Dbvisit
we know data wins

Reading the execution plan – deep dive into SGA memory

October 3, 2017

Vit Spinka | Chief Architect

Dbvisit
we know data wins

- Explain plan, dbms_xplan
- How to read the SGA and PGA
- Find the plan for a SQL
- Rows of a plan
- Column projections, filters, access predicates
- It’s a tree

About me

Dbvisit
we know data wins

- Vit Spinka

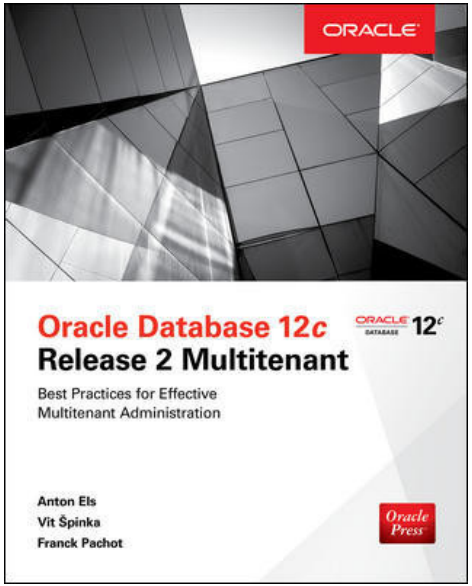
- Working with Oracle Database since 8*i*
- Oracle Certified Master
- Principal developer of Dbvisit Replicate
- … which gets its data by parsing Oracle redo logs

- @vitspinka
- vit.spinka@dbvisit.com
- This presentation download at http://vitspinka.cz/download.html

Multitenant book

- Oracle Database 12c Release 2 Multitenant
- Oracle Press
- Authors: Anton Els, Franck Pachot, Vit Spinka
- Covers many aspects of multitenant, aimed at DBAs

- Unlike this session, which targets nerds who think attaching gdb to an Oracle process is fun



Dbvisit
we know data wins

We already know where to look for an execution plan

Once upon a time, there was a SQL...

Dbvisit
we know data wins

- there are multiple ways how to see the SQL statement's execution plan
- EXPLAIN PLAN is long with us, but it has known issues
  - it's a new parse in new environment
  - bind lengths, types, charset, ...
- dbms_xplan can show an already existing plan
  - can read from AWR, baseline, SQL tuning set
  - and of course, from a cursor by sqlid and child
  - but parses fewer operations than EXPLAIN PLAN
- and v$sql_plan

display_cursor

Dbvisit
we know data wins

- misses many functions / complex operations completely
  - `filter("Q"."PROD_NAME"=)`
  - but often EXPLAIN PLAN can parse it:
  - `filter("Q"."PROD_NAME"= (SELECT LISTAGG(TO_CHAR("PROD_ID"),NULL) WITHIN GROUP ( ORDER BY "PROD_ID") FROM "PRODUCTS" "PRODUCTS"))`
- or obfuscates them
  - `filter((INTERNAL_FUNCTION("PRODUCTS"."PROD_ID") OR…`
  - `filter("PRODUCTS"."PROD_ID"=143 OR "PRODUCTS"."PROD_ID"=144 OR…`

And for some, even EXPLAIN PLAN fails

Dbvisit
we know data wins

- EXPLAIN PLAN handles a lot
- But sometime even that is not enough:
  - `filter(INTERNAL_FUNCTION("TIME_ID")=TIMESTAMP' 2000-01-01 00:00:00.000000000')`
- And of course, EXPLAIN PLAN is not a good option in the first place

Let's use an example

- Let's take one query as an example:
- ```
  SELECT prod_id, key
  FROM products
  CROSS JOIN foobar
  WHERE prod_id in (143,144,id) and id in (1,2,3);
  ```
- And see the execution plan:
  - ```
    SELECT * FROM
    TABLE(DBMS_XPLAN.DISPLAY(null,null,'ALL'));
    ```

