MySQL®
Conference & Expo

Presented by,
MySQL AB® & O’Reilly Media, Inc.

New subquery optimizations in MySQL 6.0

Sergey Petrunia
sergefp@mysql.com

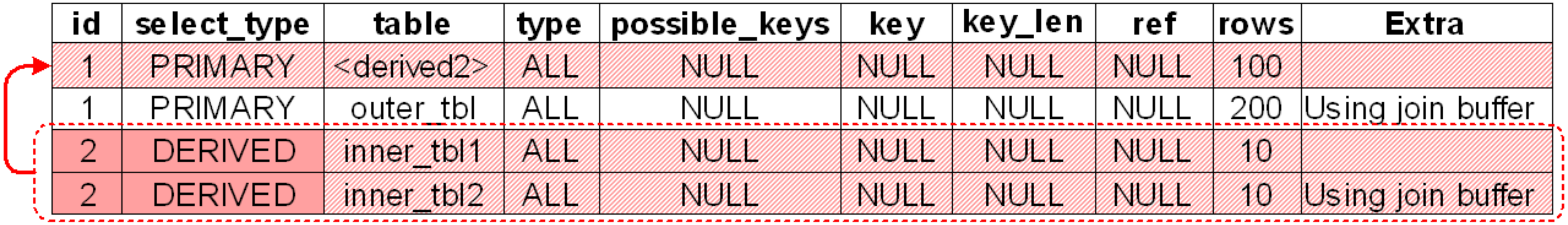
Background: subquery processing before 6.0

- FROM subqueries are pre-materialized (early)
- Scalar-context subqueries use straightforward evaluation
- Predicate subqueries
  - May perform two kinds of rewrites
  - Then use straightforward evaluation
- Originally implemented in MySQL 4.1
  by Sinisa (FROM subqueries) and Sanja (all other kinds)

Processing subqueries in the FROM clause

```
SELECT … FROM (SELECT …) AS tbl WHERE …
```

- Execution steps
  1. Optimize the subquery SELECT
  2. Execute it and capture result into a temporary table
  3. Optimize and execute the parent SELECT

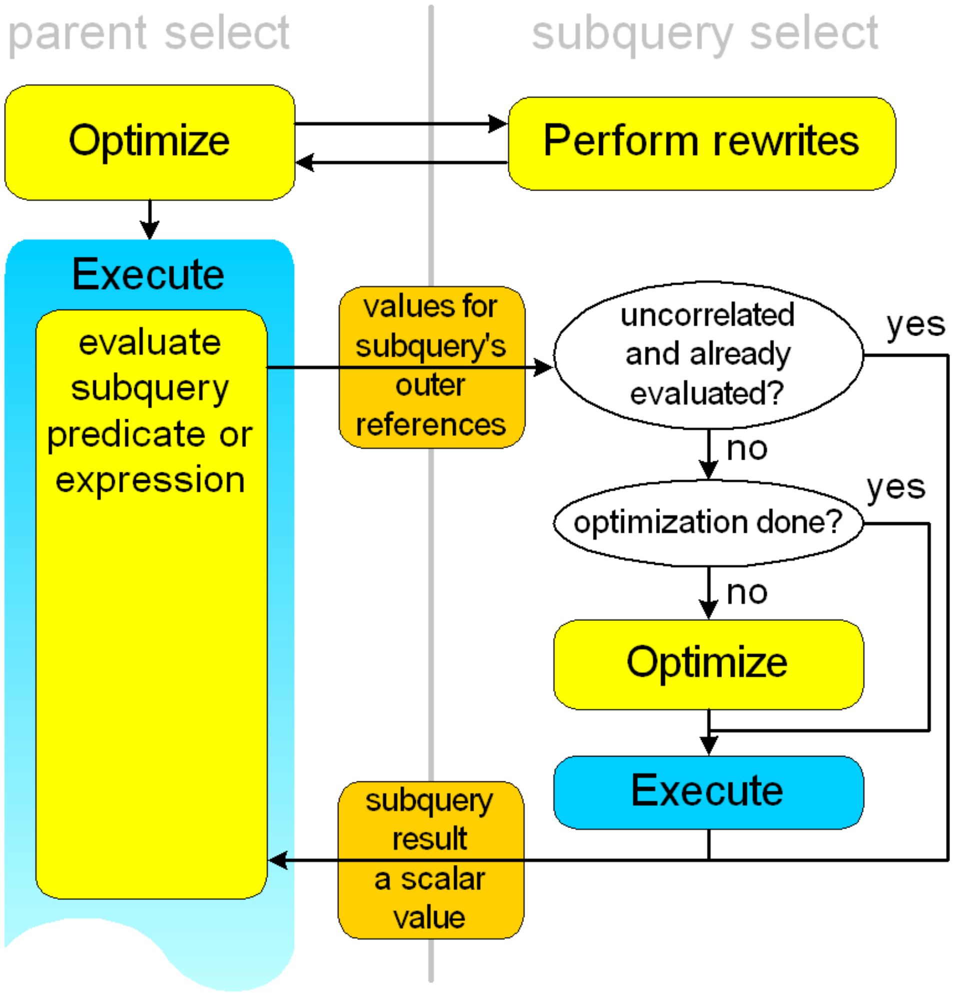


id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	100	
1	PRIMARY	outer_tbl	ALL	NULL	NULL	NULL	NULL	200	Using join buffer
2	DERIVED	inner_tbl1	ALL	NULL	NULL	NULL	NULL	10	
2	DERIVED	inner_tbl2	ALL	NULL	NULL	NULL	NULL	10	Using join buffer

- Properties
  - No optimization (can get some if you define/use a VIEW equivalent to subquery)
  - EXPLAIN command runs the subquery and thus can be very slow

