com

Time Reference

- What was true for Oracle 8i should not still be true for Oracle 23ai
- Even in very trustworthy sources like Ask TOM you can see posts that have been active for decades and whose initial answer changes over time to the opposite.

Vendor Reference

- The different implementations and optimizations by database vendors can be very important
- Even when asked explicitly, information about wrong vendors can sometimes appear quite high up in the results list.

Trustworthiness

- In-depth analysis with explanations or superficial “clickbait” posts like "always do this, never do that..."

**Thorough and extensive research is necessary...
But that's where AI comes to the rescue?!**

Problems with AI?

How do I choose between IN or EXISTS subquery in Oracle SQL? Or maybe I should use a join? Is there any difference in performance?

1. IN vs. EXISTS

Use IN when:

- The outer query's table is smaller and the subquery returns a small distinct set.
- You want to compare against a list of literal values or a small subquery.
- You don't expect NULLs in the subquery result (since NULL in an IN list can return no rows due to 3-valued logic).

Use EXISTS when:

- The subquery table is large and/or the outer table is large.
- You only need to check for existence, not values (e.g., "Is there a match?").
- You're using correlated subqueries (i.e., subquery refers to columns in the outer query).

A Slow Query Using EXISTS...

```
SQL> SELECT /*+ ... */ c.cust_id
,      c.cust_last_name, c.cust_first_name
FROM    sh.customers c
WHERE   c.cust_city = 'Koeln'
AND     EXISTS (SELECT 1
                FROM    sh.sales s
                WHERE   c.cust_id = s.cust_id );
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
	0	SELECT STATEMENT		1		44	00:00:18.36
*	1	FILTER		1		44	00:00:18.36
*	2	TABLE ACCESS FULL	CUSTOMERS	1	30	532	00:00:00.01
*	3	TABLE ACCESS FULL	SALES	532	131	44	00:00:18.35

- Slow subquery using **EXISTS**
- FILTER works as a loop evaluating a subquery for each unique value of the correlated column
- Column Starts shows **532** times for the full scan of the table SALES

A hint was used to force this execution plan



Rewrite As a Join

```
SQL> SELECT /*+ ... */ DISTINCT c.cust_last_name, c.cust_first_name
FROM   sh.customers c JOIN sh.sales s ON (c.cust_id = s.cust_id)
WHERE  c.cust_city = 'Koeln';
```

REJECTED

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
	0	SELECT STATEMENT		1		44	00:00:00.04
	1	SORT UNIQUE		1	77946	44	00:00:00.04
*	2	HASH JOIN		1	77946	4208	00:00:00.04
*	3	TABLE ACCESS FULL	CUSTOMERS	1	595	532	00:00:00.01
	4	TABLE ACCESS FULL	SALES	1	918K	918K	00:00:00.01

```
SQL> SELECT /*+ ... */ c.cust_last_name, c.cust_first_name
FROM   sh.customers c JOIN (SELECT DISTINCT cust_id FROM sh.sales) s
                        ON (c.cust_id = s.cust_id)
WHERE  c.cust_city = 'Koeln';
```

APPROVED

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
	0	SELECT STATEMENT		1		44	00:00:00.28
*	1	HASH JOIN		1	595	44	00:00:00.28
*	2	TABLE ACCESS FULL	CUSTOMERS	1	595	532	00:00:00.01
	3	VIEW		1	7014	7059	00:00:00.28
	4	SORT UNIQUE		1	7014	7059	00:00:00.28
	5	TABLE ACCESS FULL	SALES	1	918K	918K	00:00:00.02

- Rewriting as a **JOIN**
- Do not forget **DISTINCT**
- Full scan of the SALES table just once and a fast hash join
- DISTINCT in the main query is **wrong** unless it is needed for some other reasons
- Join to a **subquery** with DISTINCT!

The same hint was used to force this execution plan as in the previous slide

A Query Using IN

