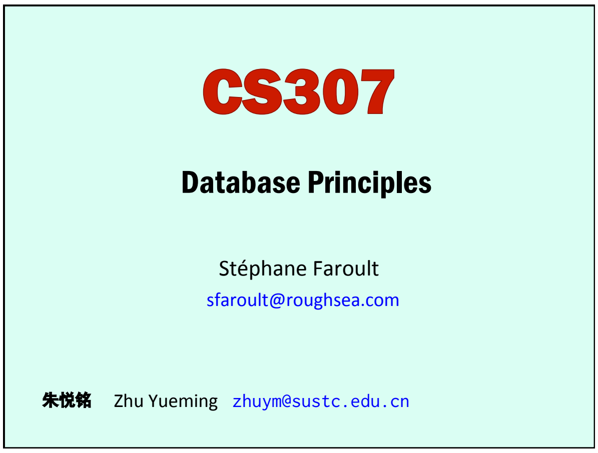
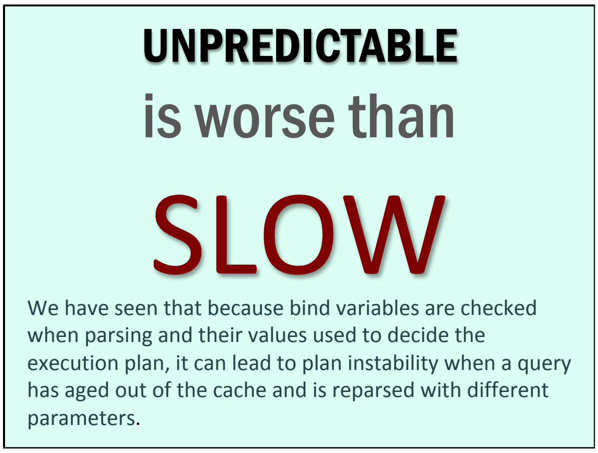


CS307

Database Principles

Stéphane Faroult
sfaroult@roughsea.com

朱悦铭 Zhu Yueming zhuym@sustc.edu.cn

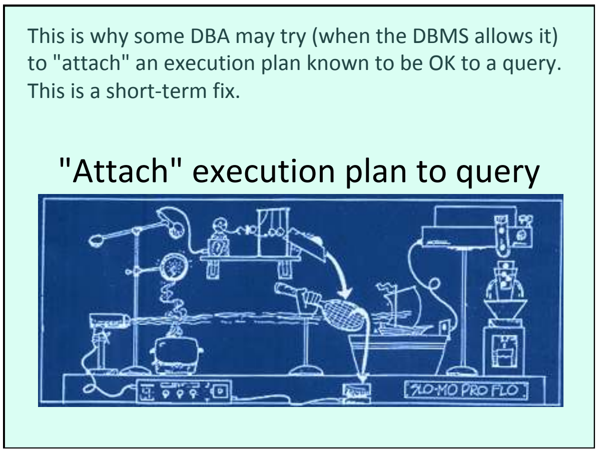


UNPREDICTABLE

is worse than

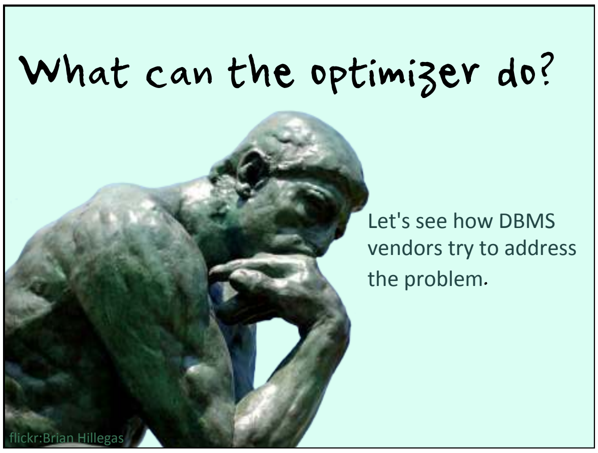
SLOW

We have seen that because bind variables are checked when parsing and their values used to decide the execution plan, it can lead to plan instability when a query has aged out of the cache and is reparsed with different parameters.



This is why some DBA may try (when the DBMS allows it) to "attach" an execution plan known to be OK to a query. This is a short-term fix.

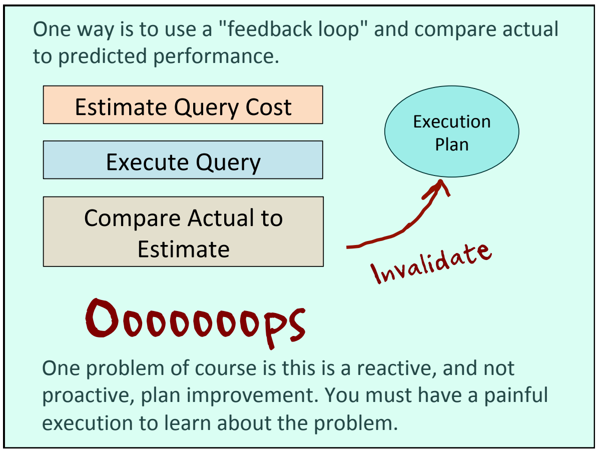
"Attach" execution plan to query



What can the optimizer do?

Let's see how DBMS vendors try to address the problem.

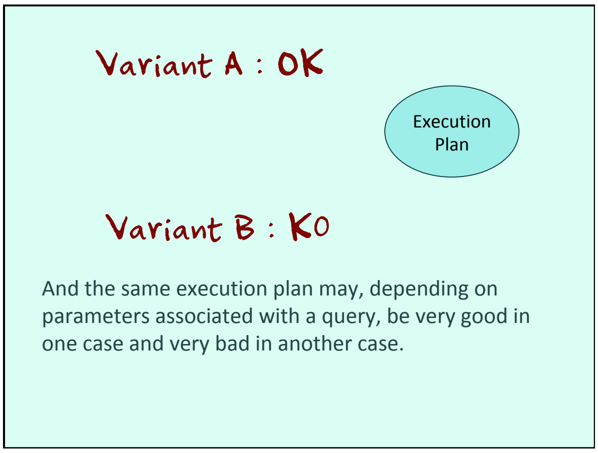
flickr:Brian Hillegas



One way is to use a "feedback loop" and compare actual to predicted performance.

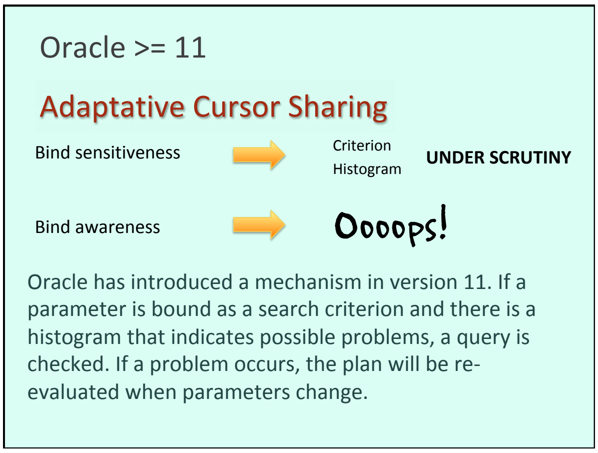
Estimate Query Cost
Execute Query
Compare Actual to Estimate
Execution Plan
Invalidate
Ooooooops

One problem of course is this is a reactive, and not proactive, plan improvement. You must have a painful execution to learn about the problem.



Variant A : OK
Execution Plan
Variant B : KO

And the same execution plan may, depending on parameters associated with a query, be very good in one case and very bad in another case.