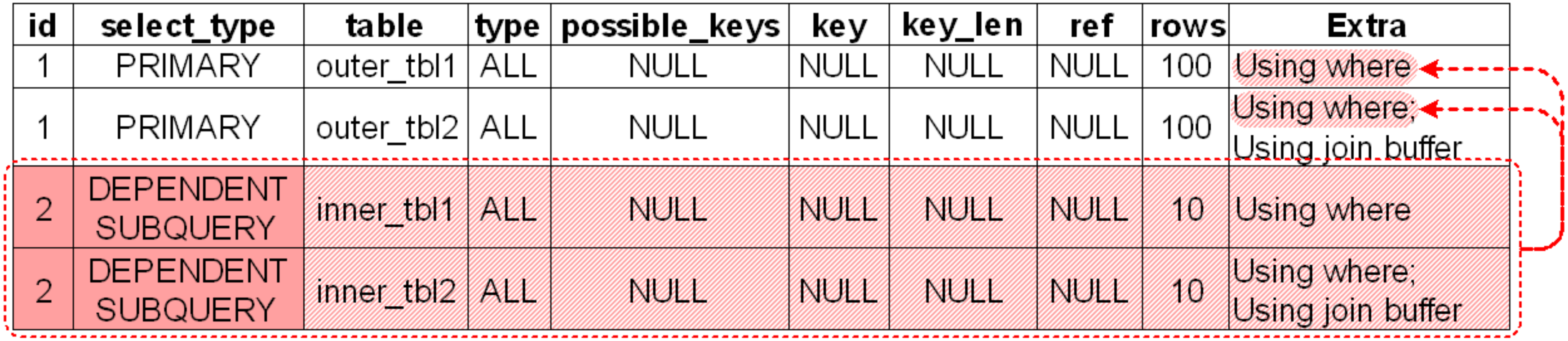
Straightforward subquery evaluation

- Used for all kinds of subqueries other than FROM:
  - expr IN (SELECT ...)
  - EXISTS (SELECT ...)
  - expr $\lesseqgtr \begin{matrix}\text{ALL}\\ \text{SOME}\\ \text{ANY}\end{matrix}$ (SELECT....)
  - scalar context subqueries
- Subquery is optimized once, all re-evaluations are done using the same plan
- Uncorrelated subqueries are evaluated only once


parent select
subquery select
Optimize
Perform rewrites
Execute
evaluate subquery predicate or expression
values for subquery's outer references
uncorrelated and already evaluated?
yes
no
optimization done?
yes
no
Optimize
Execute
subquery result a scalar value

Straightforward subquery evaluation (contd)

```
SELECT … FROM outer_tbl1,outer_tbl2
  WHERE expr IN (SELECT inner_expr
                 FROM inner_tbl1, inner_tbl2 WHERE … )
```



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	outer_tbl1	ALL	NULL	NULL	NULL	NULL	100	Using where
1	PRIMARY	outer_tbl2	ALL	NULL	NULL	NULL	NULL	100	Using where; Using join buffer
2	DEPENDENT SUBQUERY	inner_tbl1	ALL	NULL	NULL	NULL	NULL	10	Using where
2	DEPENDENT SUBQUERY	inner_tbl2	ALL	NULL	NULL	NULL	NULL	10	Using where; Using join buffer

- select_type="SUBQUERY" means subquery is run only once.
- "DEPENDENT SUBUQERY" means it is re-run on every re-evaluation
  - Subqueries in WHERE/ON are re-evaluated when their WHERE AND-part is evaluated, which is as soon as possible
  - Subqueries in select list, HAVING, etc are re-evaluated for every record combination

Subquery rewrites: IN->EXISTS (1)

"Inform the subquery about which part of its resultset we're interested in"

- IN→EXISTS transformation:

```
OuterExpr IN (SELECT InnerExpr FROM …
              WHERE subq_where)

→

EXISTS (SELECT 1 FROM …
        WHERE subq_where AND
              InnerExpr = OuterExpr)
```

Things to note

- Uncorrelated subquery becomes correlated
- This is a simplifed description, not counting cases with NULLs

Subquery rewrites: MIN/MAX (2)

“Inform the subquery about which part of its resultset we're interested in”

- MIN/MAX Transformation

```
OuterExpr > ALL(SELECT InnerExpr FROM ...)
→
OuterExpr > (SELECT MAX(InnerExpr)FROM ... )
```

handles all similar cases with

$$\text{OuterExpr} \lesseqgtr \begin{matrix}\text{ALL}\\ \text{SOME}\\ \text{ANY}\end{matrix} \text{(SELECT...)}$$

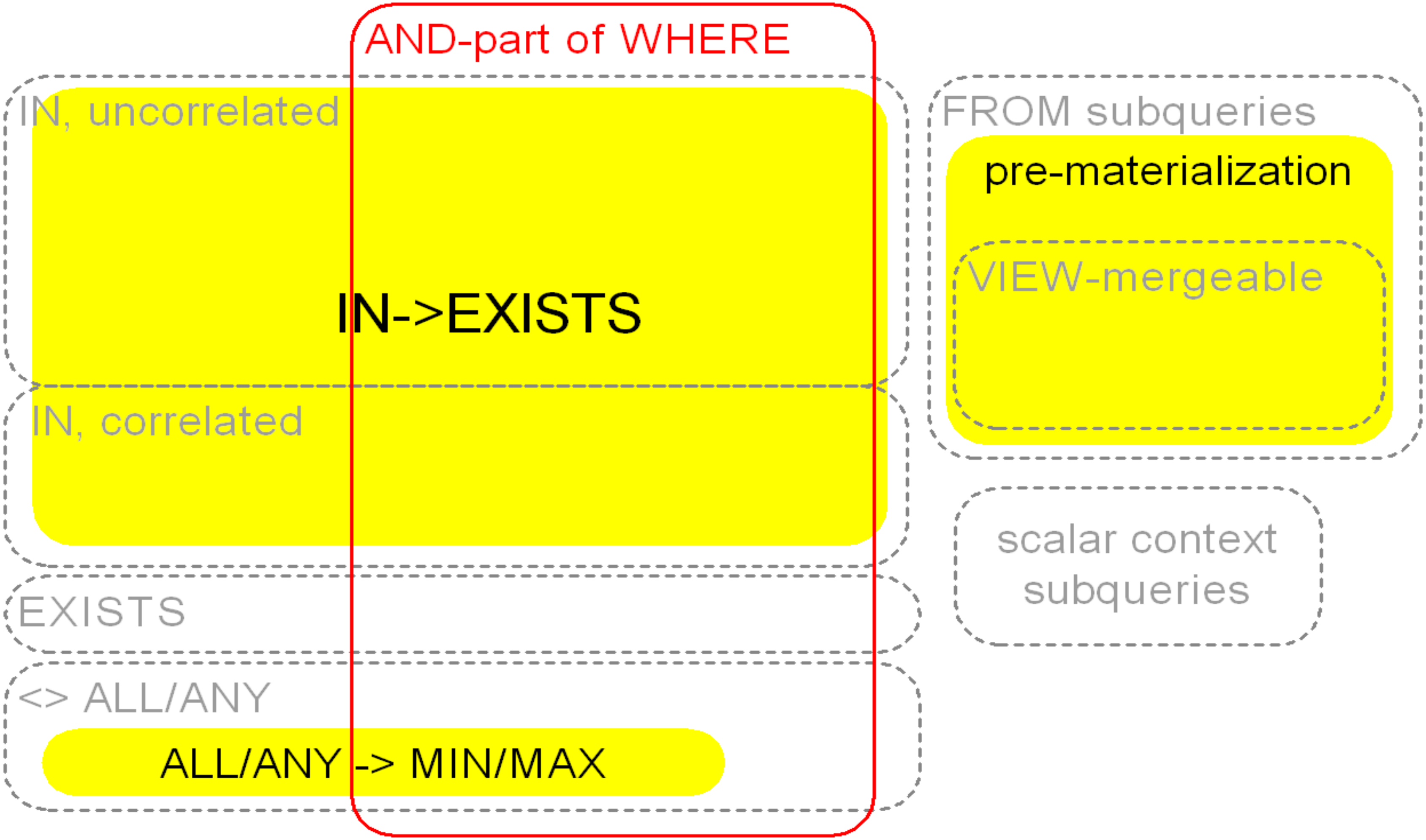
* simplifed description, not counting cases with NULLs or subqueries returning zero rows