```
SQL_ID b4hdxqwy614fa, child number 0
Plan hash value: 1321760920
-------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| CPU cost
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |       |       |     3 (100)|
|   1 |  NESTED LOOPS      |             |     1 |    34 |     3   (0)| 39293
|*  2 |   TABLE ACCESS FULL| FOOBAR      |     1 |    30 |     2   (0)|  7121
|*  3 |   INDEX FULL SCAN  | PRODUCTS_PK |     2 |     8 |     1   (0)| 32171
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(("FOOBAR"."ID"=1 OR "FOOBAR"."ID"=2 OR "FOOBAR"."ID"=3))
   3 - filter((INTERNAL_FUNCTION("PRODUCTS"."PROD_ID") OR
              "PRODUCTS"."PROD_ID"="FOOBAR"."ID"))
Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "FOOBAR"."KEY"[VARCHAR2,30], "PRODUCTS"."PROD_ID"[NUMBER,22]
   2 - "FOOBAR"."ID"[NUMBER,22], "FOOBAR"."KEY"[VARCHAR2,30]
   3 - "PRODUCTS"."PROD_ID"[NUMBER,22]
```

Plan is in v$sql_plan?

- Plan is in v$sql_plan, so we know where to start, don't we?
- The underlying table is x$kqlfxpl
- So let's have a look:
-
```
select addr, kqlfxpl_oper, kqlfxpl_oopt from x$kqlfxpl
where kqlfxpl_sqlid='b4hdxqwy614fa'
ADDR             KQLFXPL_OPER          KQLFXPL_OOPT
---------------- --------------------- ----------------
00007F9F384F2E68 SELECT STATEMENT
00007F9F384F2C10 NESTED LOOPS
00007F9F384F2990 TABLE ACCESS          FULL
00007F9F384F2698 INDEX                 FULL SCAN
```
- The addr is process memory!
- It's one of the x$ views with a 'helper' function – what we see is output of a function, not raw data in SGA

Dbvisit
we know data wins

Read the SGA or PGA

How to read the memory

Dbvisit
we know data wins

- x$ksmmem
  - Can read SGA
  - I don’t like it, did not work for me
- Direct SGA read
  - Fast and actually easy
- Read process memory
  - Can read PGA and process stack
  - But ptrace sends signals and stops process when you read data
- Or manually using gdb

Direct SGA read

- It's very easy
- Find shared memory segments
  - /proc/PID/maps
  - look at /SYSV segments
- Attach them
  - standard *shmat* call
- Read memory as any other memory address
  - *((uint8_t *)a)
- https://github.com/vit-spinka/direct-sga

Direct SGA read

Dbvisit
we know data wins

- This is easy and non-intrusive
- You just need to be *oracle* user
- But:
  - some info is not in SGA, as it's execution-specific, not cursor-specific
  - for example, bind variable values
  - or currently executed line (although…)
  - or SYSDATE
  - and as we will see, sometimes it's not the most direct way

Read process memory

Dbvisit
we know data wins

- This is not Oracle specific approach
- You can use ptrace() calls to read memory (and some other stuff) from a process
- But it stops the process between attach and detach, thus actually affecting the process
- A bit more complicated for thread-local storage
  - as it is in 12*c*
- You may hit permission issues
  - CAP_SYS_PTRACE

Read process memory

Dbvisit
we know data wins

- Again, the implementation is easy
- ptrace(PTRACE_ATTACH) to the process
  - this sends SIGSTOP to the process and you have to waitpid() for it
- ptrace(PTRACE_PEEKDATA) to read a memory address
- there are more, e.g. read a segment register
- there is even libunwind-ptrace to access the process stack in a structured way (like libunwind)
- ptrace (PTRACE_DETACH) to finish

Dbvisit
we know data wins

Find execution plan

So where is the plan really stored?

Dbvisit
we know data wins

- find out which function fills in x$kqlfxpl
  - perf
  - or gdb
  - or Google/MOS: it's kqlfgx
- and then more gdb to see what exactly does it access

But it's not that hard

Dbvisit
we know data wins

- the plan is in SGA
  - and you can query v$sql_plan to see all SQLs, so indeed all info is there
  - but if you try to follow the pointers from SQL address to cursor context, you will find it's a really long chain (14?) and a lot of magic offsets along the way
- fortunately you can also start in session context (kxscio)
  - which is in PGA
  - and cursor context is just one pointer away
- both ways work, the PGA one is more straightforward (github again)
  - but the SQL must be running *just now*

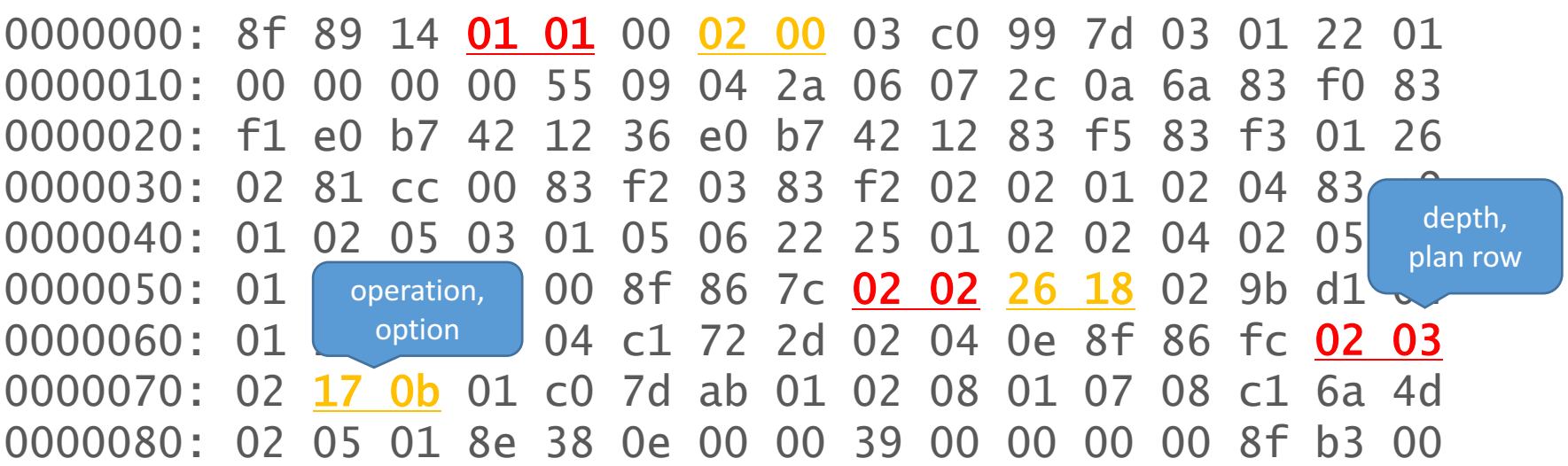
Finding the execution plan

- What are we looking for?
- First look at x$kqlfxpl and look for “interesting” numbers
  - e.g. CPU costs are large numbers, easy to spot (kqlfxpl_cpuc)
- And of course, look for the plan operation: kqlfxpl_oper, kqlfxpl_oopt
  - http://tech.e2sn.com/oracle/sql/oracle-execution-plan-operation-reference
  - x$xplton, x$xpltoo
  - in our plan, we have:
  - 0x37 SELECT STATEMENT
    0x02 NESTED LOOPS
    0x26 TABLE ACCESS 0x18 FULL
    0x17 INDEX 0x0b FULL SCAN

The plan is just one pointer away

- Looking at all the data at kxscio structure (=look at anything that might look like a pointer), we find something interesting at +0x2d0:

```
0000000: 8f 89 14 01 01 00 02 00 03 c0 99 7d 03 01 22 01
0000010: 00 00 00 00 55 09 04 2a 06 07 2c 0a 6a 83 f0 83
0000020: f1 e0 b7 42 12 36 e0 b7 42 12 83 f5 83 f3 01 26
0000030: 02 81 cc 00 83 f2 03 83 f2 02 02 01 02 04 83 e9
0000040: 01 02 05 03 01 05 06 22 25 01 02 02 04 02 05 13
0000050: 01 02 05 00 00 8f 86 7c 02 02 26 18 02 9b d1 02
0000060: 01 1e 01 07 04 c1 72 2d 02 04 0e 8f 86 fc 02 03
0000070: 02 17 0b 01 c0 7d ab 01 02 08 01 07 08 c1 6a 4d
0000080: 02 05 01 8e 38 0e 00 00 39 00 00 00 00 8f b3 00
```


0000000: 8f 89 14 01 01 00 02 00 03 c0 99 7d 03 01 22 01
0000010: 00 00 00 00 55 09 04 2a 06 07 2c 0a 6a 83 f0 83
0000020: f1 e0 b7 42 12 36 e0 b7 42 12 83 f5 83 f3 01 26
0000030: 02 81 cc 00 83 f2 03 83 f2 02 02 01 02 04 83
0000040: 01 02 05 03 01 05 06 22 25 01 02 02 04 02 05
0000050: 01
0000060: 01
0000070: 02 17 0b 01 c0 7d ab 01 02 08 01 07 08 c1 6a 4d
0000080: 02 05 01 8e 38 0e 00 00 39 00 00 00 00 8f b3 00
depth, plan row
operation, option
00 8f 86 7c 02 02 26 18 02 9b d1
04 c1 72 2d 02 04 0e 8f 86 fc 02 03

Some space saving happens, though

Dbvisit
we know data wins

- The first number is a flag – 0x8f = plan row, 0x8e = end
- The second number is a bitmap
  - Denotes fields present
    - Partition info
    - Costs
    - Temp space
    - Object id
    - And more I have no idea what they mean
- The values are compressed a bit similar to UTF-8
  - Based on first 3 bits, the value is 1-5 bytes long
  - Values <0x80 are stored verbatim

