Your queries rewritten - for you or by you?

Peter Vanroose

ABIS Training & Consulting

pvanroose@abis.be

Session Code: A3

Mon, Oct 02, 2017 15:10 | Platform: Db2 for z/OS

Abstract:
More than ever, Db2 for z/OS is being presented complex SQL queries. "Complex" does not just mean "many tables" but rather "containing many predicates", and "using newer SQL syntax" (e.g. ranking, CTEs, windowing).

Especially for those queries, Db2 is now often rewriting your SQL, e.g. reshuffle query blocks or add resp. optimize away predicates, to better optimize it.

If the end result turns out to perform badly, it's often not clear how to help the optimizer: add indexes, change cluster sequence, or just rewrite or restructure the query ourselves. Or help the optimizer deduce more precise filter factors.

In this presentation, we give several practical examples, thereby introducing and showing how to efficiently use some of the newer built-in Db2 tools, like the use of the EXPLAIN(ONLY) option of BIND, creating virtual indexes, and fine-tuning filter factors through predicate selectivity over-rides.

The presentation will also convince you that performing complex SQL is only possible when both DBAs and developers are tightly cooperating.

Objective 1: Understand when the Db2 optimizer might rewrite an SQL query, and how to find out if this happened

Objective 2: Know in which cases the simplistic performance tuning approaches don't work

Objective 3: Learn how to create virtual indexes, without the need of any fancy performance tooling

Objective 4: Understand the importance of filter factors, and how to help the optimizer in this respect

Objective 5: Learn to write readable and maintainable complex SQL queries, and to safely rewrite them for performance purposes

Dr. Peter Vanroose studied mathematics at the University of Leuven (Belgium), where he worked as a researcher for 20 years.
In 2004 Peter joined ABIS where he is teaching informatics courses (specialization: Db2 for z/OS), and is responsible for software development (COBOL, Perl, PHP) and database administration.

Peter's certificates include:

IBM Certified Database Administrator - Db2 11 for z/OS.

IBM Certified Solution Developer - Db2 9.5 SQL Procedure Developer.

Agenda

- Rewriting SQL queries
  - what? why? how?
  - some preliminary examples
- Optimizer rewrites: query transformations
  - How to detect?
    - Explain - a primer
    - Access path chronology
  - Query block reshuffling
- Taking advantage of this knowledge: manual query rewrites
- Query readability, performance, query rewrites, and explain

Two main parts:

- SQL queries rewritten by the optimizer
- SQL queries rewritten by you

=> actually, there's a tight connection between those two cases:
we get interesting ideas on how to rewrite a query
by looking at the ways the Db2 optimizer rewrites queries

=> this presentation gradually builds up some useful information about both topics
=> EXPLAIN will turn out to be of uttermost importance!
=> no need for fancy tools (although they can in handy, if you happen to have them...)

Query rewrites - definition & purpose

SQL (esp. SELECT statement) is a high-level language (4th generation)
=> descriptive, not imperative

Different SQL formulations may result in *identical output*

What ?

Query1 **is a rewrite of** Query2
if
guaranteed identical output of both, for *any* table content

Why ? possibly *more readable* and/or better *performance*

What is a "query rewrite"?

=> two queries (i.e., SELECT statements) returning the same result set

=> both queries are called **semantically equivalent**

Important to note:

should hold for **any** possible table content

(given the limitations of the table(s), e.g. uniqueness of a column, check constraint, nullability, foreign keys, ...)

=> The optimizer is allowed to use any table metadata (from the catalog) including not enforced constraints!

In practice, however, some straightforward query rewrites are never preformed

("historic legacy" which became "guarantees")

Notorious example:

```
WHERE    col1 = :HV + 0
```

is never rewritten to

```
WHERE    col1 = :HV
```

=> This opens possibilities for manual query rewrites, to help the optimizer find better performing access paths !

Query rewrites - preliminary example

```
SELECT    *       FROM products
WHERE     pr_spid IN  ( SELECT spid FROM suppliers )

SELECT    *       FROM products p
WHERE EXISTS  (  SELECT 1      FROM suppliers
                 WHERE         spid = p.pr_spid)

SELECT    DISTINCT p.*
FROM      products p   INNER JOIN   suppliers s   ON   p.pr_spid = s.spid
```

Table information:

two tables used in the first set of examples:

TABLE NAME	COLUMN NAME	CARDF	keys, indexes
PRODUCTS	PRCLASS	10000	PK(1); cluster
PRODUCTS	PRNO	100	PK(2); cluster
PRODUCTS	PRNAME	500000	UNIQ
PRODUCTS	PRSTATUS	5	
PRODUCTS	PRSTDATE	1000	
PRODUCTS	PRPRICE	50000	
PRODUCTS	PRDATEFROM	1000	
PRODUCTS	PRDATETO	1000	
PRODUCTS	PR_SPID	6000	FK (points to SUPPLIERS table)
SUPPLIERS	SPID	60000	PK
SUPPLIERS	SPNAME	60000	UNIQ; cluster
SUPPLIERS	SPSTREET	55000	
SUPPLIERS	SPSTRNO	100	
SUPPLIERS	SPTOWN	2000	
SUPPLIERS	SPTOWNNO	30000	
SUPPLIERS	SPCOUNTRY	50	
SUPPLIERS	SPTEL	50000	
SUPPLIERS	SPVAT	40000	
SUPPLIERS	SPBANKNO	50000	

Query rewrites - how?

- **Manual** rewrites:
  - starts by knowing different SQL formulations for the "same" query
  - be familiar with different ways to combine & filter data:
    - **predicates** (IN, EXISTS, ALL/ANY, ...) & combinations (AND/OR/NOT)
    - **query blocks**; subqueries; nested table expr.; common table expr. (CTE)
    - **joins** (inner, left, full; semi-join, anti-join)
- Rewrites by the **optimizer**:
  - know how to find out (e.g. Visual **Explain** with Data Studio)
  - understand what happened
  - distinction between query rewrite (= step 1) and optimization (= step 2)
- Optimizer rewrites can provide useful **ideas** for manual rewrites !