Current state of subqueries: summary

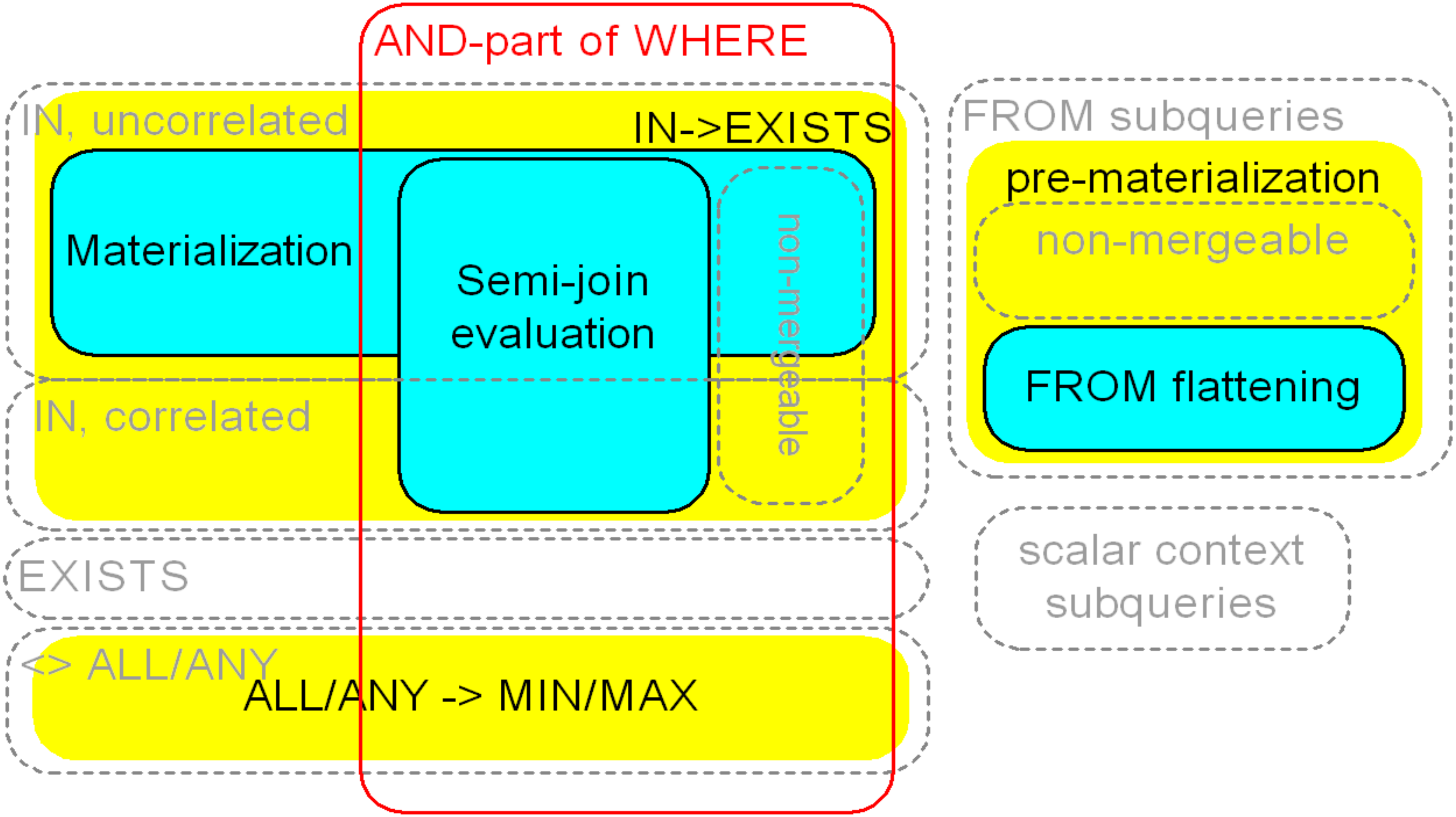
- FROM subqueries
  - are always pre-materialized, exactly like in the query

- Scalar and predicate subqueries
  - Optimized using two rule-based rewrites:
    - IN→EXISTS (pushdown the IN-equality)
    - ALL/ANY→MIN/MAX
  - Evaluated using straightforward outer-to-inner strategy
    - As early is possible if located in the WHERE
    - “Very late” if located in other parts of the query
  - Are evaluated only once if uncorellated

Subquery optimization map: 6.0



Subquery optimization map: 6.0



Semi-join subquery optimizations

- A practically important kind of subqueries:

```
SELECT * FROM ...
WHERE query_where AND
      outer_expr IN (SELECT inner_expr
                     FROM ... WHERE ...)
```

- In relational algebra, semi-join is defined as:

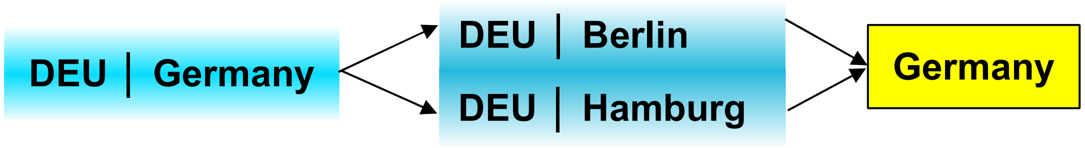
outertbl SEMI JOIN *innertbl* ON *sj_cond* =
$\{$*outertbl*.row $|$ $\exists$ *innertbl*.row, *sj_cond*(*outertbl*.row, *innertbl*.row)$\}$

- A subquery is processed as a semi-join if
  - it is an IN/=ANY subquery
  - it is an AND-part of the WHERE clause
  - it is not a UNION, has no aggregates or ORDER BY ... LIMIT
  - SELECT DISTINCT or “dummy” GROUP BY are allowed