```
0000000: 8f 89 14 01 01 00 02 00 03 c0 99 7d 03 01 22 01
0000010: 00 00 00 00 55 09 04 2a 06 07 2c 0a 6a 83 f0 83
0000020: f1 e0 b7 42 12 36 e0 b7 42 12 83 f5 83 f3 01 26
0000030: 02 81 cc 00 83 f2 03 83 f2 02 02 01 02 04 83 e9
0000040: 01 02 05 03 01 05 06 22 25 01 02 02 04 02 05 13
0000050: 01 02 05 00 00 8f 86 7c 02 02 26 18 02 9b d1 02
0000060: 01 1e 01 07 04 c1 72 2d 02 04 0e 8f 86 fc 02 03
0000070: 02 17 0b 01 c0 7d ab 01 02 08 01 07 08 c1 6a 4d
0000080: 02 05 01 8e
```

```
8914 -> 0x914
867c -> 0x67c
86fc -> 0x6fc
```

```
0000000: 8f 89 14 01 01 00 02 00 03 c0 99 7d 03 01 22 01
0000010: 00 00 00 00 55 09 04 2a 06 07 2c 0a 6a 83 f0 83
0000020: f1 e0 b7 42 12 36 e0 b7 42 12 83 f5 83 f3 01 26
0000030: 02 81 cc 00 83 f2 03 83 f2 02 02 01 02 04 83 e9
0000040: 01 02 05 03 01 05 06 22 25 01 02 02 04 02 05 13
0000050: 01 02 05 00 00 8f 86 7c 02 02 26 18 02 9b d1 02
0000060: 01 1e 01 07 04 c1 72 2d 02 04 0e 8f 86 fc 02 03
0000070: 02 17 0b 01 c0 7d ab 01 02 08 01 07 08 c1 6a 4d
0000080: 02 05 01 8e

CPU costs:
c0997d = 0x997d = 39293
  9bd1 =  0xbd1 =  7121
c07dab = 0x7dab = 32171

cost, IO cost, cardinality, bytes

object_id
```

Dbvisit
we know data wins

Columns, filters, access predicates

But so far we saw only half of the picture

Dbvisit
we know data wins

- This was the upper half of the execution plan
- But the plan also shows column projections, filters and access predicates
- And yes, these are in the plan in the SGA, too
- At cursor context + 0x320, there is an array of pointers to tree nodes

The nodes form a tree

Dbvisit
we know data wins

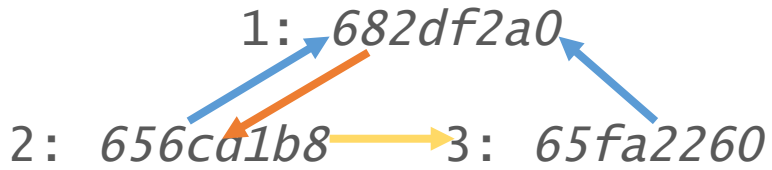
- Each plan row has a node in the tree
- Extra nodes are possible, not listed in this array
- Each node has basic header
  - plan row id (or -1 if not in that array)
  - pointer to parent
  - pointer to next sibling
  - pointer to first child