Important terminology / concepts:

Query block: single SELECT...FROM...WHERE; possibly as part of a larger SQL query; synonym: **fullselect**
Nested query: query block enclosed in parentheses, as non-leading part of an SQL statement (SELECT or UPDATE or INSERT or DELETE or MERGE)
=> can be placed in a WHERE clause (e.g. with IN or EXISTS), a FROM clause, or a SELECT clause
Subquery: nested query in a WHERE clause
Inner query / outer query: query block A belonging to the WHERE or FROM or SELECT clause of query block B: A is inner query of outer query B

Scalar fullselect: query block returning at most a single row & a single column
=> note that only scalar fullselects can be inner queries of a SELECT clause;
=> as part of a WHERE clause, a scalar fullselect can figure as LHS or RHS of a "simple comparison": = < > <= >= <>

Correlated inner query block (e.g. subquery): when column(s) from table(s) of outer query block(s) are referenced
Non-correlated query block: can be executed first, and result set plugged into its outer query, before executing the outer query

There are several good IDUG presentations on writing a certain query different ways (join, correlated/noncorrelated subqueries, CTE, ...)
=> see e.g. http://www.idug.org/p/bl/et/blogaid=493 (*Advanced SQL and the power of rewriting queries*, Tony Andrews, IDUG EMEA 2016)

Optimizer rewrites, a.k.a. query transformations

Db2 optimizer may rewrite query before choosing access path

Guaranteed identical output: based on information available to Db2 !

Typical optimizer rewrites include:

- adding DISTINCT to subquery for WHERE ... IN (SELECT **...** FROM ...)
- merging two **query blocks**
  - e.g.: subquery w. IN or EXISTS ==> JOIN
- COL1 = val1 OR COL1 = val2 ==> COL1 IN (val1,val2)
- pruning “always true” or “always false” predicate (with limitations!)
- *transitive closure* (adding predicates)
- *predicate pushdown* (moving predicate into subquery / UNION leg)

Query transformations by the optimizer:

- "conservative" approach: only some predefined set of conversions are considered
- 100% guarantee of identical output, for *any* possible / allowed table content
  - => considering data types and lengths of the columns
  - => typically *without* considering table constraints like NOT NULL; UNIQUE; foreign key; check constraint; before triggers; column cardinality

=> **manual rewrites** *could* go much further, by exploiting additional knowledge about the data

Examples:

```
PRSTATUS NOT IN ('1','2','3')          <==>     PRSTATUS IN ('0', '9') OR PRSTATUS IS NULL
                                       <==>     PRSTATUS IN ('0', '9')          -- because of NOT NULL

PR_SPID IN (SELECT SPID FROM suppliers) <==>    PR_SPID IS NOT NULL
```

Again, this opens possibilities for manual query rewrites, to help the optimizer find better performing access paths !

Optimizer rewrites: how to detect

Use **EXPLAIN**:

- embedded SQL: use (RE)BIND option EXPLAIN(YES) or **EXPLAIN(ONLY)**
- prepend query with

  EXPLAIN PLAN SET QUERYNO = *11111* FOR
- click on EXPLAIN icon in Data Studio

EXPLAIN describes the **access path** chosen by the Db2 optimizer

Analyze the response of EXPLAIN

=> if different from expected: *could* indicate a query rewrite

=> explain table DSN_QUERY_TABLE: *may* show rewritten query ...

DSN_QUERY_TABLE:

Contains full information about the query formalation, both the original one & the rewritten one
Information is stored as XML, in parsed form (query blocks, predicates, and their hierarchical connection)
Not easy to manually interpret:
- use 3rd party EXPLAIN tools
- or use at least an XML visualisation tool

EXPLAIN(ONLY):

Very useful for explaining static SQL embedded in an application program (e.g. COBOL), but **without** rebinding the package!
=> the authorization ID issuing a REBIND EXPLAIN(ONLY) only needs the **EXPLAIN authority**, not the SELECT authorities needed by the queries

Note that the explain information does **not** document the **current** access path in the package, but the new one, if a rebind would happen.

Explain - the basics

PLAN_TABLE

- contains one line per table (in FROM clause) and per *query block* (**QB**)
  => easy to detect *merging* or reshuffling / optimizing away of QBs
- example:
  SELECT * FROM products WHERE pr_spid IN (SELECT spid FROM suppliers)

  expected rows in PLAN_TABLE:

QB	Method	Table	AccessType	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype
1		products	I	ix_pr_spid	1	N		SELECT
2		suppliers	I	ix_spid	0	Y	U	NCOSUB

  observed rows in PLAN_TABLE:

QB	Method	Table	AccessType	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype
1		products	R			N		SELECT
1	1	suppliers	I	ix_spid	1	Y		SELECT

The compact output from PLAN_TABLE, as shown above, can easily be reproduced by using the following SQL query, even with SPUFI:

```
  SELECT      SUBSTR(DIGITS(QUERYNO), 7, 4) AS Qno,              SUBSTR(DIGITS(QBLOCKNO), 5, 1) AS QB,
              SUBSTR(DIGITS(METHOD), 5, 1) AS Mth,               SUBSTR(TNAME, 1, 8) AS Table,
              SUBSTR(ACCESSTYPE, 1, 2) AS AccTyp,                SUBSTR(ACCESSNAME, 1, 8) AS Index,
              CASE ACCESSNAME WHEN ' ' THEN ' ' ELSE SUBSTR(DIGITS(MATCHCOLS), 5, 1) END AS MatCol,
              CASE INDEXONLY WHEN 'Y' THEN 'Y' ELSE ' ' END AS IxOnly,
      ' ' ||  CASE SORTC_UNIQ        WHEN 'Y' THEN 'U' ELSE ' ' END ||
              CASE SORTC_ORDERBY     WHEN 'Y' THEN 'O' ELSE ' ' END ||
              CASE SORTC_GROUPBY     WHEN 'Y' THEN 'G' ELSE ' ' END ||
              CASE SORTC_JOIN        WHEN 'Y' THEN 'C' ELSE ' ' END ||
              CASE SORTN_JOIN        WHEN 'Y' THEN 'N' ELSE ' ' END    AS S_UOGCN,
              SUBSTR(PREFETCH, 1, 1) AS Pref,                    QBLOCK_TYPE AS QBtype,
              SUBSTR(TIMESTAMP, 7, 6) AS DDHHMM,                 SUBSTR(DIGITS(PARENT_QBLOCKNO), 5, 1) AS P_QB,
              CASE TABLE_TYPE
                   WHEN 'T' THEN 'TABLE'   WHEN 'S' THEN 'SUBQ'    WHEN 'C' THEN 'CTE' WHEN 'R' THEN 'CTE(r)' WHEN 'M' THEN 'MQT'
                   WHEN 'W' THEN 'WRKFIL' WHEN 'Q' THEN 'INTERM' WHEN 'B' THEN 'BUF' WHEN 'F' THEN 'FUNC'  ELSE ' '
              END AS Type_new
  FROM        PLAN_TABLE
ORDER BY      QUERYNO, BIND_TIME DESC, QBLOCKNO, PLANNO, MIXOPSEQ
```

Explain - more examples (1/3)

SELECT DISTINCT p.*
FROM products p INNER JOIN suppliers ON pr_spid = spid

expected rows in PLAN_TABLE:

QB	Method	Table	AccTyp	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype
1		products	R			N		SELECT
1	1	suppliers	I	ix_spid	1	Y		SELECT
1	3						U	SELECT

observed rows in PLAN_TABLE:

QB	Method	Table	AccTyp	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype
1		suppliers	I	ix_spid	0	Y		SELECT
1	2	products	I	ix_pr_spid	0	N		SELECT

(merge scan join instead of nested loop join)

(non-matching index access avoids sorting (for duplicate removal)

Method = 1 => nested loop join (this is the inner table; outer table has Method = 0)
Method = 2 => merge scan join (idem)
Method = 4 => hybrid join (idem)

Explain - more examples (2/3)

```
SELECT   * FROM products
WHERE  pr_spid IN  (   SELECT pr_spid FROM products
                      INTERSECT
                       SELECT spid FROM suppliers )
```

observed rows in PLAN_TABLE:

QB	Method	Table	AccTyp	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype	Parent
1		products	R			N		SELECT	0
2						N		INTERS	1
3		products	I	ix_prspid	0	Y		NCOSUB	2
4		suppliers	I	ix_spid	0	Y		NCOSUB	2

(more or less as expected, except maybe for the table scan ...)

More on EXPLAIN later

Relatively “new” SQL syntax:

Since Db2 9 we can use EXCEPT, EXCEPT ALL, INTERSECT, and INTERSECT ALL, syntactically similar to UNION & UNION ALL

INTERSECT: all rows from first QB that are *also* returned by second QB
EXCEPT: all rows from first QB that are *not* returned by seconf QB
(INTERSECT ALL & EXCEPT ALL are almost never useful...)

Explain - more examples (3/3)

Anti-join: give all suppliers without products:

SELECT * FROM suppliers s
WHERE NOT EXISTS (SELECT 1 FROM products WHERE pr_spid = s.spid)

rows in PLAN_TABLE:

QB	Method	Table	AccTyp	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype	Prefetch
1		suppliers	R		0	N		SELECT	S
2		products	I	ix_prspid	1	Y		CORSUB	

(most likely using *index look-aside* while repeatedly accessing index)

alternatives: NOT IN (with subquery), EXCEPT (inside subquery)
(both almost always *less performing*), or LEFT JOIN with pr_spid IS NULL

==> cf explain table **DSN_STATEMNT_TABLE** (col. TOTAL_COST)

Second explain table: DSN_STATEMNT_TABLE

The easiest way to compare manual query rewrites, in terms of performance, is by comparing the corresponding values in column TOTAL_COST

=> explain writes exactly one row in DSN_STATEMNT_TABLE, with the same value for QUERYNO as in PLAN_TABLE
=> per-statement summary information, essentially query **cost** estimation
=> the most important columns of DSN_STATEMNT_TABLE are the following ones:

PROCMS	estimated number of milliseconds (viz. the CPU cost of the query)
PROCSU	proportional to PROCMS, viz. number of "service units" (a measure which is independent of the processor speed)
TOTAL_COST	weighted sum of estimated CPU cost and estimated I/O cost; optimizer picks access path with lowest TOTAL_COST
COST_CATEGORY	either 'A' or 'B'; rough indication of reliability of estimated cost ('A': very reliable; 'B': less reliable)
REASON	when COST_CATEGORY is 'B': reason why the estimation is less reliable (e.g.: no statistics; intermed. tbl. card.; ...)

Query rewrites & chronology (1/2)

Query optimisation: often thanks to **early filtering**

- What is “early”?
  - optimizer writes out *access path*
  - Subsequent *steps* of access path matter: i.e., the **chronology**!

1. **Matching index access**: earliest possible filtering: avoid I/O
   => STAGE-1 filtering
2. **Index-only access**: just use the data in index
3. RID (pointer) list => possibly sort by pointer address: **list prefetch**
4. Further filter in tablespace
   => STAGE-2 filtering
5. Combine these *per-table* access paths => **QB order** & join order matter!

Explain & chronology

When reading the rows of PLAN_TABLE resulting from a single EXPLAIN in the **correct order**,
they reflect the **chronology** of the access path.

Within a single query block (QBLOCKNO):
==> ORDER BY PLANNO
This is essentially just **join** order (outer/inner tables)

Note that query block chronology is not easily deducible!
==> first the innermost non-correlated nested query block, up to the outer query, then from the outermost correlated subquery down
--> PLAN_TABLE column QBLOCK_TYPE: one of

- SELECT: outermost query block (or could be UPDATE, DELETE, INSERT, MERGE, or TRUNCA)
- SELUPD, DELCUR, UPDCUR, TRIGGR: similar, for specific variants ("for update", "where current of", "when")
- NCOSUB: non-correlated inner query block
- CORSUB: correlated inner query block
- UNION: formal "outer" query block for a UNION (no SELECT keyword); its sub-QBs are the "legs" of the UNION
  variants (similar): UNIONA, INTERS, INTERA, EXCEPT, EXCEPTA
- TABLEX: table expression, i.e., query block in a FROM clause
  CORTBLX: table expression with a "sideways reference"; often used with the XMLTABLE function
- PRUNED: query block without an access path because that QB will never be executed (always returns zero rows)

Query rewrites & chronology (2/2)

Query block chronology:

- not a choice of the optimizer: logically follows from query structure:
  - first the innermost non-correlated QB
  - then the correlated sub-QBs of that QB
  - etc., up to QB = 1 (the outer query block)
- Hence Query Rewrites (QB reshuffle) implicitly decide on chronology!

Table join chronology

- decided by optimizer, *after* query rewrite

Predicate (i.e. filter) chronology

- decided by optimizer, *after* query rewrite

Query access path steps & their chronology:

- chronology of query blocks
- within a query block:
    - chronology of tables in a join
    - chronology of filter predicates (WHERE conditions)

Note that some steps may be (and sometimes are) performed **in parallel** => see columns JOIN_DEGREE & ACCESS_DEGREE of PLAN_TABLE

Predicate pushdown (1/3)

What: query rewrite, moves a predicate from outer QB to inner QB

- Example:

  Original query:

```
WITH sp AS  (   SELECT  spid, spname FROM suppliers          | QB2
                WHERE  spcountry = :HV                   )
SELECT      prname, spname                                   |
FROM        products JOIN sp ON pr_spid = spid               | QB1
WHERE       pr_spid < 1000                                   |
```

  Rewritten query: predicate from QB1 pushed down into QB2:

```
WITH sp AS ( SELECT  spid, spname FROM suppliers
             WHERE  spcountry = :HV  AND  spid < 1000 )
SELECT  prname, spname FROM products JOIN sp ON pr_spid = spid
```

Non-correlated query blocks are evaluated **before** their parent QB

=> moving a predicate from parent to subquery / nested query / CTE gives **earlier filtering** => performs better!