Semi-join vs. inner join semantics

The difference is in duplicate outer row combinations

```
SELECT Country.Name FROM Country
  WHERE Code IN (SELECT CountryCode FROM City
                 WHERE Population > 1M)
```



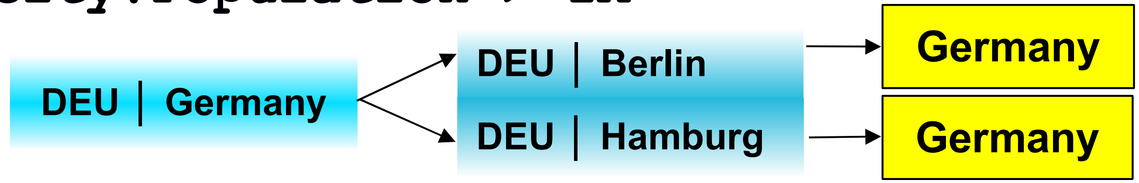
DEU | Germany

DEU | Berlin

DEU | Hamburg

Germany

```
SELECT Country.Name FROM Country, City
  WHERE Country.Code=City.CountryCode AND
        City.Population > 1M
```



DEU | Germany

DEU | Berlin

DEU | Hamburg

Germany

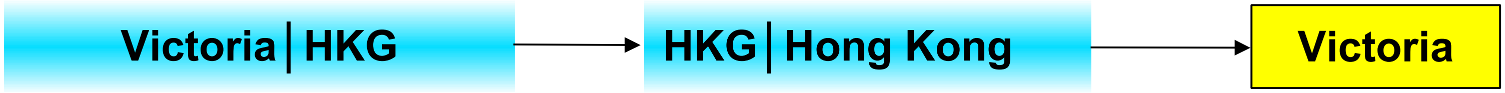
Germany

=> *semi-join is like inner join but we need some way to remove the duplicates*

Semi-join strategy #1: Table pullout

If a subquery table is functionally dependent on the parent query tables, it can be "pulled out" of the subquery

```
SELECT City.Name FROM City
WHERE City.Country IN (SELECT Country.Code FROM Country
                       WHERE Country.SurfaceArea < 2K)
```



Victoria | HKG → HKG | Hong Kong → Victoria

is converted into

```
SELECT City.Name FROM City, Country
WHERE City.Country = Country.Code AND
      Country.SurfaceArea < 2K)
```

If the subquery has several tables, will pull out those tables that don't generate duplicate matches

Semi-join strategy #1: Table pullout: example

```
EXPLAIN EXTENDED SELECT City.Name FROM City
  WHERE City.Country IN (SELECT Country.Code FROM Country
                       WHERE Country.SurfaceArea < 10);
SHOW WARNINGS;
```

In MySQL 4.1/5.x :

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	City	ALL	NULL	NULL	NULL	NULL	4079	Using where
2	DEPENDENT SUBQUERY	Country	unique_subquery	PRIMARY, SurfaceArea	PRIMARY	3	func	1	Using where

select \`world\`.\`City\`.\`Name\` AS \`Name\` from \`world\`.\`City\` **where <in_optimizer>**(\`world\`.\`City\`.\`Country\`,**<exists>(<primary_index_lookup>**(<cache>(\`world\`.\`City\`.\`Country\`) in Country on PRIMARY where ...

In MySQL 6.0 :

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	Country	range	PRIMARY, SurfaceArea	SurfaceArea	4	NULL	3	Using index condition; Using MRR
1	PRIMARY	City	ref	Country	Country	3	Country.Code	18	

select \`world\`.\`City\`.\`Name\` AS \`Name\` **from \`world\`.\`Country\` join \`world\`.\`City\`** where ((\`world\`.\`City\`.\`Country\` = \`world\`.\`Country\`.\`Code\`) and (\`world\`.\`...

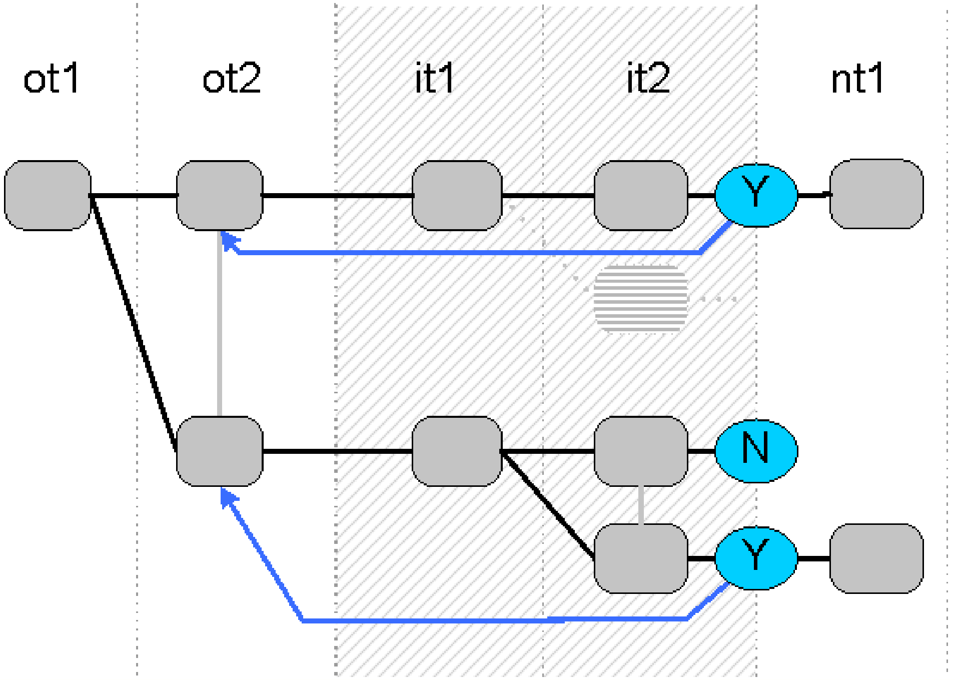
Semi-join strategy #1: Table pullout: summary

- In two words, this is subquery-to-join conversion
- Properties
  - It is rule-based, pullout is done whenever possible
  - It enables the join optimizer to make a cost-based choice from a greater variety of query plans (including a plan that is eqivalent to pre-6.0 server strategy)
- Applicability
  - Pullout is done before any other semi-join strategy considerations
  - Can handle correlated subqueries (analogous functionality in PostgreSQL, surprisingly, doesn't)
  - Can handle arbitarily deep subquery nesting

Semi-join strategy #2: FirstMatch

Short-cut enumeration of subquery tables as soon as we get first matching row combination

```
SELECT * FROM ot1,ot2,nt1, ...
WHERE expr(ot1,ot2) IN (SELECT … FROM it1,it2 …)
```



```
● if (table condition satisfied) {
    do join with next tables;
    jump out to the last otN;
  } else {
    discard row combination;
    continue current table scan;
  }
```