```
SQL> SELECT /*+ ... */ c.cust_id
,      c.cust_last_name, c.cust_first_name
FROM    sh.customers c
WHERE    c.cust_city = 'Koeln'
AND      c.cust_id IN (SELECT  s.cust_id
                        FROM    sh.sales s)
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
	0	SELECT STATEMENT		1		44	00:00:00.23
*	1	HASH JOIN		1	595	44	00:00:00.23
*	2	TABLE ACCESS FULL	CUSTOMERS	1	595	532	00:00:00.01
	3	VIEW	VW_NSO_1	1	7014	7059	00:00:00.23
	4	SORT UNIQUE		1	7014	7059	00:00:00.23
	5	TABLE ACCESS FULL	SALES	1	918K	918K	00:00:00.01

- A subquery using IN is fast as well
- It has the same execution plan as a query with a join

The same hint was used to force this execution plan as in the previous slide

Three approaches – two different plans

```
SQL> SELECT /*+ ... */ c.cust_id, c.cust_last_name, c.cust_first_name
FROM   sh.customers c
WHERE  c.cust_city = 'Koeln'
AND    c.cust_id IN (SELECT s.cust_id
                     FROM   sh.sales s)
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		44	00:00:00.23
* 1	HASH JOIN		1	595	44	00:00:00.23
* 2	TABLE ACCESS FULL	CUSTOMERS	1	595	532	00:00:00.01
3	VIEW	VW_NSO_1	1	7014	7059	00:00:00.23
4	SORT UNIQUE		1	7014	7059	00:00:00.23
5	TABLE ACCESS FULL	SALES	1	918K	918K	00:00:00.01

```
SQL> SELECT /*+ ... */ c.cust_id, c.cust_last_name, c.cust_first_name
FROM   sh.customers c JOIN (SELECT DISTINCT cust_id FROM sh.sales) s
                        ON (c.cust_id = s.cust_id)
WHERE  c.cust_city = 'Koeln';
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		44	00:00:00.28
* 1	HASH JOIN		1	595	44	00:00:00.28
* 2	TABLE ACCESS FULL	CUSTOMERS	1	595	532	00:00:00.01
3	VIEW		1	7014	7059	00:00:00.28
4	SORT UNIQUE		1	7014	7059	00:00:00.28
5	TABLE ACCESS FULL	SALES	1	918K	918K	00:00:00.02

```
SQL> SELECT /*+ ... */ c.cust_id
,      c.cust_last_name, c.cust_first_name
FROM   sh.customers c
WHERE  c.cust_city = 'Koeln'
AND    EXISTS (SELECT 1
               FROM   sh.sales s
               WHERE  c.cust_id = s.cust_id );
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		44	00:00:18.36
* 1	FILTER		1		44	00:00:18.36
* 2	TABLE ACCESS FULL	CUSTOMERS	1	30	532	00:00:00.01
* 3	TABLE ACCESS FULL	SALES	532	131	44	00:00:18.35

Revealing the hint:



optimizer_features_enable('8.1.7')

It is an ancient optimizer behaviour

How does it look these days?

```
SQL> SELECT c.cust_id
,      c.cust_last_name, c.cust_first_name
FROM    sh.customers c
WHERE    c.cust_city = 'Koeln'
AND      EXISTS (SELECT 1
                  FROM    sh.sales s
                  WHERE    c.cust_id = s.cust_id );
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
	0	SELECT STATEMENT		1		44	00:00:00.04
*	1	HASH JOIN SEMI		1	595	44	00:00:00.04
*	2	TABLE ACCESS FULL	CUSTOMERS	1	595	532	00:00:00.01
	3	TABLE ACCESS FULL	SALES	1	918K	918K	00:00:00.01

- HASH JOIN SEMI instead of a FILTER operation
- A query transformation Subquery Unnesting makes it possible



Subquery Unnesting – Key Facts



Subquery Unnesting is a query transformation.