Predicate pushdown (2/3)

Special case: pushdown into **legs of UNION** (or EXCEPT or INTERSECT)

```
WITH p(name,street,town,country) AS
        (   SELECT spname, spstreet, sptown, spcountry        FROM suppliers
        UNION ALL
            SELECT whname, whstreet, whtown, whcountry        FROM warehouses
        )
SELECT      *  FROM p   WHERE country = :HV
```

=> Early filtering! (May avoid excessive *materialisation* further on)

```
    SELECT   spname AS name, spstreet AS street, sptown AS town, spcountry AS country
    FROM     suppliers    WHERE spcountry = :HV
UNION ALL
    SELECT   whname AS name, whstreet AS street, whtown AS town, whcountry AS country
    FROM     warehouses    WHERE whcountry = :HV
```

=> especially useful when UNION is in definition of a **VIEW**

Table information:

extra table, used in this example:

TABLE NAME	COLUMN NAME	CARDF	keys, indexes
WAREHOUSES	WHID	100	PK
WAREHOUSES	WHNAME	100	unique
WAREHOUSES	WHSTREET		
WAREHOUSES	WHSTRNO		
WAREHOUSES	WHTOWN	80	cluster
WAREHOUSES	WHTOWNNO	90	
WAREHOUSES	WHCOUNTR	5	
WAREHOUSES	WHSTATUS	7	
WAREHOUSES	WHCAPACITY	20	

Predicate pushdown (3/3)

The good news:

- predicate pushdown is *query transformation*: need not do it manually
- is also applied on VIEWs (which cannot be rewritten manually)

Query readability:

- don't be afraid of writing query in a modular way = **use CTEs** !
- write out the filtering in the "most logical" place (=as early as possible)

How to detect predicate pushdown? => EXPLAIN:

QB	Method	Table	AccTyp	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype
1		P	R			N		SELECT
2						N		UNIONA
3		suppliers	I	ix_spcountr	1	N		NCOSUB
4		warehouses	I	ix_whcountr	1	N		NCOSUB

Explain - detecting materialisation

Materialisation:

- whenever Db2 must store intermediate result (temp table / work file)
- **only** necessary when "streaming" is impossible

  => because some kind of **sorting** is required:
  - for ORDER BY, for GROUP BY, or for DISTINCT ("unique")
  - for pre-sorting to prepare a merge-scan join (outer/inner table, or both)
  - [for "*list prefetch*" of index] ("L" in column "prefetch")

SELECT * FROM products, suppliers WHERE pr_spid = spid ORDER BY prno

QB	Method	Table	AccTyp	IxName	MatCol	IxOnly	Sort_UOG_CN	Prefetch	QBtype
1		products	R			N		S	SELECT
1	2	suppliers	R			N	CN	S	SELECT
1	3						O		SELECT

There are essentially only 4 reasons for sorting (and hence materialisation) of table data;
this is reflected in 5 columns of PLAN_TABLE with “Y” (yes; no sorting for that reason is indicated by “N”):

SORTC_UNIQ	sorting before removing duplicates (“distinct” a.k.a. “unique”)
SORTC_ORDERBY	just sorting because the user asks for it (“ORDER BY ...”)
SORTC_GROUPBY	sorting before aggregation (viz. for a GROUP BY clause)

the forth reason is purely a decision of the optimizer: just before starting a JOIN activity, the two tables involved in the join could be pre-sorted:

SORTC_JOIN	the “composite” or “outer” (or first) table is sorted
SORTN_JOIN	the “new” or “inner” (or second) table is sorted

Important to note: both indications are placed on the PLAN_TABLE line for the *inner* table, i.e., the line containing the non-zero METHOD value.
This means that SORTC_JOIN = ‘Y’ does *not* refer to the table mentioned on that line (in contrast to the other 4 SORT* columns) !

Query block merging (1/3)

A "simple" nested QB (or VIEW definition or Common Table Expression) may be automatically merged into the parent QB

- "simple": just containing SELECT (= projection), WHERE, JOIN
- Example:

```
SELECT   prname, spname
FROM     (SELECT      pr_spid, prname
          FROM        products JOIN stocks ON prclass=st_prclass AND prno=st_prno
          WHERE       stquantity > ?)  p_s
         JOIN   suppliers ON spid = pr_spid    WHERE   spcountry = ?
```

make that a single QB with a 3-table join; keep all WHERE predicates:

```
SELECT prname, spname FROM products JOIN stocks ON ...  JOIN suppliers ON spid=pr_spid
WHERE  stquantity > ?  AND  spcountry = ?
```

Table information:

extra table, used in this example:

TABLE NAME	COLUMN NAME	CARDF	keys, indexes	
STOCKS	ST_PRCLASS	10000	FK (points to PRODUCTS table)	PK
STOCKS	ST_PRNO	100	"	"
STOCKS	ST_WHID	100	FK (points to WAREHOUSES table)	"
STOCKS	STQUANTITY	500		
STOCKS	STSTATUS	500		
STOCKS	STDATE	1500		

Rewritten query:

```
SELECT   prname, spname
FROM     productsJOIN stocks ON prclass=st_prclass AND prno=st_prno
                 JOIN suppliers ON spid = pr_spid
WHERE    stquantity > ?
```

Query block merging (2/3)

More complex QB merging (NTE, CTE, or VIEW):

- when the NTE/CTE contains a GROUP BY or DISTINCT
- Example:

```
WITH p_s AS ( SELECT    pr_spid, prname, SUM(stquantity) AS prquantity
              FROM      products, stocks WHERE prclass=st_prclass AND prno=st_prno
              GROUP BY  prclass, prno, pr_spid, prname)
SELECT  prname, spname, prquantity      FROM  p_s  JOIN  suppliers ON spid = pr_spid
```

=> becomes a single QB with 3-table join & GROUP BY

(and note the presence of an additional `spname` in the GROUP BY !)

```
SELECT   prname, spname, SUM(stquantity) AS prquantity
FROM     products JOIN stocks ON ... JOIN suppliers ON ...
GROUP BY    prclass, prno, pr_spid, spname, prname
```

Rewritten query:

```
SELECT   prname, spname, SUM(stquantity) AS prquantity
FROM     productsJOIN stocks ON prclass=st_prclass AND prno=st_prno
                   JOIN suppliers ON spid = pr_spid
GROUP BY prclass, prno, pr_spid, prname, spname
```

Note that a WHERE predicate on prquantity in the outer QB would become a HAVING predicate !