Semi-join strategy #2: FirstMatch: example

```
EXPLAIN EXTENDED
SELECT Name FROM Country
WHERE
  Country.Continent='Europe'  AND
  Country.Code IN (SELECT City.Country FROM City
                   WHERE City.ID != Country.Capital AND
                    Population > 1M)
```

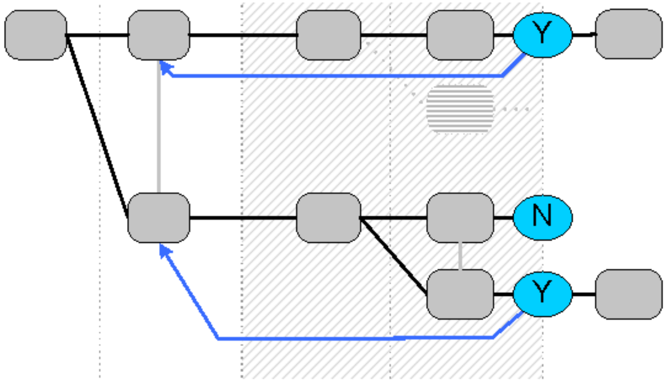
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	Country	ref	PRIMARY, Continent	Continent	21	const	8	Using index condition;

1	PRIMARY	City	ref	Population, Country	Country	3	Country.Code	18	Using where;
									FirstMatch(Country)

select \`world\`.\`Country\`.\`Name\` AS \`Name\` from **\`world\`.\`Country\` semi join (\`world\`.\`City\`)** where ((\`world\`.\`City\`.\`Country\` = \`world\`.\`Country\`.\`Code\`) and (\`world\`.\`Country\`.\`Continent\` = 'Europe') and (\`world\`.\`City\`.\`ID\` <> \`world\`.\`Country\`.\`Capital\`) and (\`world\`.\`City\`.\`Population\` > 1000000))

Semi-join strategy #2: FirstMatch (contd)

- Similar to IN->EXISTS
  - Strictly outer-to-inner join order



- Better than IN->EXISTS
  - Don't have to evaluate subquery immediately after outer tables it refers to:

```
SELECT employee.*
  FROM employee NATURAL JOIN office
  WHERE employee.hire_date > '2008-01-01' AND
        office.country='EU' AND
        employee_id IN (SELECT employee_id
                        FROM conference_speaker)
```

4.1/5.x: employee--conference_speaker, office

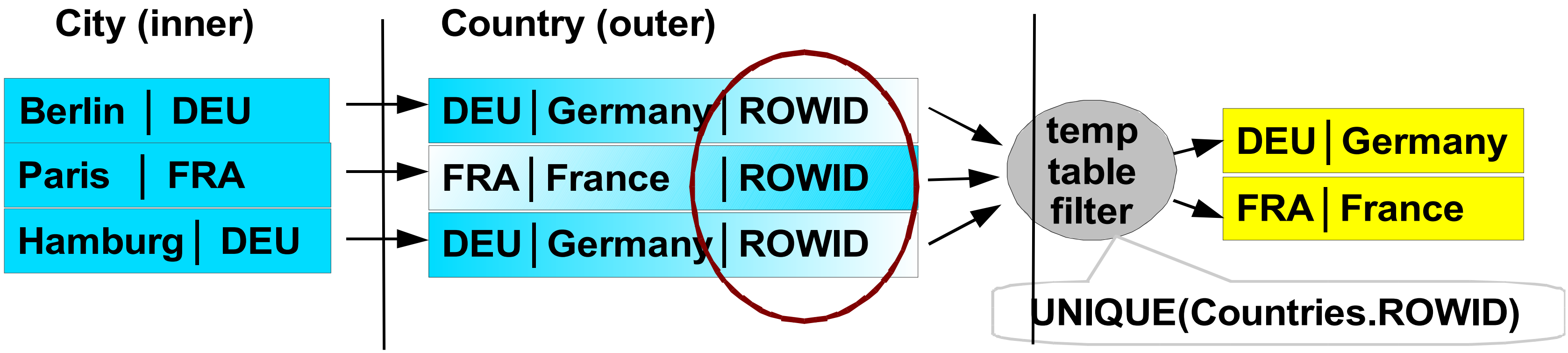
6.0 employee, office, conference_speaker

  - Doesn't force IN->EXISTS rewrite so allows for other optimizations

Semi-join strategy #3: duplicate elimination

Use temporary table with unique key (or constraint) to eliminate duplicate row combinations of outer tables

```
SELECT Country.Name FROM Country
WHERE Code IN (SELECT Country FROM City
               WHERE Population > 1M)
```



Duplicate elimination: example

```
SELECT Country.Name FROM Country
  WHERE Code IN (SELECT Country FROM City
                 WHERE Population > 1M)
```

select_type	table	type	possible_keys	key	key_len	ref	rows	Extra

PRIMARY	City	range	Population, Country	Population	4	NULL	246	Using index condition;
								Using MRR;
								Start temporary