Rewrites a query merging a subquery into the main query, either directly or using an inline view



Enables the optimizer to consider additional access paths, join methods and join orders, allows for parallelization



The transformation is cost based



Subject to validity checks – to retain the statement semantics



Can be controlled by the hints UNNEST, NO_UNNEST, but not to override the validity checks

A Query Using IN - With Subquery Unnesting

```
SQL> SELECT c.cust_id, c.cust_last_name, c.cust_first_name
FROM   sh.customers c
WHERE  c.cust_city = 'Koeln'
AND    c.cust_id IN (SELECT  s.cust_id
                     FROM    sh.sales s)
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		44	00:00:00.04
* 1	HASH JOIN SEMI		1	595	44	00:00:00.04
* 2	TABLE ACCESS FULL	CUSTOMERS	1	595	532	00:00:00.01
3	TABLE ACCESS FULL	SALES	1	918K	918K	00:00:00.01

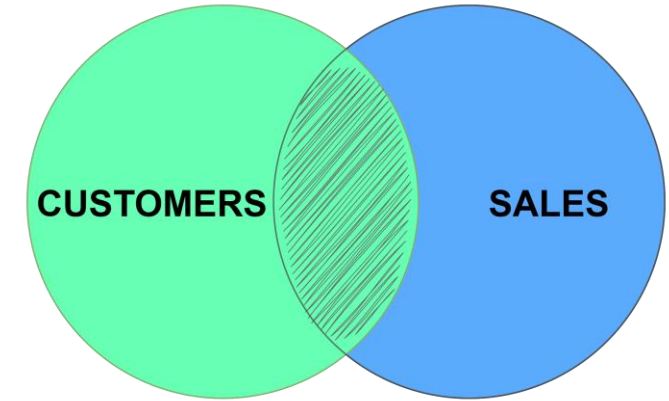
From CBO-trace

Final query after transformations:

```
SELECT "C"."CUST_ID" "CUST_ID", "C"."CUST_LAST_NAME" "CUST_LAST_NAME",
"C"."CUST_FIRST_NAME" "CUST_FIRST_NAME"
FROM "SH"."SALES" "S", "SH"."CUSTOMERS" "C"
WHERE "C"."CUST_CITY"='Koeln' AND "C"."CUST_ID"="S"."CUST_ID"
```

- If subquery unnesting could be applied, the query using IN has the same execution plan

Semi Join – Key Facts



Subquery Unnesting makes a **Semi Join** possible.



Semi Join is used for filtering the rows from left input based on the matches in the right input

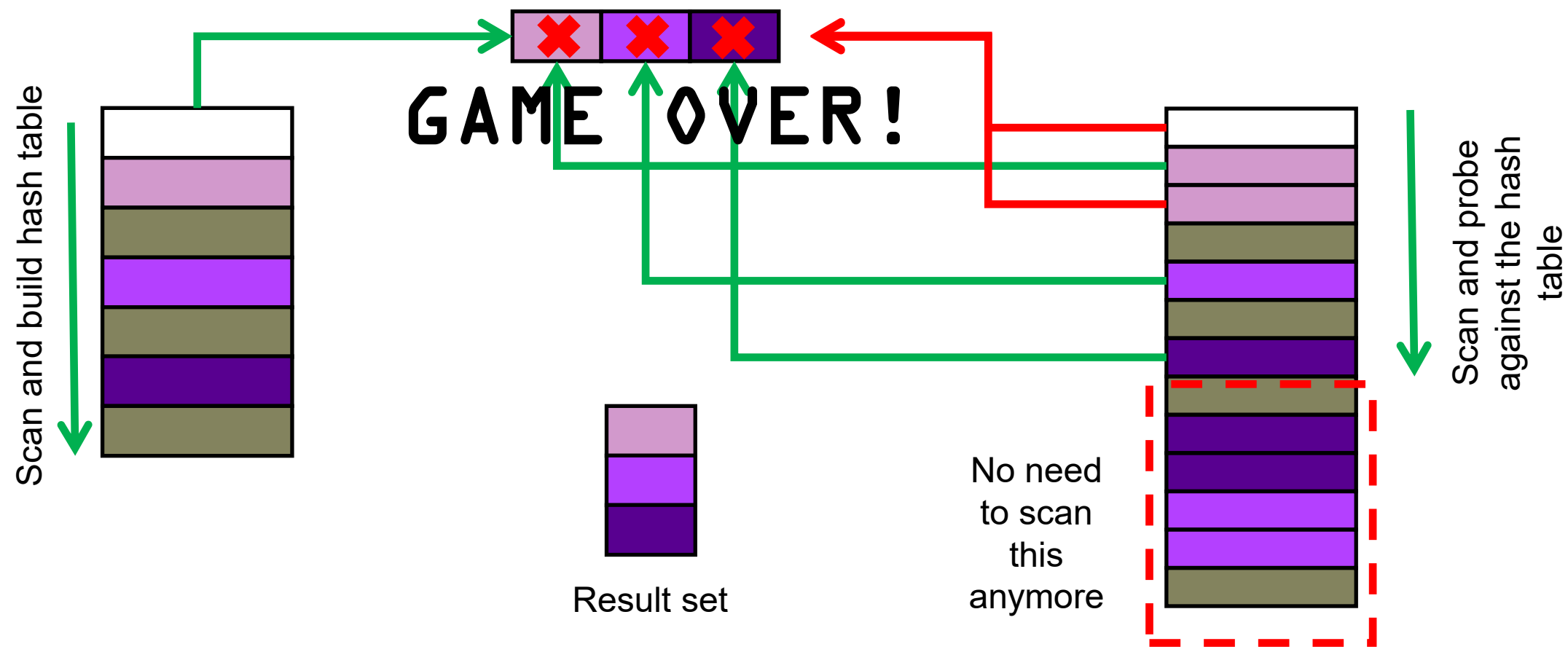


No data can be selected from the right input



No multiplication of rows from the left input in case of multiple matches in the right input

Hash Join Semi



Hash Join Semi Performance Boost