```
0000000 00000000682df2a0 00000000656cd1b8
0000020 0000000065fa2260

682df2a0
0000000 0001000100000008 0000000000000000      row     parent
0000020 0000000000000000 00000000656cd1b8      sibling child

656cd1b8
0000000 0001000200000008 00000000682df2a0
0000020 0000000065fa2260 0000000000000000

65fa2260
0000000 0001000300000308 00000000682df2a0
0000020 0000000000000000 0000000000000000
```



Start with first node, then traverse down by the child and sibling links.

Column projection

- And the next pointer is to column projections; follow it and you arrive at:
  - **column count**, pointers to their definitions

```
1: 00000000682df268
0000000 0000000000010002 0000000000000000
0000020 0000000065fa2ac8 0000000065fa2190

2: 00000000656cd158
0000000 0000000000010002 0000000000000000
0000020 0000000065fa2bc8 0000000065fa2ac8

3: 0000000065fa2208
0000000 0000000000010001 0000000000000000
0000020 0000000065fa2190
```

Column definition

- A column definition has a **datatype** (oacdty like in 10046 or oci.h), **length**
- And **type**… but more about it later. Let's assume it's simply a column

```
"FOOBAR"."KEY"[VARCHAR2,30]
0000000: 0b 00 00 00 01 01 00 00 00 00 00 00 1e 00 00 00
0000010: 69 03 01 00 00 00 00 00 20 00 00 00 d8 02 00 00
…
0000050: 90 2a fa 65 00 00 00 00

"PRODUCTS"."PROD_ID"[NUMBER,22]
0000000: 0b 00 00 00 02 01 04 00 00 00 00 00 16 00 00 00
0000010: 06 00 00 00 00 00 00 00 20 00 00 00 d8 03 00 00
…
0000050: 58 21 fa 65 00 00 00 00
```

Column definition

- If we follow the **pointer** at 0x50, we get column definition: schema, table, column
  - parts are optional, e.g. there is no schema here

```
"PRODUCTS"."PROD_ID"[NUMBER,22]
0000000 0000000000000000 0000000065fa2130
0000020 0000000065fa2108

0000000065fa2130
1900 0000 0800 5052 4f44 5543 5453 0000  ......PRODUCTS..

0000000065fa2108
0000 0000 0700 5052 4f44 5f49 4400 0000  ......PROD_ID...
```

Filters and access predicates

- Tree nodes contain pointers to filters and access predicates, too
- The exact location depends on flag at +0x34 (and there is no obvious pattern)
- A node can have filter, access, both, more of them.

```
1: 682df2a0
at 0x34: 0x52 -> nothing

2: 656cd1b8
at 0x34: 0x17 -> filter at +0x78: 0000000065fa2998

3: 65fa2260
 at 0x34: 0x51 -> filters at +0x48 and +0x68:
                  0000000065fa2078 00000000682df370
```

use this one in further examples

Filters and access predicates

Dbvisit
we know data wins

- The format is the same as for column projections!
- Because they don't describe "column", but an "expression". We saw 0xb – which is simply a column reference. But there are more, e.g.:
  - operation (**0xc**)
  - constant
  - new column (think "STRDEF" you see with unions etc.)
  - PL/SQL

```
0000000065fa2078:
0000000: 0c 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00
0000030: 87 02 00 00 00 00 00 00 02 00 00 00 00 00 00 00
0000060: 0000000000000000 0000000065fa1fe8
0000070: 00000000682df400 00b38f0000000029
```

Filters and access predicates

```
0000000065fa2078:
0000000: 0c 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00
0000030: 87 02 00 00 00 00 00 00 02 00 00 00 00 00 00 00
0000060: 0000000000000000 0000000065fa1fe8
0000070: 00000000682df400 00b38f0000000029
```

- 0x287 is function id:

```
select name, descr from V$SQLFN_METADATA where func_id=0x278;
NAME    DESCR
------- ------
OPTIOR  IOR
```

- 0x2 is argument count
- And the green ones are the arguments
  - let's follow the first one

Filters and access predicates