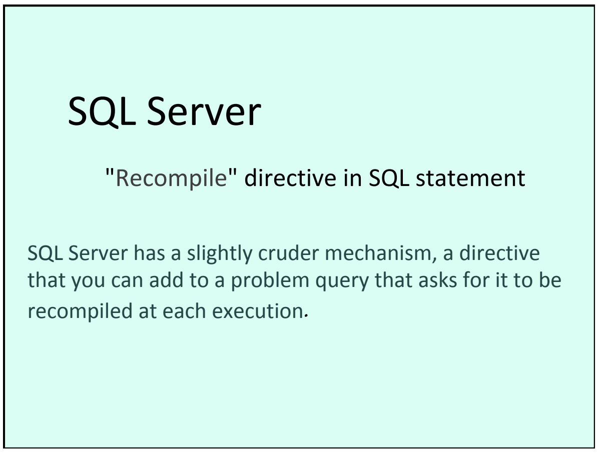


Oracle >= 11

Adaptative Cursor Sharing

Bind sensitiveness
Criterion
Histogram
UNDER SCRUTINY
Bind awareness
Oooops!

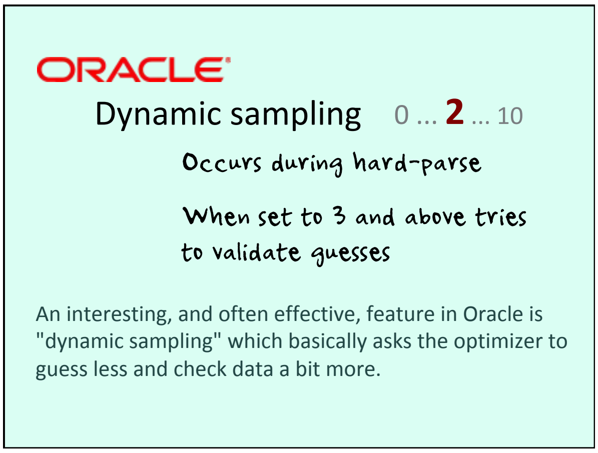
Oracle has introduced a mechanism in version 11. If a parameter is bound as a search criterion and there is a histogram that indicates possible problems, a query is checked. If a problem occurs, the plan will be re-evaluated when parameters change.



SQL Server

"Recompile" directive in SQL statement

SQL Server has a slightly cruder mechanism, a directive that you can add to a problem query that asks for it to be recompiled at each execution.



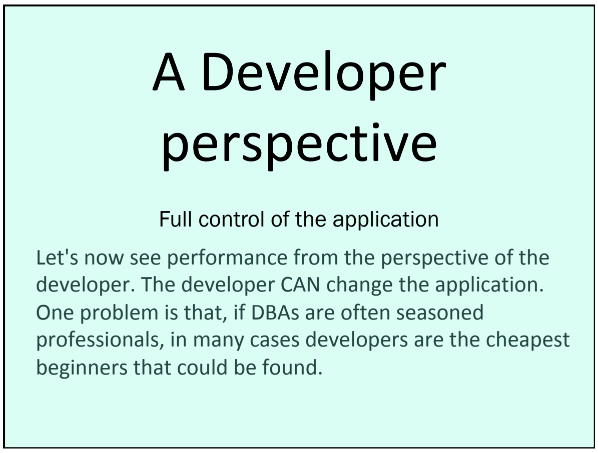
ORACLE

Dynamic sampling 0 ... **2** ... 10

Occurs during hard-parse

When set to 3 and above tries to validate guesses

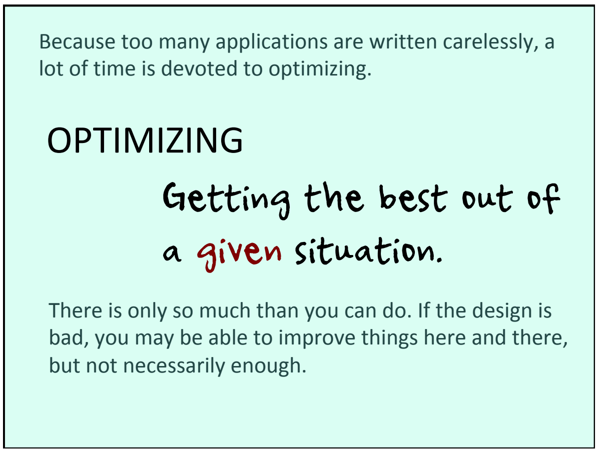
An interesting, and often effective, feature in Oracle is "dynamic sampling" which basically asks the optimizer to guess less and check data a bit more.



A Developer perspective

Full control of the application

Let's now see performance from the perspective of the developer. The developer CAN change the application. One problem is that, if DBAs are often seasoned professionals, in many cases developers are the cheapest beginners that could be found.

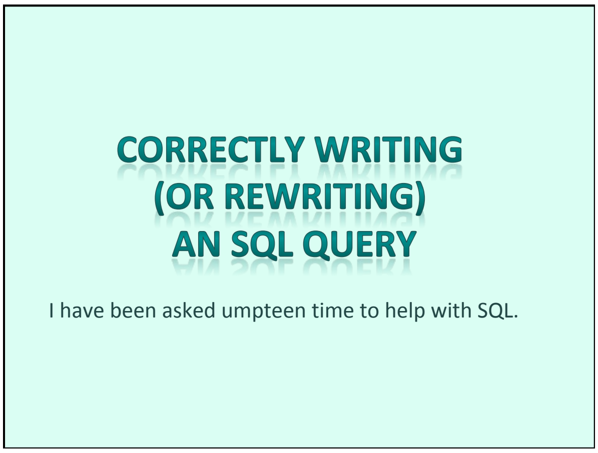


Because too many applications are written carelessly, a lot of time is devoted to optimizing.

OPTIMIZING

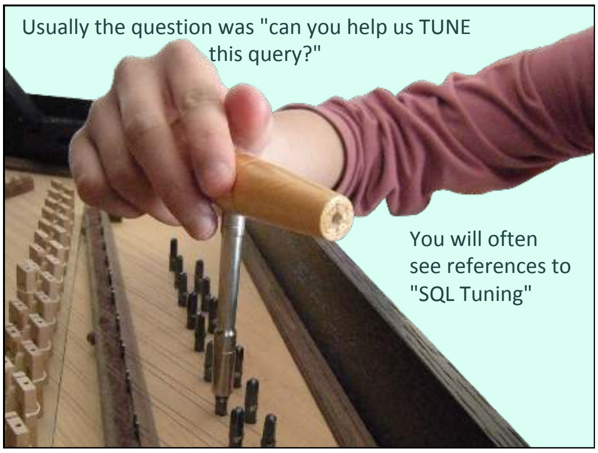
Getting the best out of a given situation.

There is only so much than you can do. If the design is bad, you may be able to improve things here and there, but not necessarily enough.



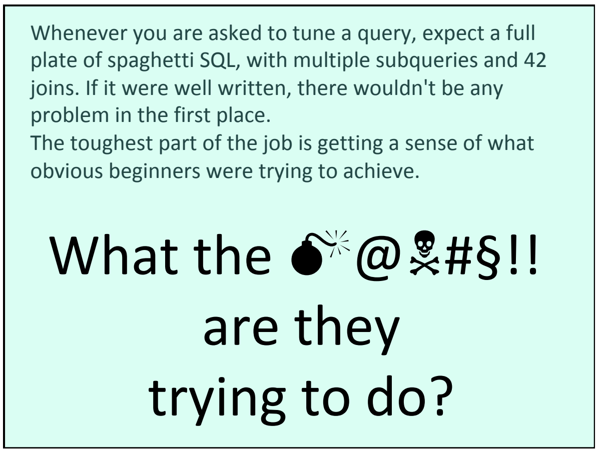
CORRECTLY WRITING (OR REWRITING) AN SQL QUERY

I have been asked umpteen time to help with SQL.



Usually the question was "can you help us TUNE this query?"