PRIMARY	Country	eq_ref	PRIMARY	PRIMARY	3	City.Country	1	**End temporary**

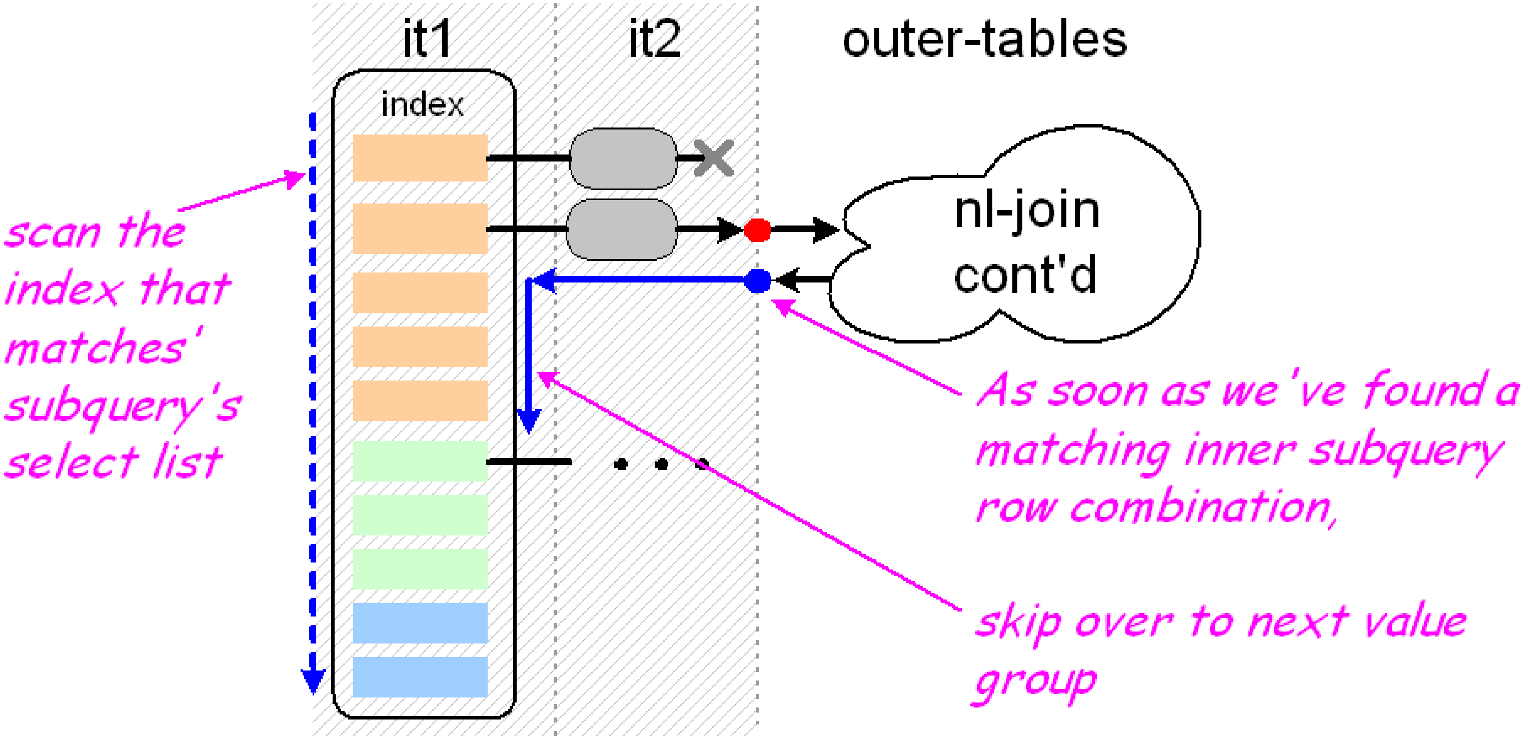
select `world`.`Country`.`Name` AS `Name` from **`world`.`Country` semi join (`world`.`City`)** where ((`world`.`Country`.`Code` = `world`.`City`.`Country`) and (`world`.`City`.`Population` > 1000000))

- Can have outer/inner tables in any order
  - Including Inner-to-outer which wasn't possible till 6.0
- Can handle correlated subqueries
- Similar to materialization (uses memory, hash lookups)
  - But different – temp table gets **outer** table's rows (not inner) **that have matches** (materialization stores all inner table rows)

Semi-join strategy #4: InsideOut

Scan inner table(s) in a way that doesn't produce duplicates

```
SELECT … FROM outer_tbl
WHERE outer_expr IN (SELECT it1.poor_key
                     FROM it1,it2 WHERE cond(it1, it2))
```



scan the index that matches' subquery's select list

As soon as we've found a matching inner subquery row combination,

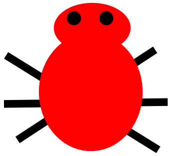
skip over to next value group

Semi-join strategy #4: InsideOut: example

```
SELECT * FROM outer_tbl
  WHERE key1 IN (SELECT poor_key FROM inner_tbl);
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	inner_tbl	index	poor_key	poor_key	5	NULL	10000	Using index; LooseScan
1	PRIMARY	outer_tbl	ref	key1	key1	5	inner_tbl.poor_key	1	Using index

- Only inner-to-outer join orders
- Subquery must be uncorrelated
- There must be an index that covers subquery' select list
- At the moment usable with 'ref' or 'index'
  - Should be also usable with 'range' but not yet
- CAUTION there are known bugs.

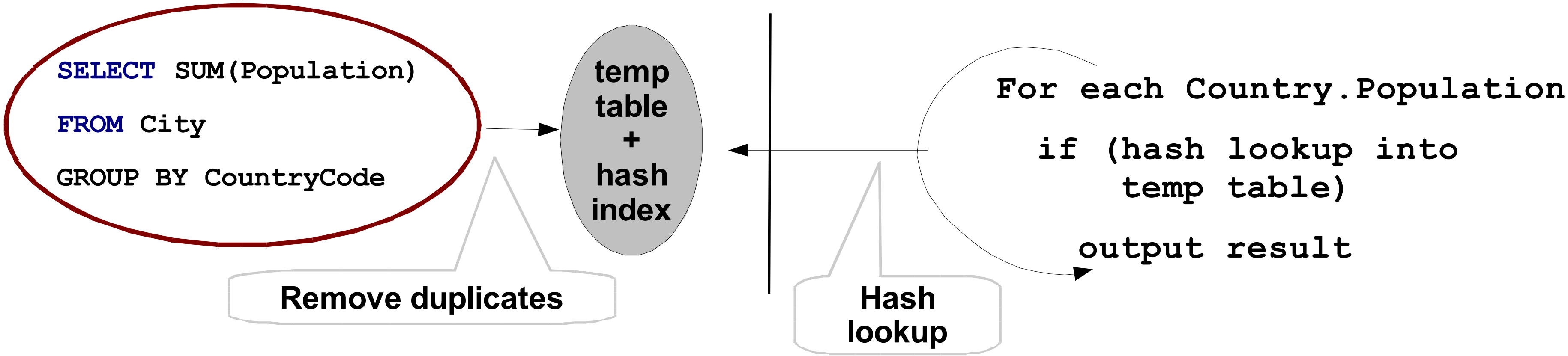


Materialization strategy

Use temporary table with unique constraint to

- Materialize the subquery result
- Create unique hash index on it
- Lookup outer tuples into temp table index

```
SELECT Country.Name FROM Country
WHERE Population IN (SELECT SUM(Population)
                     FROM City
                     GROUP BY CountryCode)
```



```
SELECT SUM(Population)

FROM City

GROUP BY CountryCode
```

```
For each Country.Population

  if (hash lookup into
        temp table)

     output result
```

Controlling new subquery optimizations

- Currently, a server variable:

```
@@optimizer_switch =
  'no_semijoin,no_materialization'
```

(like set-type column: any order, no space after comma)

- this will likely to change into being a part of a bigger optimization on/off scheme (WL#4046)

- Already seeing a need for hints but no WL entry for this yet thinking of syntax like