```
0000000065fa1fe8:
0000000: 0c 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00
0000030: 06 02 00 00 00 00 00 00 02 00 00 00 00 00 00 00
0000060: 0000000000000000 0000000065fa1f70
0000070: 0000000065fa1ec0 00b38f0000000029
```

- 0x206 is function id:

```
select name, descr from V$SQLFN_METADATA where func_id=0x206;
NAME      DESCR
--------- ----------------------
OPTTINLO  inlist equality ORACLE
```

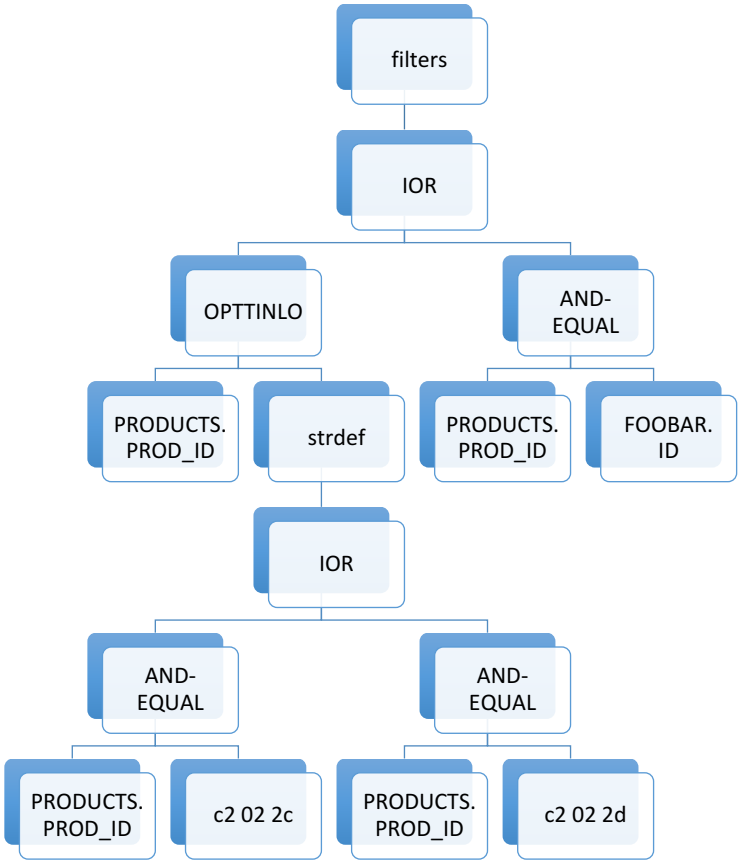
- And again…
- it's OPTTINLO(PRODUCTS.PROD_ID, *STRDEF*)… And the STRDEF is another IOR of two AND-EQUAL of column and constant.

Filters and access predicates

```
SQL:
where prod_id in (143,144,id)

dbms_xplan: filter(
(INTERNAL_FUNCTION("PRODUCTS".
"PROD_ID") OR
"PRODUCTS"."PROD_ID"="FOOBAR".
"ID"))

EXPLAIN PLAN: filter(
"PRODUCTS"."PROD_ID"=143 OR
"PRODUCTS"."PROD_ID"=144 OR
"PRODUCTS"."PROD_ID"="FOOBAR".
"ID")
```



Last few notes

Dbvisit
we know data wins

- Often the expressions are shared among nodes
- In this case, the two filters pointed to the same IOR
- Column projections build up, they can refer to an expression established in a different node
  - this allows to correctly identify where exactly data come from, e.g. with unions, temporary tables etc.

Dbvisit
we know data wins

Anyone still awake for questions?

Bonus

- So what was the second example where even EXPLAIN PLAN failed?
- It's from *Database VLDB and Partitioning Guide*, on SH schema:
- ```
  SELECT SUM(quantity_sold)
  FROM sales
  WHERE time_id=TO_TIMESTAMP('1-jan-2000','dd-mon-yyyy');
  ```

- And the hidden function is…

```
NAME           DESCR
-------------- ------------------------------
OPTDAT2TS1     Srray Date=>Array Timestamp
```
```