```
SELECT  c.cust_last_name, c.cust_first_name, c.cust_id
FROM    sh.customers c
WHERE   c.cust_city = 'Koeln'
AND     c.cust_last_name LIKE 'Nappi%'
AND     c.cust_id IN (SELECT  s.cust_id
                      FROM    sh.sales s );
```

CUST_LAST_NAME	CUST_FIRST_NAME	CUST_ID
Nappier	Beryl	2397

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
	0	SELECT STATEMENT		1		1	00:00:00.01
*	1	HASH JOIN SEMI		1	1	1	00:00:00.01
*	2	TABLE ACCESS FULL	CUSTOMERS	1	1	1	00:00:00.01
	3	TABLE ACCESS FULL	SALES	1	918K	13156	00:00:00.01

- Just ~13K out of ~900K rows processed



Hash Join Semi Performance Boost (2)

```
SQL> CREATE TABLE sales_ordered AS
SELECT * FROM sh.sales
ORDER BY CASE cust_id WHEN 2397 THEN 0 ELSE 1 END

Table created.

SQL> SELECT c.cust_id
,      c.cust_last_name, c.cust_first_name
FROM    sh.customers c
WHERE   c.cust_city = 'Koeln'
AND     c.cust_last_name LIKE 'Nappi%'
AND     c.cust_id IN (SELECT  s.cust_id
                        FROM    sh.sales_ordered s)
```

	Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
	0	SELECT STATEMENT		1		1	00:00:00.01
*	1	HASH JOIN SEMI		1	1	1	00:00:00.01
*	2	TABLE ACCESS FULL	CUSTOMERS	1	1	1	00:00:00.01
	3	TABLE ACCESS FULL	SALES_ORDERED	1	918K	256	00:00:00.01

- With favorable physical order the query stops almost immediately



Should a Join Be Considered?