```
  outer_expr IN (SELECT no_materialize ...)
```

Benchmarking new optimizations

- A look at standard benchmarks: DBT-{1,2,3}
  - DBT-3 has 10 subquery cases
  - Of which 8 are not covered by new optimizations (2 are covered)
  - Query #18: covered (materialization), execution times:

Engine	Query time
MySQL 6.0, no new optimizations	> 3 hours
MySQL 6.0, materialization	3.76 sec
PostgreSQL	6.52 sec

~1800 times faster now

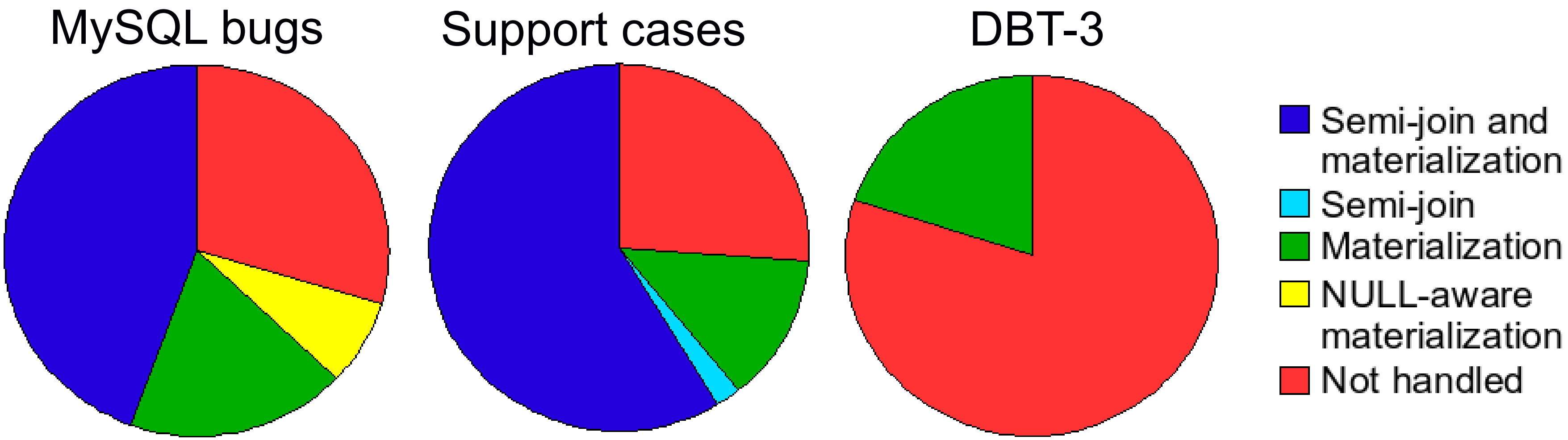
  - Query #16: will be covered by NULL-aware materialization

Engine	Query time
MySQL 6.0, no new optimizations	0.55 sec
PostgreSQL	1.16 sec

* used DBT3 scale=1, InnoDB, all default settings

Benchmarking new optimizations (contd)

- MySQL bugs/customer cases and DBT-3 have different subquery populations



- No idea about the reason of the difference
- We intend to develop some subquery benchmark to cover subqueries like in MySQL bugs/support db

Benchmarking new optimizations (contd)

- MySQL bugs/customer issues that are easily repeatable
  - Found 10 subquery cases
  - Taking PostgreSQL's speed as 1.0:

	No 6.0 optimizations	Materialization	Semi-join
	67285.714	34.286	1.429
	59490.000	780.000	n/a
	9.477	2.109	0.004
	151.429	206.667	0.476
	1360.000	490.000	10.000
	670.453	0.264	1.052
	16.364	0.455	0.182
	10.000	0.625	n/a
	5648.649	3.243	0.270
	962.500	1.500	n/a
Medians:	**816.48**	**2.68**	**0.48**

Semj-join and materialization together: **0.84**

Run parameters

- MySQL 6.0.3
- PostreSQL 8.3.0
- No tuning, all default settings
- Small query population
- => numbers only show order of magnitude

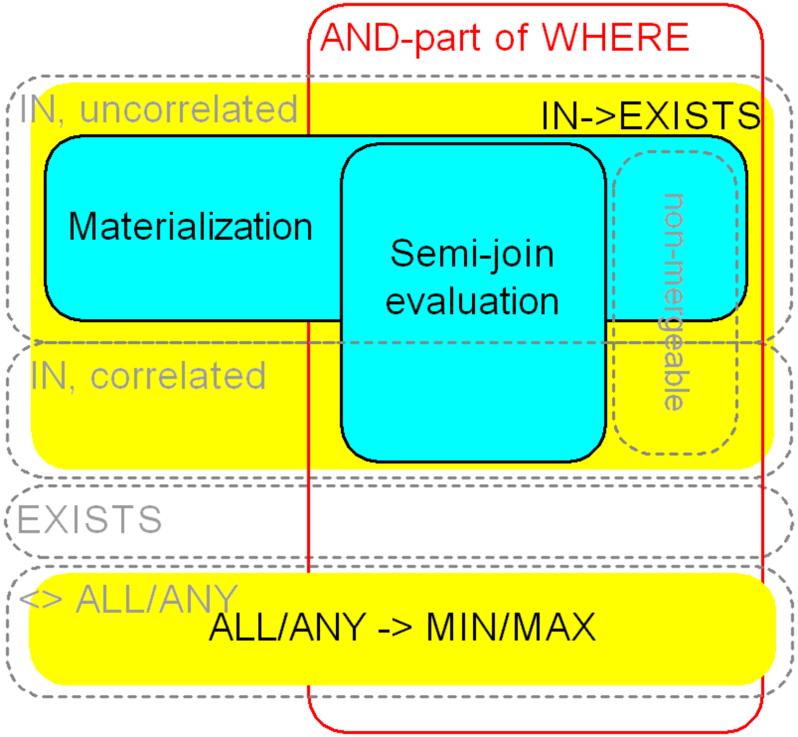
BTW, about PostgreSQL

- We compare against PostgreSQL often
- That's not because we have a goal to compete or outperform PostgreSQL
- It's just an
  - OSS DBMS
  - That is easy to use
  - Has a feature-rich optimizer
  - Does some things differently than MySQL
  - And some of us have expirience with
  - => Natural first choice but we'd like to compare with other databases too

Coverage of new optimizations

Subquery classification:

- Correlated/uncorrelated:
  - MySQL 4.1/5.x: correlate if possible
  - MySQL 6.0:
    - Flattening, FirstMatch, temptable – don't care
    - InsideOut, Materialization: uncorrelated only.
  - PostgreSQL:
    - Flattening (equvalent), Hash/Merge Join – seem to handle uncorrelated only



Coverage of new optimizations

Uncorrelated semi-join subquery classification

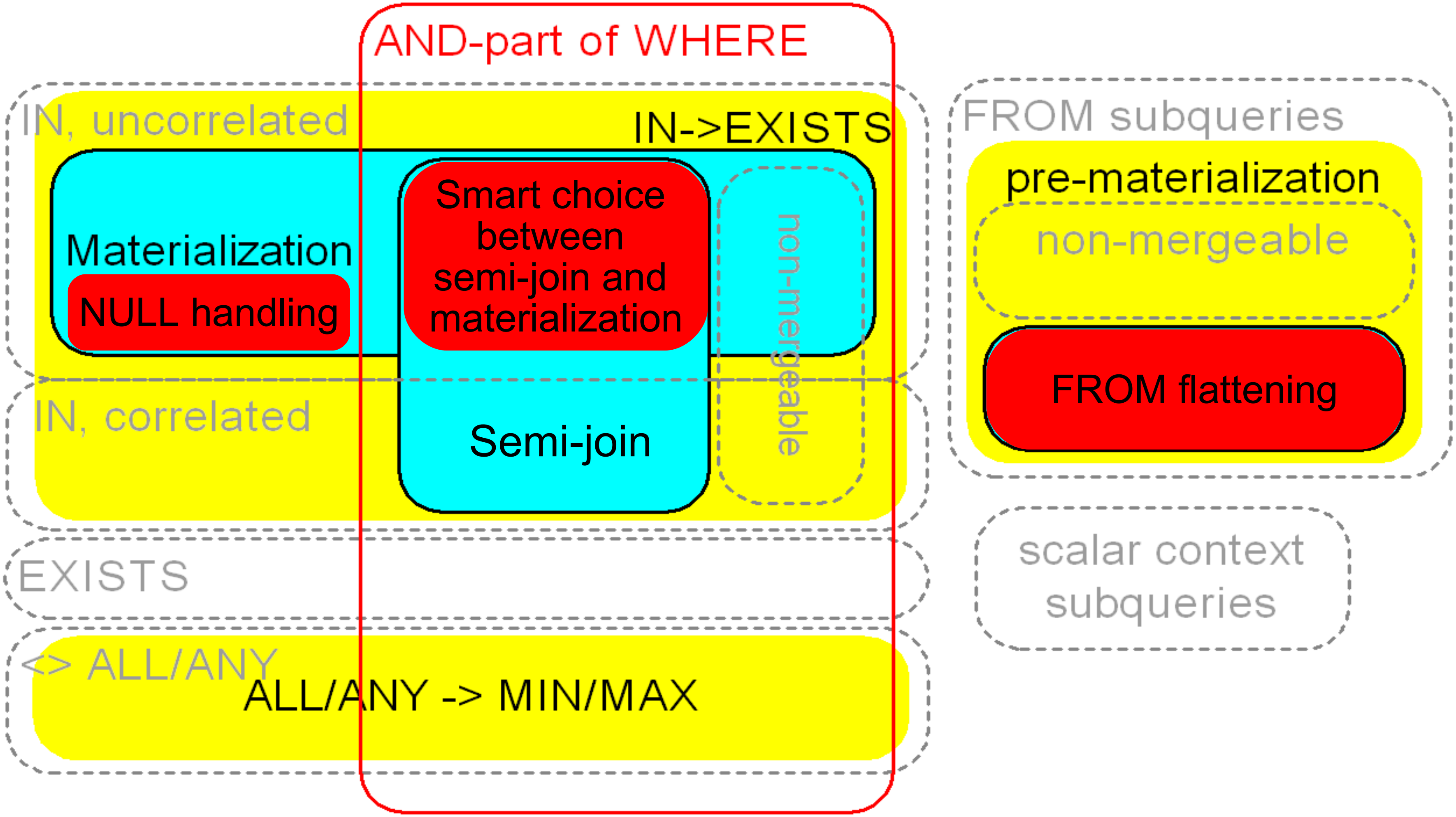
	MySQL 4/5.x	MySQL 6.0	PostgreSQL *
Conversion to inner join	Nothing	Flattening	Flattening
Outer-to-inner			
Lookup index available	IN->EXISTS (with all its limitations)	First Match	IN NL-Join
No usable index	Nested Loop join without buffering	Duplicate Elimination, Materialization	IN NL-Join, Hash join
Inner-to-outer			
Can use index to remove duplicates	Nothing	InsideOut	Merge join, Hash join
No index for duplicate elimination	Nothing	Duplicate Elimination, Materialization	Hash join variants, Sort+Unique

* based on our observations, may be incomplete

Future subquery work

- Doing now
  - Bug fixing
  - WL#3485 FROM subquery flattening
  - WL#3985 Smart choice between semi-join and materialization
  - WL#3830 Partial matching of tuples with NULL components: let materialization handle
    - `nullable_col IN (SELECT ...)`
    - `smth IN (SELECT nullable_expr ...)`
- Intend to do
  - SQL-level subquery hints
  - subquery_predicate_value(correlation_values) cache

Ongoing and future subquery work



WORK IN PROGRESS

FROM subquery flattening

Merge the FROM subquery into the upper join