You will often see references to "SQL Tuning"



Whenever you are asked to tune a query, expect a full plate of spaghetti SQL, with multiple subqueries and 42 joins. If it were well written, there wouldn't be any problem in the first place.
The toughest part of the job is getting a sense of what obvious beginners were trying to achieve.

What the 💣@☠#§!!
are they
trying to do?

When you have understood that, "tuning" is usually a matter or rewriting everything from scratch.

Flickr: Gandhirama

Go for the

core

What matters in the query? For instance, country names are accessory, but country identifiers are important.

Phase 1

Flickr: Hendriko

Remove what doesn't

shape

the result set

What is really important is what determines the magnitude of the number of rows returned by the query. When you rewrite a big, ugly query, first get rid of everything that changes nothing or little to the final number of rows returned.

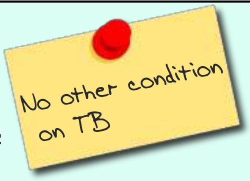
Joins to tables without any `col = constant` condition

When you join to tables to which no direct criteria are applied, are these joins really important?

foreign key or **NOT** foreign key ?

If there is a foreign key, probably not: you know that for every row you'll find a match in the other table, so the join won't affect the number of rows returned.

```
select …
from TA, TB
where …
  and TA.C = TB.C
…
```



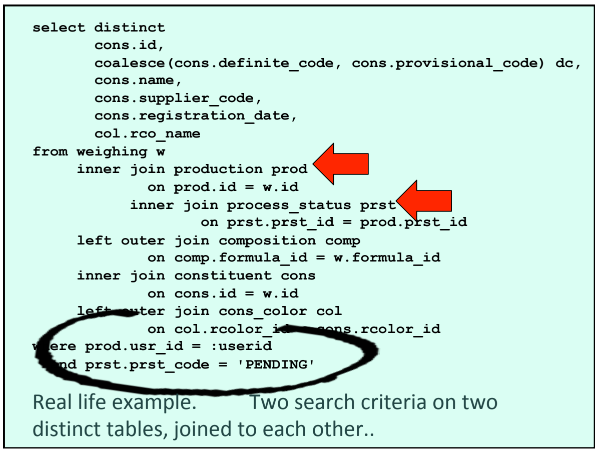
If C is a **foreign key** for A that references B,

TB **DOESN'T** belong to the core of the query

If some values of C in TA **ARE NOT** in TB

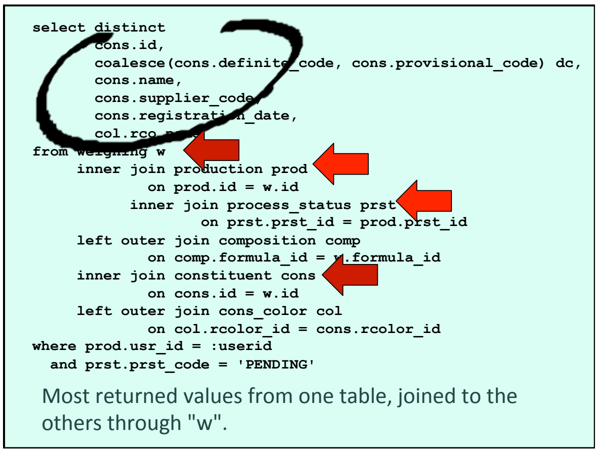
TB **BELONGS** to the core of the query

In that case the join filters rows.



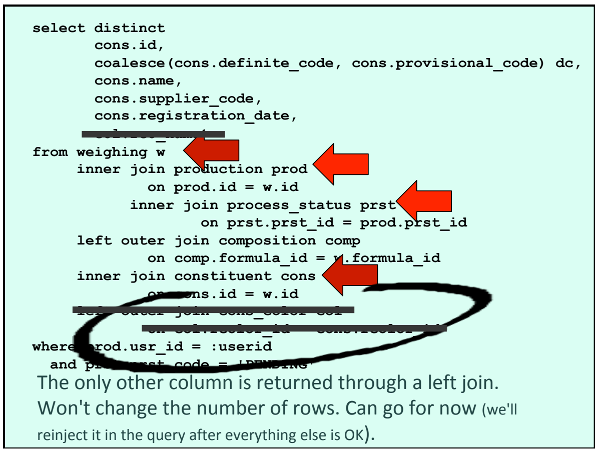
```
select distinct
       cons.id,
       coalesce(cons.definite_code, cons.provisional_code) dc,
       cons.name,
       cons.supplier_code,
       cons.registration_date,
       col.rco_name
from weighing w
     inner join production prod
         on prod.id = w.id
           inner join process_status prst
               on prst.prst_id = prod.prst_id
     left outer join composition comp
         on comp.formula_id = w.formula_id
     inner join constituent cons
         on cons.id = w.id
     left outer join cons_color col
         on col.rcolor_id = cons.rcolor_id
where prod.usr_id = :userid
  and prst.prst_code = 'PENDING'
```

Real life example. Two search criteria on two distinct tables, joined to each other..



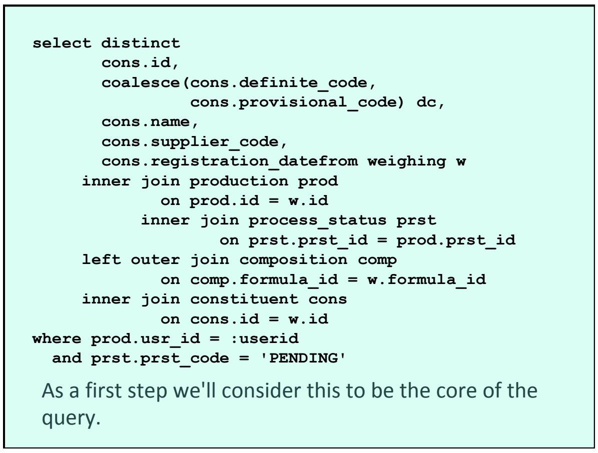
```
select distinct
       cons.id,
       coalesce(cons.definite_code, cons.provisional_code) dc,
       cons.name,
       cons.supplier_code,
       cons.registration_date,
       col.rco_name
from weighing w
     inner join production prod
         on prod.id = w.id
           inner join process_status prst
               on prst.prst_id = prod.prst_id
     left outer join composition comp
         on comp.formula_id = w.formula_id
     inner join constituent cons
         on cons.id = w.id
     left outer join cons_color col
         on col.rcolor_id = cons.rcolor_id
where prod.usr_id = :userid
  and prst.prst_code = 'PENDING'
```

Most returned values from one table, joined to the others through "w".



```
select distinct
       cons.id,
       coalesce(cons.definite_code, cons.provisional_code) dc,
       cons.name,
       cons.supplier_code,
       cons.registration_date,
       ~~col.rco_name~~
from weighing w
     inner join production prod
         on prod.id = w.id
           inner join process_status prst
               on prst.prst_id = prod.prst_id
     left outer join composition comp
         on comp.formula_id = w.formula_id
     inner join constituent cons
         on cons.id = w.id
     ~~left outer join cons_color col~~
         ~~on col.rcolor_id = cons.rcolor_id~~
where prod.usr_id = :userid
  and prst.prst_code = 'PENDING'
```

The only other column is returned through a left join. Won't change the number of rows. Can go for now (we'll reinject it in the query after everything else is OK).