Query block merging (3/3)

QB merging with UNION (ALL):

- replace JOIN with CTE containing UNION, by UNION of 2x the JOIN

```
    WITH     datasets(spacename, dbname, dstype) AS
             (SELECT name, dbname, 'TS' FROM sysibm.systablespace   WHERE creator = ?
             UNION ALL
              SELECT indexspace, dbname, 'IX' FROM sysibm.sysindexes WHERE creator = ?)
    SELECT   dbname, spacename, type, implicit, dstype
FROM sysibm.sysdatabase db JOIN datasets ds ON db.dbname = ds.dbname
```

No automatic QB (NTE/CTE/VIEW) merging:

- when the NTE/CTE is “too complex”, e.g. has a materializing ORDER BY

=> consider *manual rewrite* if necessary (performance <--> readability)

The query could be rewritten by Db2 into:

```
      SELECT    ds.dbname, ds.name AS spacename, db.type, db.implicit, 'TS' AS dstype
      FROM      sysibm.sysdatabase db JOIN sysibm.systablespace ds ON db.dbname = ds.dbname
      WHERE     ds.creator = ?
UNION ALL
      SELECT    ds.dbname, ds.name AS spacename, db.type, db.implicit, 'IX' AS dstype
      FROM      sysibm.sysdatabase db JOIN sysibm.sysindexes ds ON db.dbname = ds.dbname
      WHERE     ds.creator = ?
```

with of course “early filtering” (before the respective joins) for the predicate ds.creator = ?

More query block magic (1/3)

IN to JOIN

- classical case: IN with subquery (but ... may generate duplicates)

```
SELECT * FROM p                                      =>    SELECT p.* FROM p JOIN q ON p.c = q.x
WHERE c IN (SELECT x FROM q WHERE ...)                     WHERE ...
```

- also may happen automatically, even for IN with explicit list
  => creation of temporary (auxiliary) table for IN list (= materialisation)

```
SELECT * FROM products WHERE prclass IN (?,?,?,?) AND prstatus IN (?,?)
```

=> auxiliary (in-memory) 2-column table with 8 rows

Variant: NOT EXISTS or NOT IN subquery => anti-join:

```
SELECT * FROM p                                  =>    SELECT p.* FROM p LEFT JOIN q ON p.c = q.x
WHERE NOT EXISTS (SELECT 1 FROM q                      WHERE q.x IS NULL
                  WHERE x = p.c    )
```

An “IN” predicate with non-correlated subquery can always be converted into an inner join,
although a DISTINCT will be needed unless the table structures (e.g. primary key) guarantee that this is not necessary

The other way around, a JOIN of tables p and q can only be converted into a outer query on p, with an IN predicate and subquery on q,
when the original JOIN query had no columns from table q in one of its clauses (e.g. the SELECT clause).

More query block magic (2/3)

OR expansion

- rewrite a QB with an OR predicate into a UNION ALL
  - is never an automatic query rewrite
  - careful: don’t forget to add a negative AND predicate to one of the QB !
- variant: *multi-index* access path:

SELECT * FROM products
WHERE prname LIKE ? OR prclass = ?

QB	Method	Table	AccTyp	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	Prefetch
1		products	M			N		L
1		products	MX	ix_prid	1	Y		
1		products	MX	ix_prname	1	Y		
1		products	MU			N		

=> 2 indexes used; list of RID pointers is “UNION”ed (“M**U**”): list prefetch!

OR into UNION ALL

- actually, rewriting into UNION could be done blindly (but that would almost never yield a better performing access path...)
- manual rewrite into UNION ALL (thus avoiding additional sort to remove duplicates):
  - => only makes sense when (1) there certainly are no duplicates, or (2) duplicates don't matter further on in the processing
  - => alternatively, remove the duplicates by adding an extra WHERE predicate to one of the two legs of the UNION ALL
    This extra predicate is the negation of the full predicate of the other leg

Example:

```
    SELECT *
    FROM   products
    WHERE prname like 'A%'  OR  prclass = '7'
```

=>

```
    SELECT *
    FROM   products
    WHERE prname = '7'
UNION ALL
    SELECT *
    FROM   products
    WHERE prname like 'A%'   AND  prclass <> '7'
```

More query block magic (3/3)

Two special cases: **MQTs** and **temporal tables**

- Suppose data warehouse has materialised query table (MQT):

  CREATE TABLE pr AS (SELECT pr_spid, COUNT(*) AS cnt FROM products GROUP BY pr_spid)

  The following 2-QB query *could* be simplified by the optimizer to 1 QB:

  WITH x AS (SELECT COUNT(*) AS nr_prods, pr_spid FROM products GROUP BY pr_spid)
  SELECT spname, nr_prods FROM suppliers JOIN x ON spid = pr_spid

converted to: SELECT spname, cnt AS nr_prods FROM suppliers JOIN pr ON spid = pr_spid

- Suppose products is a system-period temporal table

  SELECT prname, prstatus FROM products AS OF TIMESTAMP '2017-01-01 00:00:00'

to: SELECT prname, prstatus FROM products WHERE valid_from >= '2017-01-01 00:00:00'
UNION ALL
SELECT prname, prstatus FROM prod_hist WHERE '2017-01-01 00:00:00' BETWEEN ...

Temporal tables:

If you are not familiar with the concept: see e.g. http://www.abis.be/resources/presentations/gsebedb220130606temporaldata.pdf

Simply stated: tables that remember their *past state*, hence can be queried for their content *at a certain given time instant*.

MQTs:

Materialized Query Tables.

=> logically speaking, they are *views*, but technically they are *pre-materialized*, i.e., stored in their proper tablespace

=> only useful for slowly changing table content (e.g. Data Warehouses with once per day refresh)

Predicate rewrites

Rewrite of a WHERE clause:

- query transformations: more difficult to find out
- manual rewrites: very important tool to influence performance !

Preliminary examples of auto-rewrites:

WHERE prstatus IN (?) ==> WHERE prstatus = ?

prstatus = ? OR prstatus = ? ==> prstatus IN (?,?)

prstatus IN (?,?) OR prstatus = ? ==> prstatus IN (?,?,?) -- less evident!

prdate > ? OR (prdate = ? AND prclass > ?) (prdate,prclass) > (?,?) -- ???

(very useful in "repositioning", e.g. for restartability & also for paging!)

Elaboration on the last example: “repositioning”

prdate	prclass
03/01	**1**
03/01	**2**
03/01	**3**
03/02	**1**
03/02	**3**
03/04	**2**
03/04	3
03/05	4
03/05	7
03/10	2
03/10	4

Suppose the first 6 entries have already been processed. (E.g.: previous screen of an interactive “paging” application, or a restartable batch application that crashed halfway.)

How to select the *same set* of rows except for the ones already processed?