```
SQL> SELECT /*+ ... */ DISTINCT c.cust_id, c.cust_last_name, c.cust_first_name
FROM   sh.customers c JOIN sh.sales s ON (c.cust_id = s.cust_id)
WHERE  c.cust_city = 'Koeln' AND c.cust_last_name LIKE 'Nappi%';
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.04
1	HASH UNIQUE		1	1	1	00:00:00.04
* 2	HASH JOIN		1	53	86	00:00:00.04
* 3	TABLE ACCESS FULL	CUSTOMERS	1	1	1	00:00:00.01
4	TABLE ACCESS FULL	SALES	1	918K	918K	00:00:00.01

```
SQL> SELECT /*+ ... */ c.cust_id, c.cust_last_name, c.cust_first_name
FROM   sh.customers c JOIN (SELECT DISTINCT cust_id FROM sh.sales) s
      ON (c.cust_id = s.cust_id)
WHERE  c.cust_city = 'Koeln' AND c.cust_last_name LIKE 'Nappi%';
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.04
1	VIEW	VM_NWVW_1	1	1	1	00:00:00.04
2	HASH UNIQUE		1	1	1	00:00:00.04
* 3	HASH JOIN		1	53	86	00:00:00.04
* 4	TABLE ACCESS FULL	CUSTOMERS	1	1	1	00:00:00.01
5	TABLE ACCESS FULL	SALES	1	918K	918K	00:00:00.01

- Both execution plans are less efficient than a semi join
- But to force them, `optimizer_features_enable` was set to values less than 12.1.0.1

Partial Join Evaluation

```
SQL> SELECT DISTINCT c.cust_id, c.cust_last_name, c.cust_first_name
FROM sh.customers c JOIN sh.sales s ON (c.cust_id = s.cust_id)
WHERE c.cust_city = 'Koeln' AND c.cust_last_name LIKE 'Nappi%';
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.01
1	HASH UNIQUE		1	1	1	00:00:00.01
* 2	HASH JOIN SEMI		1	1	1	00:00:00.01
* 3	TABLE ACCESS FULL	CUSTOMERS	1	1	1	00:00:00.01
4	TABLE ACCESS FULL	SALES	1	918K	13156	00:00:00.01

```
SQL> SELECT c.cust_id, c.cust_last_name, c.cust_first_name
FROM sh.customers c JOIN (SELECT DISTINCT cust_id FROM sh.sales) s
ON (c.cust_id = s.cust_id)
WHERE c.cust_city = 'Koeln' AND c.cust_last_name LIKE 'Nappi%';
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.01
1	VIEW	VM_NWVW_1	1	1	1	00:00:00.01
2	HASH UNIQUE		1	1	1	00:00:00.01
* 3	HASH JOIN SEMI		1	1	1	00:00:00.01
* 4	TABLE ACCESS FULL	CUSTOMERS	1	1	1	00:00:00.01
5	TABLE ACCESS FULL	SALES	1	918K	13156	00:00:00.01

- Starting with 12.1.0.1 **partial join evaluation** has been introduced, an optimization to avoid generating duplicate rows that would otherwise be removed by a distinct operator later in the plan
- A semi join is executed instead of a plain join
- Hints: (NO_)PARTIAL_JOIN
- Parameter: `_optimizer_partial_join_eval`



NULL Handling With IN?

Table A

ID
1
2
3
<u>(NULL)</u>

Table B

ID
1
2
2
<u>(NULL)</u>

- Select ID from Table A based on the presence in Table B
- **IN** is a shortcut for **=ANY()** or **=SOME()**
- Since only rows with the comparison result TRUE will qualify, it doesn't matter why other rows don't qualify (comparison evaluates to FALSE or UNKNOWN)
- Despite a three-valued logic using IN works as expected
- All three alternatives (IN, EXISTS, JOIN) behave the same

```
SQL> WITH a(id) AS
      (values (1),(2),(3),(null))
,      b(id) AS
      (values (1),(2),(2),(null))
SELECT *
FROM   a
WHERE  id IN (SELECT id FROM b)
```

ID
1
2

```
SQL> WITH a(id) AS
      (values (1),(2),(3),(null))
,      b(id) AS
      (values (1),(2),(2),(null))
SELECT *
FROM   a
WHERE  EXISTS (SELECT 1 FROM b
               WHERE  b.id = a.id)
```

ID
1
2