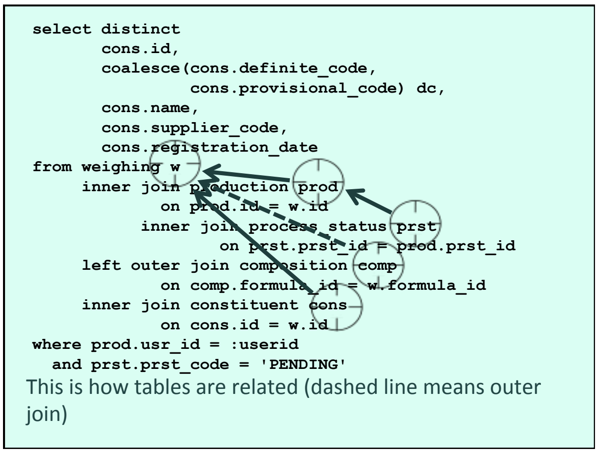
```
select distinct
       cons.id,
       coalesce(cons.definite_code,
                cons.provisional_code) dc,
       cons.name,
       cons.supplier_code,
       cons.registration_datefrom weighing w
     inner join production prod
           on prod.id = w.id
        inner join process_status prst
              on prst.prst_id = prod.prst_id
     left outer join composition comp
           on comp.formula_id = w.formula_id
     inner join constituent cons
           on cons.id = w.id
where prod.usr_id = :userid
  and prst.prst_code = 'PENDING'
```

As a first step we'll consider this to be the core of the query.

Optimize the core of the query

Now let's try to retrieve all the rows as fast as possible.

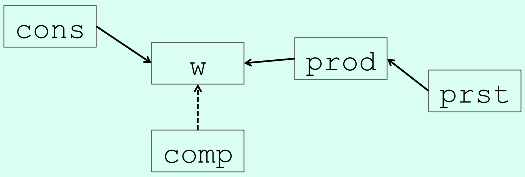
Phase 2



```
select distinct
       cons.id,
       coalesce(cons.definite_code,
                cons.provisional_code) dc,
       cons.name,
       cons.supplier_code,
       cons.registration_date
from weighing w
     inner join production prod
             on prod.id = w.id
           inner join process_status prst
                   on prst.prst_id = prod.prst_id
     left outer join composition comp
             on comp.formula_id = w.formula_id
     inner join constituent cons
             on cons.id = w.id
where prod.usr_id = :userid
  and prst.prst_code = 'PENDING'
```

This is how tables are related (dashed line means outer join)

Chain of tables



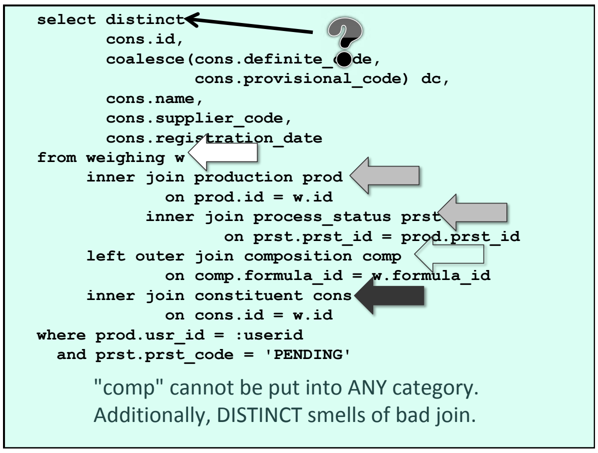
It can be represented as a "chain" of tables linked to each other through joins.

Tables from which data is returned
With or without conditions

Tables from which **NO** data is returned
Conditions only

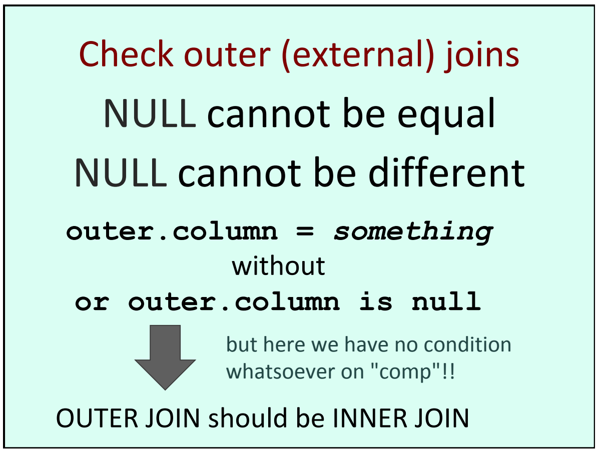
Glue Tables
Join other tables

Classifying tables



```
select distinct
        cons.id,
        coalesce(cons.definite_code,
                 cons.provisional_code) dc,
        cons.name,
        cons.supplier_code,
        cons.registration_date
from weighing w
      inner join production prod
              on prod.id = w.id
            inner join process_status prst
                    on prst.prst_id = prod.prst_id
      left outer join composition comp
              on comp.formula_id = w.formula_id
      inner join constituent cons
              on cons.id = w.id
where prod.usr_id = :userid
  and prst.prst_code = 'PENDING'
```

"comp" cannot be put into ANY category.
Additionally, DISTINCT smells of bad join.



Check outer (external) joins

NULL cannot be equal

NULL cannot be different

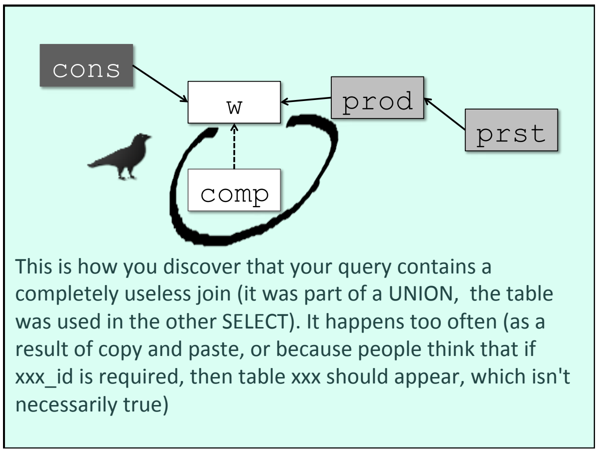
`outer.column = something`

without

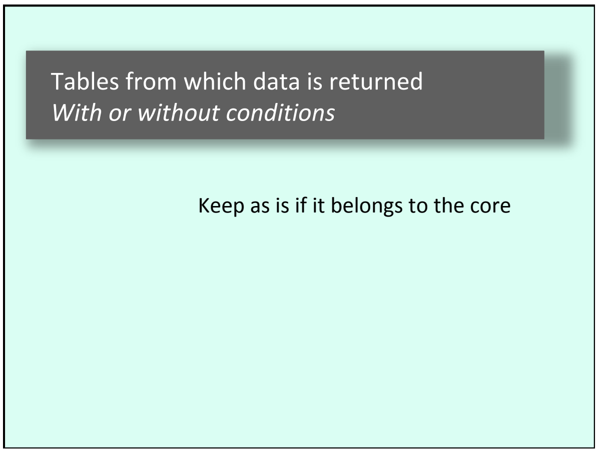
`or outer.column is null`

but here we have no condition whatsoever on "comp"!!

OUTER JOIN should be INNER JOIN



This is how you discover that your query contains a completely useless join (it was part of a UNION, the table was used in the other SELECT). It happens too often (as a result of copy and paste, or because people think that if xxx_id is required, then table xxx should appear, which isn't necessarily true)



Tables from which data is returned
With or without conditions

Keep as is if it belongs to the core

Tables from which **NO** data is returned
Conditions only

Turn into a subquery

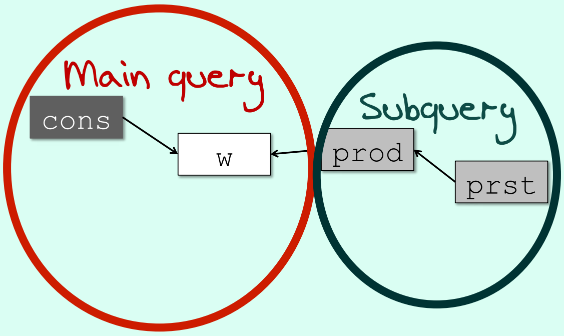
Subqueries (especially IN () that takes care of duplicates) are a good way to get rid of DISTINCT, which should always be a black flag at the top of a big query. You may decide later which type of subquery you need and whether you should turn them into joins but they are a great way to make a query more legible.