```
SELECT ...
FROM (SELECT * FROM inner_tbl WHERE ...) tbl,
      ...
WHERE tbl.col='foo' AND ...
```

- Work in progress (done by Evgen Potemkin)
- Applicability conditions are same as for materialized VIEWs: subquery must
  - Not be a UNION
  - No GROUP BY or aggregates
  - No ORDER BY ... LIMIT or DISTINCT

WORK IN PROGRESS

Materialization and NULLs

```
.. outer_expr [NOT] IN (SELECT innercol
                         FROM inner_tbl WHERE ...)
```

NULL problems:

- On the left
  - NULL IN (SELECT something) = NULL
  - NULL IN (SELECT nothing) = FALSE
- On the right:
  - 'foo' IN (SELECT col FROM) = TRUE/FALSE

'bar'
'baz'
...

  - 'foo' IN (SELECT col FROM) = NULL

'bar'
'baz'
NULL

References

- 6.0 Subquery optimizations cheatsheet
  http://forge.mysql.com/wiki/6.0_Subquery_Optimization_Cheatsheet
- Technlical specs: Subquery optimizations: semijoin: WL#3985 and its subtasks
  http://forge.mysql.com/worklog/task.php?id=3985
- Technical specs: Subquery optimizations: materialization: WL#1110
  http://forge.mysql.com/worklog/task.php?id=1110
- MySQL 6.0 Subquery optimization benchmarks
  http://forge.mysql.com/wiki/6.0_Subquery_Optimization_Benchmarks
- Observations and news about subquery development
  http://s.petrunia.net/blog/

The end

Thank you

Q & A