```
SQL> WITH a(id) AS
      (values (1),(2),(3),(null))
,      b(id) AS
      (values (1),(2),(2),(null))
SELECT DISTINCT a.*
FROM   a JOIN b ON b.id = a.id
```

ID
1
2

NULL Handling With NOT IN!

Table A

ID
1
2
3
(NULL)

Table B

ID
1
2
2
(NULL)

- Select ID from Table A based on the absence in Table B
- NOT IN is a shortcut for `<>ALL()`
- If NULL's are present in the NOT IN-subquery, the query returns **no rows**

```
SQL> WITH a(id) AS
      (values (1),(2),(3),(null))
,      b(id) AS
      (values (1),(2),(2),(null))
SELECT *
FROM   a
WHERE  id NOT IN (SELECT id FROM b)

no rows selected.
```

```
SQL> WITH a(id) AS
      (values (1),(2),(3),(null))
,      b(id) AS
      (values (1),(2),(2),(null))
SELECT *
FROM   a
WHERE  NOT EXISTS (SELECT 1 FROM b
                  WHERE b.id = a.id)

      ID
-----
      3
2 rows selected.
```

```
SQL> WITH a(id) AS
      (values (1),(2),(3),(null))
,      b(id) AS
      (values (1),(2),(2),(null))
SELECT DISTINCT a.*
FROM   a LEFT JOIN b ON b.id = a.id
WHERE  b.id IS NULL

      ID
-----
      3
2 rows selected.
```

The Anti Join – NOT IN and NOT EXISTS

```
SQL> SELECT DISTINCT c.cust_income_level
FROM   customers c
WHERE  c.cust_city = 'Koeln'
AND    NOT EXISTS (SELECT /*+ no_unnest */ 1
                   FROM   customers c2
                   WHERE  c2.cust_city = 'Hamburg'
                   AND    c2.cust_income_level=c.cust_income_level)
```

CUST_INCOME_LEVEL

K: 250,000 - 299,999

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.02
1	HASH UNIQUE		1	12	1	00:00:00.02
* 2	FILTER		1		10	00:00:00.02
* 3	TABLE ACCESS FULL	CUSTOMERS	1	474	532	00:00:00.01
* 4	TABLE ACCESS FULL	CUSTOMERS	12	5	11	00:00:00.02

```
2 - filter( NOT EXISTS (SELECT /*+ NO_UNNEST */ 0 FROM "CUSTOMERS" "C2"
                        WHERE "C2"."CUST_CITY"='Hamburg'
                        AND "C2"."CUST_INCOME_LEVEL"=:B1))
3 - filter("C"."CUST_CITY"='Koeln')
4 - filter(("C2"."CUST_CITY"='Hamburg' AND "C2"."CUST_INCOME_LEVEL"=:B1))
```

- Without subquery unnesting the execution plan is based on FILTER operation
- The subquery is evaluated once per distinct value of the connecting column

The Anti Join – NOT IN and NOT EXISTS

```
SELECT DISTINCT c.cust_income_level
FROM customers c
WHERE c.cust_city = 'Koeln'
AND c.cust_income_level NOT IN (SELECT /*+ no_unnest */ c2.cust_income_level
                                FROM customers c2
                                WHERE c2.cust_city = 'Hamburg')
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		1	00:00:00.01
1	HASH UNIQUE		1	12	1	00:00:00.01
* 2	FILTER		1		10	00:00:00.01
* 3	TABLE ACCESS FULL	CUSTOMERS	1	474	532	00:00:00.01
* 4	TABLE ACCESS FULL	CUSTOMERS	12	5	11	00:00:00.01

```
2 - filter( NOT EXISTS (SELECT /*+ NO_UNNEST */ 0 FROM "CUSTOMERS"
                        "C2" WHERE "C2"."CUST_CITY"='Hamburg' AND
                        LNNVL("C2"."CUST_INCOME_LEVEL"<>:B1)))
3 - filter("C"."CUST_CITY"='Koeln')
4 - filter(("C2"."CUST_CITY"='Hamburg' AND
            LNNVL("C2"."CUST_INCOME_LEVEL"<>:B1)))
```

```
-- This query will be semantically the same
SELECT DISTINCT c.cust_income_level FROM customers c
WHERE c.cust_city = 'Koeln'
AND NOT EXISTS (SELECT /*+ no_unnest */ 1
                FROM customers c2
                WHERE c2.cust_city = 'Hamburg'
                AND LNNVL(c2.cust_income_level<>c.cust_income_level));
```