Glue Tables
Join other tables

Useless at the end of a chain

Main query or subquery

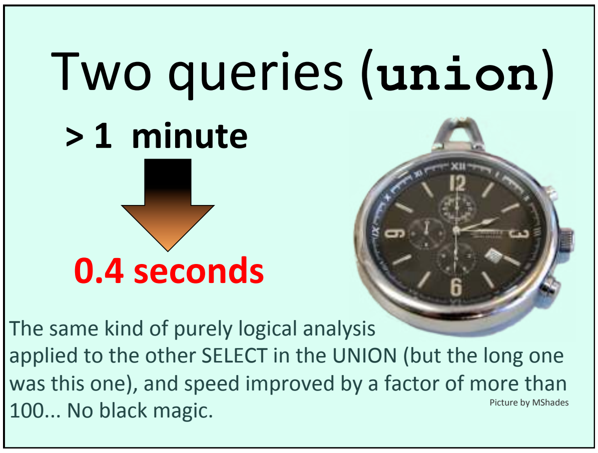
A "glue" table that is glued to one useful table and nothing else can be safely disposed of, as we did with "comp". Remaining glue tables that link outer query tables to inner query tables can indifferently appear in the inner or outer query.



I have chosen to keep (for balance ...) "w" in the outer query, and to have a subquery that joins "prod" and "prst"

```
Select cons.id,
       coalesce(cons.definite_code,
                cons.production_code) dc,
       cons.name,
       cons.supplier_code,
       cons.registration_date
from weighing w
     inner join constituent cons
             on cons.id = w.id
where w.id in
      (select prod.id
       from production prod
            inner join process_status prst
                    on prst.prst_id = prod.prst_id
       where prod.usr_id = :userid
         and prst.prst_code = 'PENDING')
```

And lo, DISTINCT is gone. Is "w" really necessary? There was no FK, I couldn't tell, I kept it.



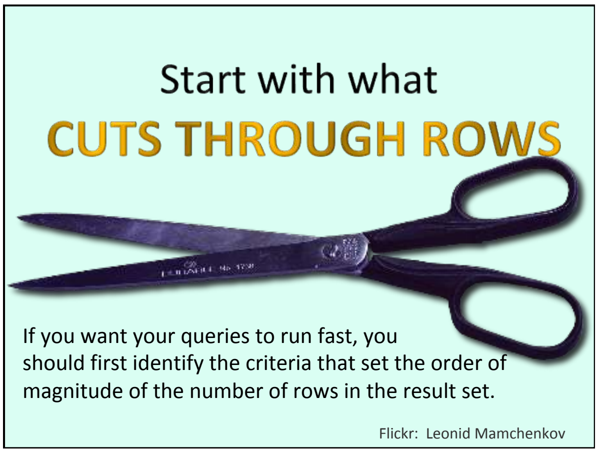
Two queries (`union`)

> 1 minute

0.4 seconds

The same kind of purely logical analysis applied to the other SELECT in the UNION (but the long one was this one), and speed improved by a factor of more than 100... No black magic.

Picture by MShades



Start with what CUTS THROUGH ROWS

If you want your queries to run fast, you should first identify the criteria that set the order of magnitude of the number of rows in the result set.

Flickr: Leonid Mamchenkov

GROUP and sort AS LITTLE AS POSSIBLE

Then use aggregates while only working with identifiers and "small" data, not big, fully joined rows.

Flickr: Belinda Hankins Miller

Join Late!

Keep for the very end (the top level, the outer query) all joins that change nothing (or very little) to the size of the result set.

Flickr: Jim WInstead

What about

Hints?

The approach that is advocated here is based on the understanding of the functional aspect of a query, and focusing on a logical data search. Many self-styled experts take another approach, consider that the optimizer had it wrong (often true) and prefer telling the optimizer what to do using "hints".

Hint: special directive to tell the optimizer what to do

Those "hints" exist with all DBMS products and were mostly introduced when query optimizers were young; they were a convenient way to override optimizer bugs.

[Hint;Query hint]

Microsoft® SQL Server®

```
SELECT p.Name, pr.ProductReviewID
FROM Production.Product AS p
LEFT OUTER HASH JOIN
    Production.ProductReview AS pr
        ON p.ProductID = pr.ProductID
ORDER BY ProductReviewID DESC;
```

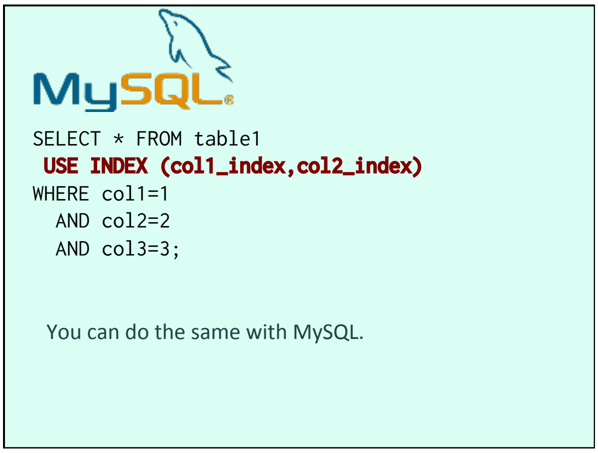
[Hint;Query hint]

For instance you can tell to SQL Server which join algorithm it should use.

Microsoft® SQL Server®

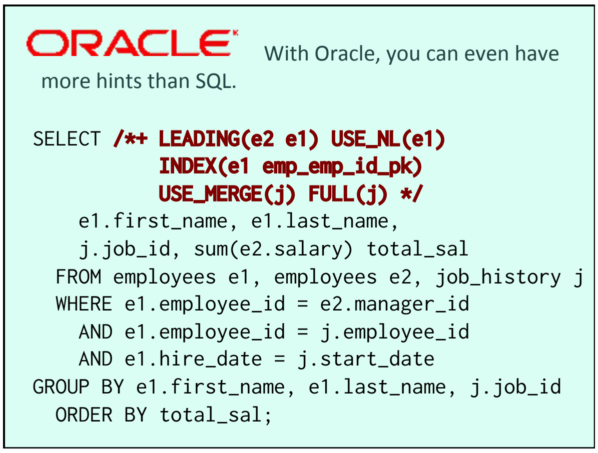
```
SELECT c.LastName, c.FirstName, e.Title
FROM HumanResources.Employee AS e
   WITH (NOLOCK, INDEX (PK_Employee_EmployeeID))
     JOIN Person.Contact AS c
       ON e.ContactID = c.ContactID
WHERE e.ManagerID = 3;
```

Or which index it should use.



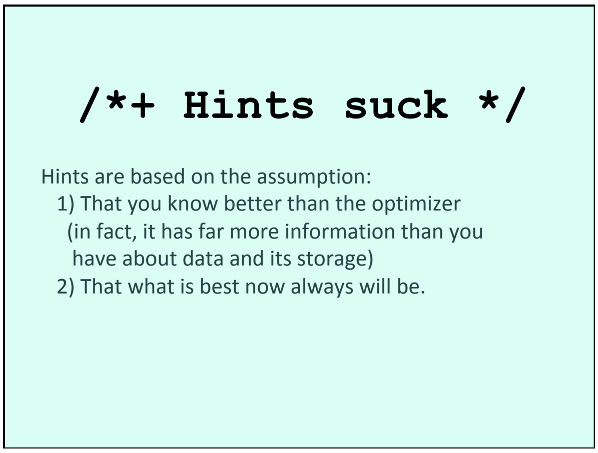
```
SELECT * FROM table1
 USE INDEX (col1_index,col2_index)
WHERE col1=1
  AND col2=2
  AND col3=3;
```

You can do the same with MySQL.