=> make sure (also the first time) to present the rows in a well-defined order (here: ORDER BY prdate, prclass)

=> for the repositioning: use the original predicates, plus the following one:

```
(PRDATE, PRCLASS) > ('2017-03-04', 2)                    --  note: this is Db2 12 syntax
```

or equivalently (see the example to understand why this is equivalent):

```
PRDATE > '2017-03-04'   OR   (PRDATE = '2017-03-04' AND PRNO > 2)
```

Simple predicate manipulations: contraposition

Negation of an AND is an OR, and vice versa! (De Morgan's rules)

Examples:

- ... WHERE prname NOT LIKE 'A%' OR prname NOT LIKE 'B%'

  is equivalent to

  ... WHERE NOT (prname LIKE 'A%' AND prname LIKE 'B%')

  which is nonsense (actually: always true)

- ... WHERE prstatus <> '1' AND (prclass IN (?,?) OR prno > ?)

  ... WHERE NOT (prstatus = '1' OR prclass NOT IN (?,?) AND prno <= ?)

- ... WHERE NOT (pr_spid < **ANY** (SELECT spid FROM suppliers WHERE ...))

  ... WHERE pr_spid >= **ALL** (SELECT spid FROM suppliers WHERE ...)

Contraposition of AND and OR

The so-called laws of De Morgan (see e.g. https://en.wikipedia.org/wiki/De_Morgan%27s_laws) are fairly simple but very useful:

if P and Q are two predicates, then the following two are always logically equivalent:

NOT (P AND Q) NOT P OR NOT Q

as are the following two:

NOT (P OR Q) NOT P AND NOT Q

Extension to ALL and ANY:

ALL (subquery) is a repeated AND,
ANY (subquery) is a repeated OR

Extension to IN & NOT IN:

a IN (subquery) is equivalent to a = ANY (subquery), which is a repeated OR

Extension to EXISTS:

EXISTS (subquery) is essentially a repeared OR

Simple predicate manipulations: transitive closure

Adding “superfluous” predicate(s), consequence of 2 other ones

Examples:

- ... FROM products JOIN suppliers ON pr_spid=spid WHERE spid = ?

  add the following:

  ... AND pr_spid = ? -- *important for early filtering ! (only with inner join)*

- a = b AND b = c => a = c
- a <= b AND b < c => a < c -- essentially: triangle inequality
- a IN (?,?) AND b = a => b IN (?,?) -- or any other condition on a
- a BETWEEN x AND y AND b < a => b < y
- ... a < 20 AND a IN (SELECT x FROM p WHERE ...)=> add “AND x < 20”

Transitive closure

- is an important standard technique, applied by the optimizer during the “query rewrite” phase
- especially important across tables and (if possible) across query blocks

Simple predicate manipulations: stage-2 to stage-1

Rewrite stage-2 predicates (non-indexable) to stage-1

Examples:

- ... WHERE prdate + 7 days = ? => prdate = ? - 7 days
- ... WHERE prprice * 1.21 > 1000 => prprice > 1000 / 1.21
  => be careful with "*" and "/" (floating point arithmetic)
- ... WHERE substr(sptown, 1, 1) = 'A' => sptown LIKE 'A%'
- ... WHERE upper(whtown) = 'LISBOA' => whtown IN ('Lisboa', 'lisboa', 'LISBOA')
- ... WHERE prdate + 1 MONTH = current date => ???
- ... WHERE year(prdate) = 2017 => prdate BETWEEN ...

Avoiding stage-2 predicates

Any scalar function call (including arithmetic + - * / and text concat || operator) are “stage 2”, meaning that
- the data manager (first stage of data access) cannot apply this filtering => postponed until “stage 2”, i.e., the RDS component
or better said:
- the optimizer cannot delegate this part of the filtering to the data manager,
especially:
- the optimizer cannot use matching index access (and thus: avoid data access) for implementing this predicate

On the other hand, all “range predicates” (= < > <= >= BETWEEN LIKE) can be implemented through matching index access

MANUAL QUERY REWRITE:
    avoid stage-2 predicates where possible => a **quick win** !

Currently the only “automatic” query rewrites in this family are:

    YEAR(c) = ? => c BETWEEN ... AND ...
    SUBSTR(c,1,n) = ... => c LIKE '...%'

Additionally, the optimizer now has the possibility to apply “early stage-2 filtering”, e.g. during index screening

stage-2 and explain

- Stage-2 predicate with substantial filtering & **index** available:

  SELECT * FROM products WHERE prclass BETWEEN '3000 ' AND '3999 '

  ==> actually becomes: prclass || ' ' BETWEEN '3000 ' AND '3999 '

QB	Method	Table	AccessType	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype
1		products	R			N		SELECT

- same query, manually rewritten to be stage-1:

  SELECT * FROM products WHERE prclass BETWEEN '3000' AND '3999'

QB	Method	Table	AccessType	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype
1		products	I	ix_prid	1	N		SELECT

- but careful with e.g.: (not the same access path !!)

  SELECT * FROM products WHERE prclass LIKE '__10' ORDER BY PRCLASS

  SELECT * FROM products WHERE SUBSTR(prclass, 3, 2) = '10' ORDER BY PRCLASS

QB	Method	Table	AccessType	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	QBtype
1		products	I	ix_prid	0	N		SELECT

stage-2 and EXPLAIN:
Looking at the most important columns of PLAN_TABLE often indicates the presence of a stage-2 predicate:
table scan instead of expected matching index access

Actually, a stage-1 but non-indexable predicate could also yield a table scan;
on the other hand, an ORDER BY on the cluster column could trigger a non-matching index access, even with a stage-2 predicate ...

To understand the access path chosen by Db2 (and the consequences for performance) one has to look at the **full picture**!

stage-2 versus non-indexable

- both are equally "bad" (early, matching index filtering not considered)
- but... stage-1 **non-indexable** is applied earlier (by data manager)
- non-indexable stage 1 predicates are all **non-range** predicates:
  - "negatives": e.g. NOT BETWEEN, <>, IS NOT NULL
  - LIKE starting with wildcard, e.g. col LIKE '%A' col LIKE '__A%'
- useful information in **DSN_FILTER_TABLE**
  - contains 1 row per *predicate*
  - mentions predicate "stage" (matching/screening/stage1/stage2), and chronology of evaluation of each predicate (column "orderno")
- look in **DSN_PREDICAT_TABLE**, column **TEXT** to identify the predicate
  - note: "*early stage-2 filtering*" since Db2 10 => no "black/white" story...