- Oracle rewrites NOT IN as NOT EXISTS but because of LNNVL function it behaves like NOT IN
- LNNVL returns TRUE if the tested condition is FALSE or UNKNOWN

Performance Of The Anti-Join

```
SQL> SELECT --+ optimizer_features_enable('10.2.0.4')
        DISTINCT c.cust_income_level
FROM      customers c
WHERE     c.cust_city = 'Koeln'
AND       c.cust_income_level NOT IN
        (SELECT c2.cust_income_level
         FROM    customers c2
         WHERE   c2.cust_city = 'Hamburg')
```

	Id	Operation	Name
	0	SELECT STATEMENT	
	1	HASH UNIQUE	
*	2	FILTER	
*	3	TABLE ACCESS FULL	CUSTOMERS
*	4	TABLE ACCESS FULL	CUSTOMERS

```
SQL> SELECT --+ optimizer_features_enable('10.2.0.4')
        DISTINCT c.cust_income_level
FROM      customers c
WHERE     c.cust_city = 'Koeln'
AND      c.cust_income_level IS NOT NULL
AND       c.cust_income_level NOT IN
        (SELECT c2.cust_income_level
         FROM    customers c2
         WHERE   c2.cust_city = 'Hamburg'
         AND      c2.cust_income_level IS NOT NULL);
```

	Id	Operation	Name
	0	SELECT STATEMENT	
	1	HASH UNIQUE	
*	2	HASH JOIN RIGHT ANTI	
*	3	TABLE ACCESS FULL	CUSTOMERS
*	4	TABLE ACCESS FULL	CUSTOMERS

- Before 11g if NULL's were expected (no constraints defined), no subquery unnesting took place
- Workaround was to add additional NOT NULL conditions in the WHERE clause



Null-Aware Anti-Joins

```
SQL> SELECT
      DISTINCT c.cust_income_level
FROM    customers c
WHERE   c.cust_city = 'Koeln'
AND     c.cust_income_level NOT IN
      (SELECT c2.cust_income_level
       FROM   customers c2
       WHERE  c2.cust_city = 'Hamburg')
```

	Id	Operation	Name
	0	SELECT STATEMENT	
	1	HASH UNIQUE	
*	2	HASH JOIN RIGHT ANTI NA	
*	3	TABLE ACCESS FULL	CUSTOMERS
*	4	TABLE ACCESS FULL	CUSTOMERS

```
SQL> SELECT DISTINCT c.cust_income_level
FROM    customers c
WHERE   c.cust_city = 'Koeln'
AND     c.cust_income_level NOT IN
      (SELECT c2.cust_income_level
       FROM   customers c2
       WHERE  c2.cust_city = 'Hamburg'
       AND    c2.cust_income_level IS NOT NULL);
```

	Id	Operation	Name
	0	SELECT STATEMENT	
	1	HASH UNIQUE	
*	2	HASH JOIN RIGHT ANTI SNA	
*	3	TABLE ACCESS FULL	CUSTOMERS
*	4	TABLE ACCESS FULL	CUSTOMERS

- Starting with 11g Null-Aware Anti-Joins have been introduced. Subquery unnesting can take place even with nullable columns
- Warning: it doesn't change the behaviour of NOT IN with respect to the NULL's



SQL is 4GL — Let the Optimizer Drive



SQL is **declarative**: you describe *what* you want, not *how* to do it



Thinking “IN vs EXISTS vs JOIN” affects performance = **procedural mindset**



Write the SQL that best matches your business logic



Let the optimizer transform them into efficient plans



Learn to read execution plans - that's the real skill



Thank You