With Oracle, you can even have more hints than SQL.

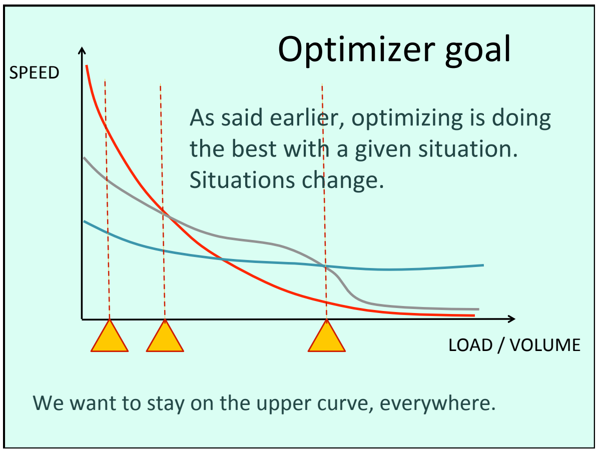
```
SELECT /*+ LEADING(e2 e1) USE_NL(e1)
           INDEX(e1 emp_emp_id_pk)
           USE_MERGE(j) FULL(j) */
    e1.first_name, e1.last_name,
    j.job_id, sum(e2.salary) total_sal
  FROM employees e1, employees e2, job_history j
  WHERE e1.employee_id = e2.manager_id
    AND e1.employee_id = j.employee_id
    AND e1.hire_date = j.start_date
GROUP BY e1.first_name, e1.last_name, j.job_id
  ORDER BY total_sal;
```



/*+ Hints suck */

Hints are based on the assumption:

1) That you know better than the optimizer (in fact, it has far more information than you have about data and its storage)
2) That what is best now always will be.



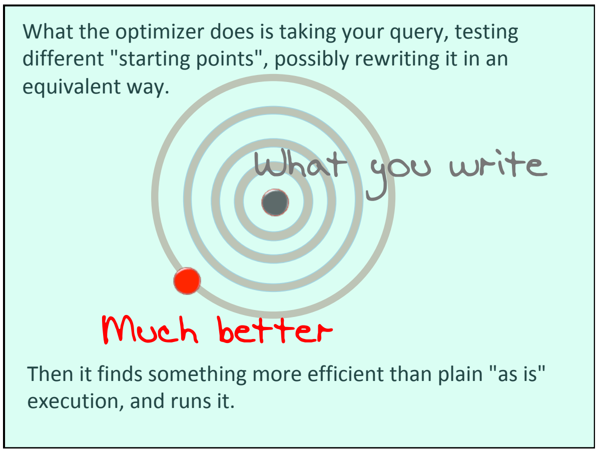
Optimizer goal

As said earlier, optimizing is doing the best with a given situation. Situations change.

We want to stay on the upper curve, everywhere.

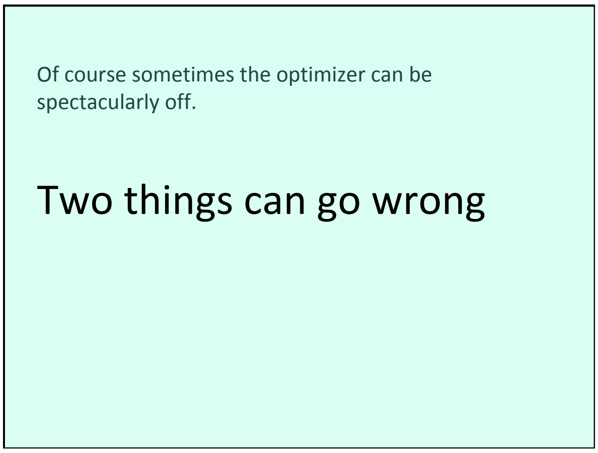
I don’t care about execution plans
I care about correctness and **SPEED**

Whether the DBMS is scanning, using indexes, nested loops, hash joins, is irrelevant as long as it's fast.



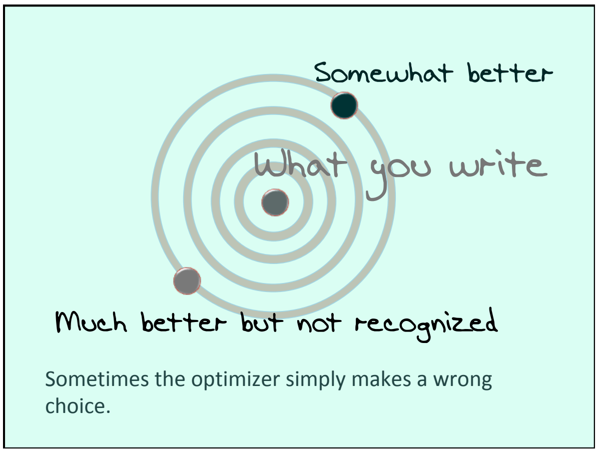
What the optimizer does is taking your query, testing different "starting points", possibly rewriting it in an equivalent way.

Then it finds something more efficient than plain "as is" execution, and runs it.

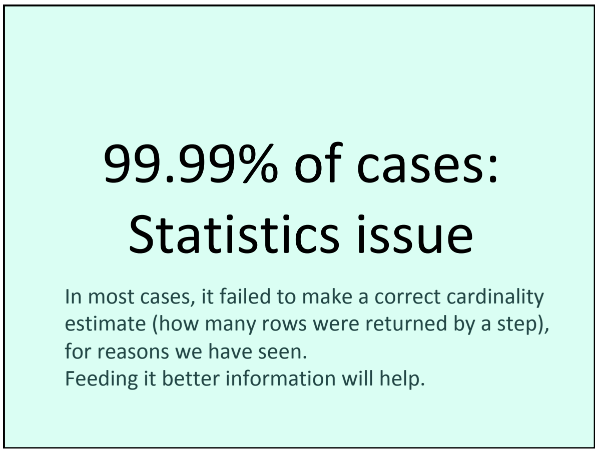


Of course sometimes the optimizer can be spectacularly off.

Two things can go wrong

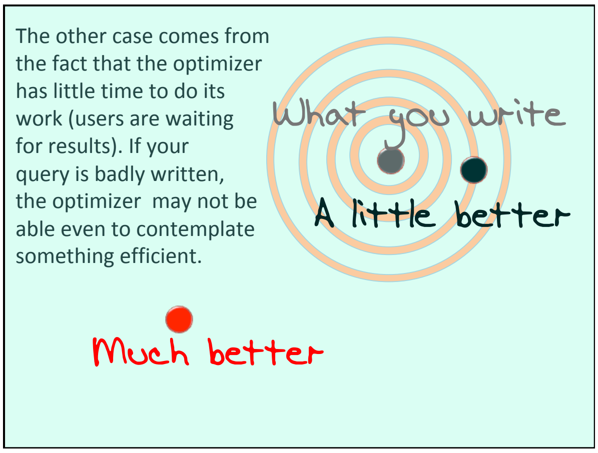


Sometimes the optimizer simply makes a wrong choice.



99.99% of cases: Statistics issue

In most cases, it failed to make a correct cardinality estimate (how many rows were returned by a step), for reasons we have seen.
Feeding it better information will help.