Useful query on combined PLAN_TABLE and DSN_PREDICAT_TABLE:

```
SELECT p.QUERYNO AS Q, p.QBLOCKNO AS QB, p.METHOD, rtrim(TNAME) || '(' || TABLE_TYPE || ')' AS TABLE,
       p.ACCESSTYPE AS A, p.MATCHCOLS AS MC, p.ACCESSNAME AS idx, c.FILTER_FACTOR AS FF, f.STAGE, f.ORDERNO,
       CASE c.AFTER_JOIN WHEN 'A' THEN 'after join' WHEN 'D' THEN 'during join' ELSE ' ' END AS when,
       CASE c.SEARCHARG WHEN 'Y' THEN 'stage-1' ELSE 'stage-2' END AS stage,
       c.TEXT AS predicate
FROM  plan_table p
      LEFT JOIN dsn_filter_table f ON p.queryno = f.queryno AND p.qblockno = f.qblockno AND p.progname = f.progname AND p.collid = f.collid
                                      AND p.bind_time = f.explain_time AND p.planno = f.planno
       LEFT JOIN dsn_predicat_table c ON f.predno = c.predno AND p.queryno = c.queryno AND p.qblockno = c.qblockno
                                      AND p.progname = c.progname AND p.bind_time = c.explain_time
WHERE p.BIND_TIME > CURRENT TIMESTAMP - 10 minutes        --  i.e., from a recent EXPLAIN
ORDER BY p.TIMESTAMP DESC, 1, 2, p.PLANNO, p.MIXOPSEQ, f.orderno
```

predicate information with EXPLAIN

Example:

```
SELECT *    FROM products LEFT JOIN suppliers ON pr_spid = spid
WHERE  prname NOT LIKE 'De%'    -- 4
  AND  prstdate > '1992-12-31'  -- 3
  AND  UPPER(sptown) = ?        -- stage-2 !
  AND  pr_spid = '08012'        -- 1 (matching index access, thus avoiding I/O)
  AND  prstatus = '9'           -- 2
  AND  prno LIKE '_9'           -- 5
```

Mth	Table	AcTyp	Idx	FilterFactor	Stage	OrderNo	When	Predicate
0	suppliers(T)	I	spid	1.666665e-5	MATCHING	1		SPID='08012'
0	suppliers(T)	I	spid	4.999998e-4	STAGE2	2		UCASE(SPTOWN)=(EXPR)
1	products(T)	I	prstatus	4.506759e-5	MATCHING	1		PR_SPID='08012'
1	products(T)	I	prstatus	3.000000e-1	STAGE1	2		PRSTATUS='9'
1	products(T)	I	prstatus	8.415346e-1	STAGE1	3		PRSTDATE>'1992-12-31'
1	products(T)	I	prstatus	9.999980e-1	STAGE1	4		PRNAME NOT LIKE 'De%'
1	products(T)	I	prstatus	1.000000e-1	STAGE1	5		PRNO LIKE '_9'

Most important info from EXPLAIN table DSN_PREDICAT_TABLE:

"Stage" (matching / stage1 / stage2)

"Filter Factor" (= predicate selectivity)

=> See http://www.idug.org/p/bl/ar/blogaid=568 by Joe Geller (IDUG EMEA 2016) for a good overview on Filter Factors

Filter Factor fine-tuning (1/2)

estimated filter factor of a predicate is crucial for optimizer !

- based on catalog statistics (RUNSTATS),
  esp. (colcardf, high2key, low2key) of SYSIBM.SYSCOLUMNS
  - is accurate for “=”, “<=” etc, BETWEEN, LIKE ‘A%’ etc, IN predicates *provided* that data is not “skew”
  - is only accurate for “=” and IN, with *host variables*
- also based on freq. statistics in SYSIBM.SYSCOLDIST (if available)

manual query rewrite can modify filter factors, if we know better:

... WHERE stquantity <= ?   (FF=0.33333)   => stquantity BETWEEN 0 AND ? (FF=0.1)
... WHERE sptown = ?   (FF=0.000556)   => sptown BETWEEN ? AND ?   (FF=0.1)
... WHERE prstatus = ?   (FF=0.2)   => prstatus = ? AND prstatus <= ?   (FF=0.06667)

(assuming **static SQL**)

Filter factors

The optimizer calculates filter factors as follows, given that a standard RUSTATS has been run on all tables:

predicate	FF	
col = val	1/colcardf	
col IN (list)	list-size / colcardf	
col <= val	(val - Low2key) / (High2key - Low2key)	(similar for < > >=)
col <= ?	1/3	
col BETWEEN v1 AND v2	(v2 - v1) / (High2key - Low2key)	
col BETWEEN ? AND ?	0.1	
col LIKE ‘lit%’	as for the equivalent BETWEEN	
pred1 AND pred2	FF1 * FF2	
pred1 OR pred2	FF1 + FF2 - FF1 * FF2	
NOT (pred1)	1 - FF1	

Filter Factor fine-tuning (2/2)

- Manual query rewrite:
  - is the preferred way to “hint” the optimizer!
  - clearly document why you write the predicates in such a “strange” way!
- “External” way (without having to modify the query):
  - “freeze” a certain access path of your choice
    - this is a **bad idea** (later statistics changes & new indexes: ignored)
    - very cumbersome: need to run EXPLAIN, then modify PLAN_TABLE, then BIND with option OPTHINT
    - hint is lost at next REBIND, or when PLAN_TABLE is emptied ...
  - since Db2 11: use BIND QUERY cmd & tbl DSN_PREDICATE_SELECTIVITY

    see e.g. http://www.idug.org/p/bl/et/blogaid=366

Filter factor tuning:

preferably use the “new” **predicate selectivity override** mechanism

=> cf. other IDUG presentations, on-line blogs, or the IBM manuals on the BIND QUERY command and the DSN_PREDICATE_SELECTIVITY table

Virtual indexes

- Query rewrites may cause different access paths=>*better performing?*
- Some (future) access paths are not yet available => how to test ?
  - e.g. because an index does not yet exist,
  - or because a table has the "wrong" cluster sequence,
  - or because an existing index needs extra columns
- Solution: create a **virtual index**, next run EXPLAIN
  - e.g.   REBIND PACKAGE(a.b) EXPLAIN(ONLY)
- Create: insert meta-data into explain table **dsn_virtual_indexes**

```
INSERT INTO dsn_virtual_indexes (tbcreator, tbname, ixcreator, ixname) VALUES (..., ...) ;
UPDATE dsn_virtual_indexes SET enable='Y', mode='C', colcount=2, clustering='N',
    uniquerule='D', pgsize=4, padded='N', indextype='D', colno1=3, ordering1='A',
    colno2=7, ordering2='D', nleaf=423, nlevels=3, firstkeycardf=1000,  clusterratiof=0.9
```

Virtual indexes

- Once created (one line in your table DSN_VIRTUAL_INDEXES), can easily be disabled by setting column ENABLE to ‘N’
- Existing (real) indexes can be virtually dropped by inserting a row with MODE=’D’ (instead of ‘C’)
- Most other fields are those of SYSIBM.SYSINDEXES, including RUNSTATS information
- Information from SYSIBM.SYSKEYS (for real indexes) is to be stored in columns COLNO1 (etc) and ORDERING1 (etc)

Virtual Indexes and Query Rewrites

- dsn_virtual_indexes, column mode: “C” = create, “D” = delete
  => use “D” to disable an existing index
- actually, there is an easier way to *disable an index* (permanently):

```
SELECT *     FROM products LEFT JOIN suppliers ON pr_spid = spid
WHERE  prname NOT LIKE ?     -- 3
  AND  prstdate > ?          -- 2
  AND  prstatus = ?          -- 1     matching index access     => I don't want this
  AND  prno LIKE ?           -- 4                                  (real filter factor too high)
```

rewrite as:

```
SELECT *     FROM products LEFT JOIN suppliers ON pr_spid = spid
WHERE  prname NOT LIKE ?     -- 3
  AND  prstdate > ?          -- 1
  AND  prstatus = ? || ''    -- 2     => index use is disabled     => 2nd choice is taken
  AND  prno LIKE ?           -- 4
```

Two documented "tricks" to disable matching index access on a predicate:

- for numeric column:	replace	COL <=> value	by	COL <=> value + 0
- for text column:	replace	COL <=> value	by	COL <=> value || ''

Not so common SQL syntactic constructions (1/3)

Some SQL constructs are not so well-known

but may improve query *readability* and/or query *performance* !

Example: *give per supplier the number of products he is responsible for*:

```
SELECT  spname, sptown, COUNT(*) AS cnt  FROM suppliers JOIN products ON spid=pr_spid
GROUP BY spid, spname, sptown       -- too many grouping cols: a bit cumbersome ...
```

=> dsn_statemnt_table.procsu = **89301** (needs sorting for both GROUP BY & JOIN)

```
WITH  prod_count AS (SELECT pr_spid, COUNT(*) AS cnt FROM products GROUP BY pr_spid)
SELECT spname, sptown, cnt  FROM suppliers JOIN prod_count ON spid = pr_spid
```

=> dsn_statemnt_table.procsu = **20964** (cost_category='B': "table cardinality")

```
SELECT   spname, sptown, (SELECT COUNT(*) FROM products WHERE pr_spid = s.spid) AS cnt
FROM     suppliers s                  -- correlated subquery in the SELECT clause !
```

=> dsn_statemnt_table.procsu = **12212**

Access paths for these three queries:

QB	method	table	actyp	index	mcol	IXONLY	S_UOGCN	prefetch	QBtype
1	0	suppliers	R					S	SELECT
1	1	products	I	ix_PRSPID	1	Y	C		SELECT
1	3						G		SELECT
1	0	prod_count	R					S	SELECT
1	4	suppliers	I	ix_SPID	1		N	L	SELECT
2	0	products	I	ix_PRSPID	0	Y		S	TABLEX
1	0	suppliers	R					S	SELECT
2	0	products	I	ix_PRSPID	1	Y			CORSUB

Not so common SQL syntactic constructions (2/3)

Ranking and filtering.

Example: *give the 10 "most busy" suppliers (most products)*:

```
SELECT   spname, sptown, (SELECT COUNT(*) FROM products WHERE pr_spid = s.spid) AS cnt
FROM     suppliers s
ORDER BY cnt DESC   FETCH FIRST 10 ROWS ONLY
```

=> not a fair ranking: maybe the 11th one has the same number of products ...

```
WITH     sp AS
(SELECT  spname, sptown, (SELECT COUNT(*) FROM products WHERE pr_spid = s.spid) AS cnt
 FROM    suppliers s),
         ranked_sp AS (SELECT sp.*, RANK() OVER (ORDER BY cnt DESC) AS rnk FROM sp)
SELECT * FROM ranked_sp WHERE rnk <= 10
```

=> may return more than 10 rows; PROCSU = 9511 while first query only had 6666

Access paths for these two queries:

QB	method	table	actyp	index	mcol	IXONLY	S_UOGCN	prefetch	QBTYPE
1	0	suppliers	R					S	SELECT
1	3						O		SELECT
2	0	products	I	ix_prspid	1	Y			CORSUB

1	0	ranked_sp	R					S	SELECT
2	0	sp	R					S	TABLEX
2	3						O		TABLEX
3	0	suppliers	R					S	TABLEX
4	0	products	I	ix_prspid	1	Y			CORSUB

Not so common SQL syntactic constructions (3/3)

Scalar subqueries: only need to guarantee a 1 by 1 result ...

Example: *give per product the warehouse with the highest stock*:

```
SELECT   prname,
         (SELECT whname || ', ' || whtown FROM warehouses JOIN stocks ON whid=st_whid
          WHERE st_prclass = p.prclass AND st_prno = p.prno
          ORDER BY stquantity DESC   FETCH FIRST ROW ONLY )  AS warehouse
FROM     products p
```

QB	Method	Table	AccTyp	AccessName	MatchCols	IndexOnly	Sort_UOG_CN	Prefetch	QBtype
1		products	R			N		S	SELECT
2		stocks	I	ix_stid	2	N			CORSUB
2	1	warehouses	I	ix_whid	1	N			CORSUB
2	3					N	O		CORSUB

=> PROCSU = 102252; alternative queries are orders of magnitude worse!
(only the alternative with RANK() & PARTITION BY has same performance)

Alternative query formulations:

```
WITH st AS (SELECT  st_prclass, st_prno, MAX(stquantity)  AS maxquantity  FROM stocks
                   GROUP BY st_prclass, st_prno)
,        st_wh AS (SELECT st.*, st_whid
                         FROM  st JOIN stocks ON st.st_prclass=stocks.st_prclass AND st.st_prno=stocks.st_prno AND st.maxquantity = stocks.stquantity)
SELECT prname, whname, whtown
FROM   products LEFT JOIN st_wh ON prclass=st_prclass AND prno=st_prno
                          LEFT JOIN warehouses ON whid = st_whid
```

==> PROCSU = 40843344 : 400 times worse! (because the very large table STOCKS is accessed twice ...)

```
WITH st AS (SELECT  st_prclass, st_prno, st_whid, RANK() OVER (PARTITION BY st_prclass, st_prno ORDER BY stquantity DESC)  AS q_rank
                   FROM stocks)
SELECT prname, whname, whtown
FROM   products LEFT JOIN st ON prclass=st_prclass AND prno=st_prno
                          LEFT JOIN warehouses ON whid=st_whid
WHERE q_rank = 1
```

==> PROCSU = 107542 : identical, but logically slightly better since it returns *all* equally ranking warehouses

Your queries rewritten - for you or by you?

Peter Vanroose
ABIS Training & Consulting
pvanroose@abis.be

Session code: A3

Please fill out your session evaluation before leaving!