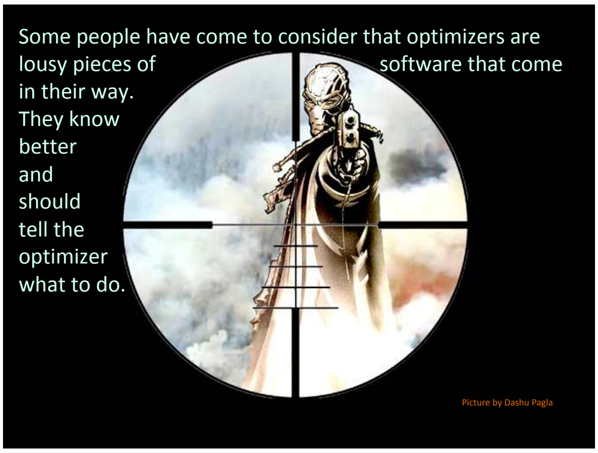


The other case comes from the fact that the optimizer has little time to do its work (users are waiting for results). If your query is badly written, the optimizer may not be able even to contemplate something efficient.

What you write

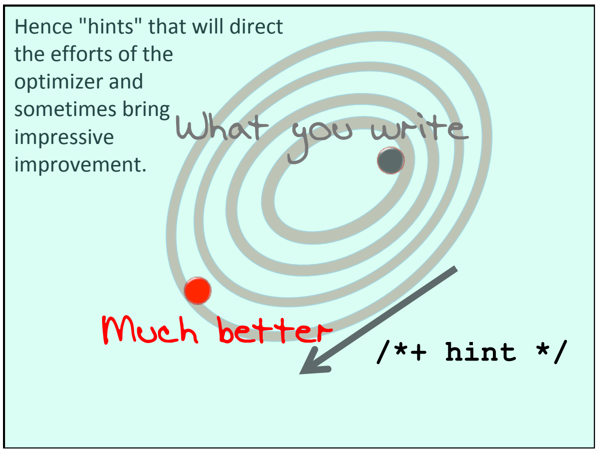
A little better

Much better



Some people have come to consider that optimizers are lousy pieces of software that come in their way. They know better and should tell the optimizer what to do.

Picture by Dashu Pagla

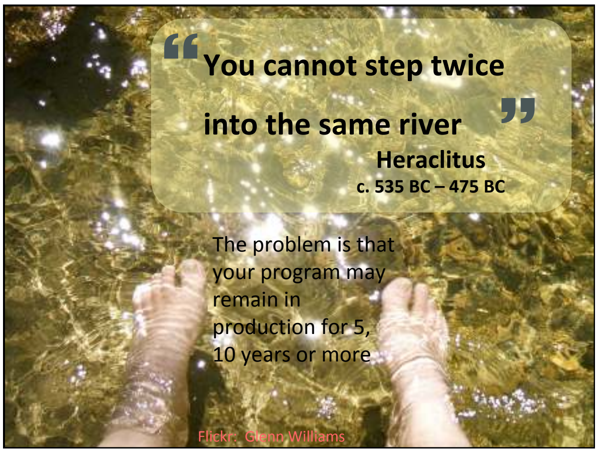


Hence "hints" that will direct the efforts of the optimizer and sometimes bring impressive improvement.

What you write

Much better

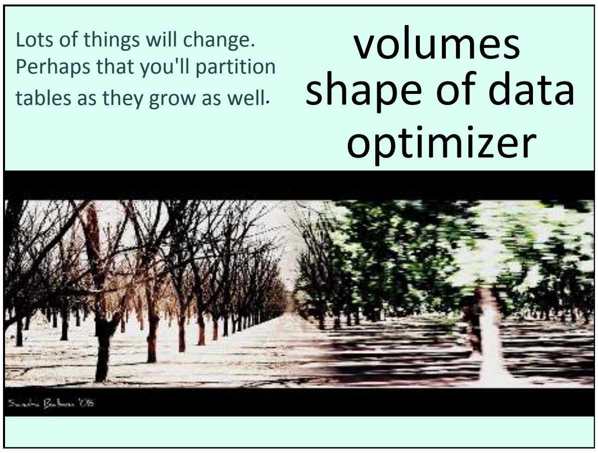
`/*+ hint */`



> You cannot step twice into the same river
>
> Heraclitus
> c. 535 BC – 475 BC

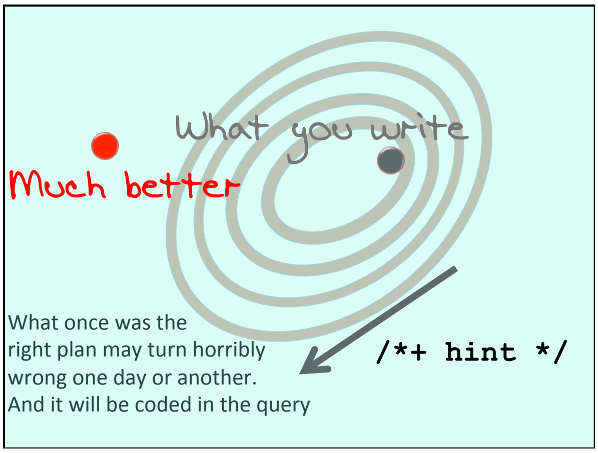
The problem is that your program may remain in production for 5, 10 years or more

Flickr: Glenn Williams

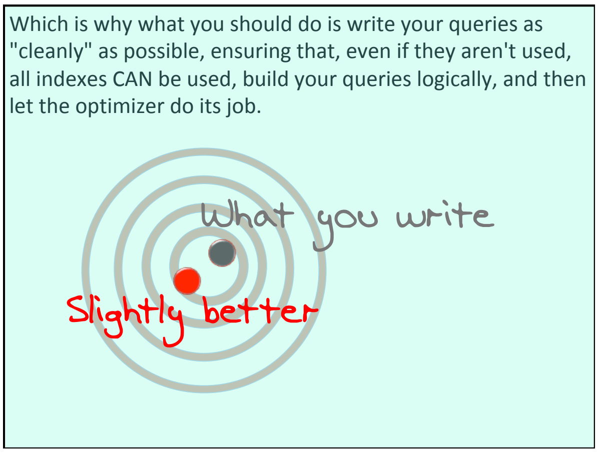


Lots of things will change. Perhaps that you'll partition tables as they grow as well.

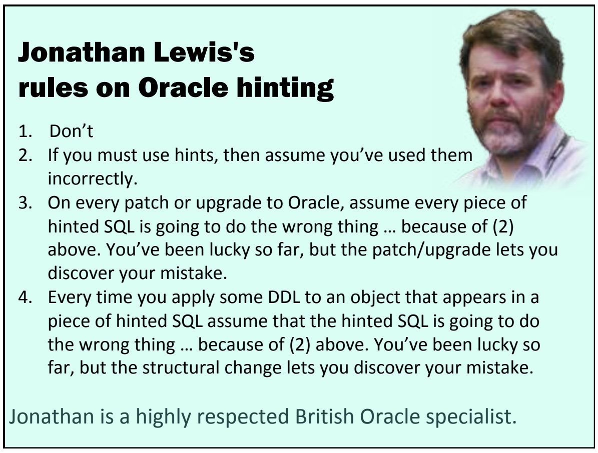
volumes
shape of data
optimizer



What once was the right plan may turn horribly wrong one day or another.
And it will be coded in the query



Which is why what you should do is write your queries as "cleanly" as possible, ensuring that, even if they aren't used, all indexes CAN be used, build your queries logically, and then let the optimizer do its job.



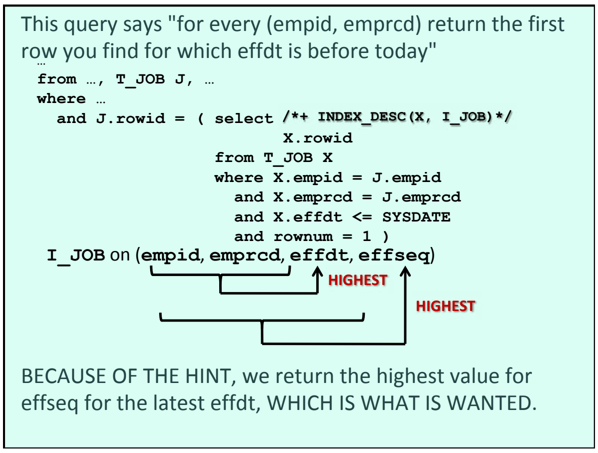
Jonathan Lewis's rules on Oracle hinting

1. Don't
2. If you must use hints, then assume you've used them incorrectly.
3. On every patch or upgrade to Oracle, assume every piece of hinted SQL is going to do the wrong thing … because of (2) above. You've been lucky so far, but the patch/upgrade lets you discover your mistake.
4. Every time you apply some DDL to an object that appears in a piece of hinted SQL assume that the hinted SQL is going to do the wrong thing … because of (2) above. You've been lucky so far, but the structural change lets you discover your mistake.

Jonathan is a highly respected British Oracle specialist.

A very, **VERY** bad usage of hints

Some people raise the misuse of hints to an art form.



This query says "for every (empid, emprcd) return the first row you find for which effdt is before today"

```
…
from …, T_JOB J, …
where …
  and J.rowid = ( select /*+ INDEX_DESC(X, I_JOB)*/
                         X.rowid
                  from T_JOB X
                  where X.empid = J.empid
                    and X.emprcd = J.emprcd
                    and X.effdt <= SYSDATE
                    and rownum = 1 )
```

I_JOB on (empid, emprcd, effdt, effseq)

HIGHEST
HIGHEST

BECAUSE OF THE HINT, we return the highest value for effseq for the latest effdt, WHICH IS WHAT IS WANTED.

What happens if:

Index renamed

Index redefined

Optimizer changes

?

Flickr: Tony Crider

Technically speaking, a hint is a comment. A missing index won't make the query fail.

Hint silently ignored

The query will still do as instructed: return one row for which effdt is before today. But will it be the highest effdt before today and the highest effseq value for that effdt?

Wrong result

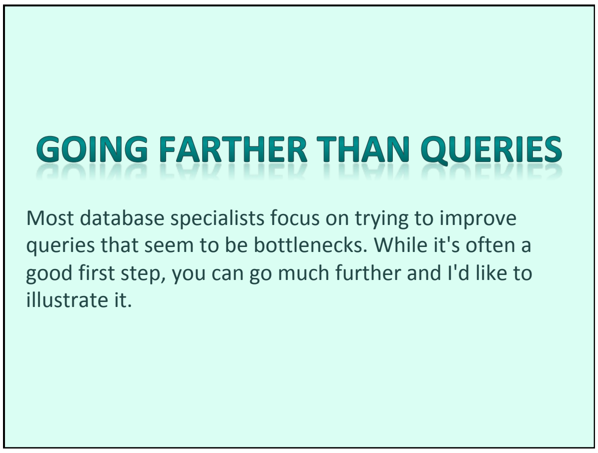
Or perhaps not. Who knows.

```
…
from …, T_JOB J, …
where …
  and J.effdt = (select max(X.effdt)
                 from T_JOB X
                 where X.empid = J.empid
                   and X.emprcd = J.emprcd
                   and X.effdt <= SYSDATE)
  and J.effseq = (select max(Y.effseq)
                  from T_JOB Y
                  where Y.empid = J.empid
                    and Y.emprcd = J.emprcd
                    and Y.effdt =  J.effdt)
```

This states exactly what we want; except that two correlated subqueries, one dependent on the result of the other, kill performances.

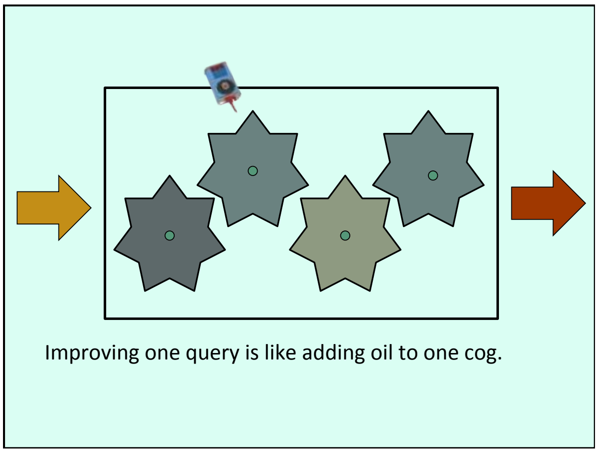
```
…
from …,
    (select …,
            rank() over (partition by empid,
                                      emprcd
                         order by effdt desc,
                                  effseq desc)
                                  rnk
      from T_JOB
      where effdt <= SYSDATE) J, …
where …
  and J.rnk = 1
```

But there are other, efficient ways to write it, which don't rely on side-effects to return the correct result.

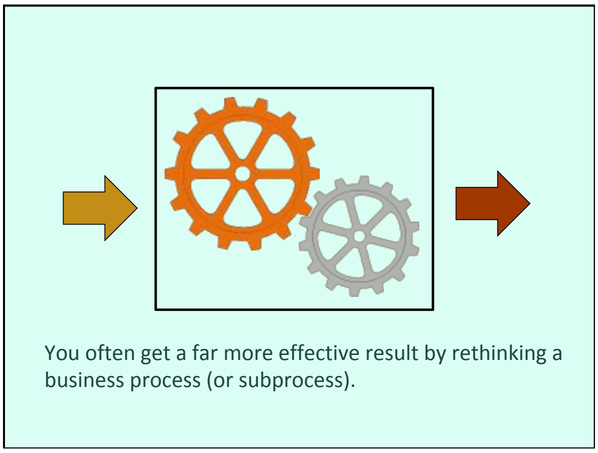


GOING FARTHER THAN QUERIES

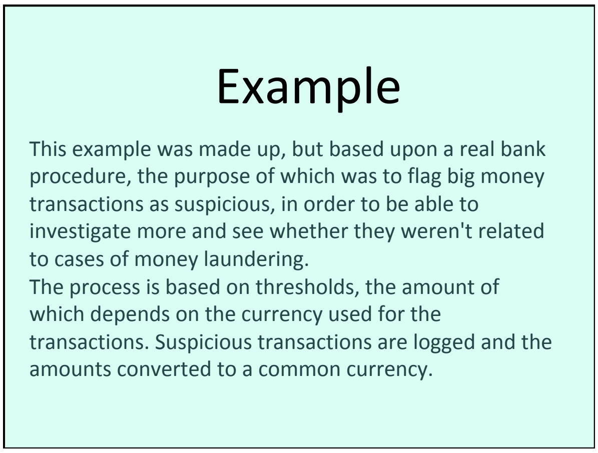
Most database specialists focus on trying to improve queries that seem to be bottlenecks. While it's often a good first step, you can go much further and I'd like to illustrate it.



Improving one query is like adding oil to one cog.

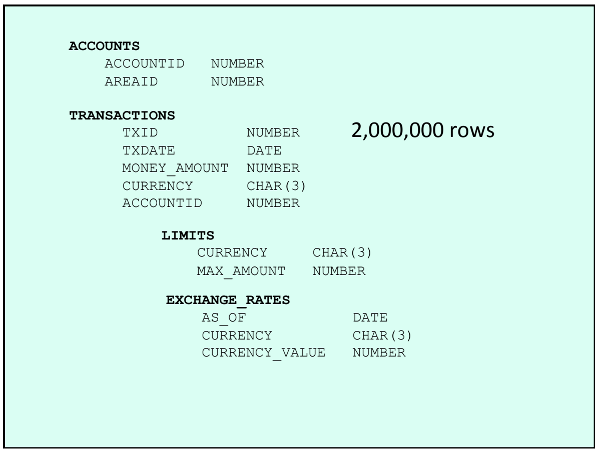


You often get a far more effective result by rethinking a business process (or subprocess).



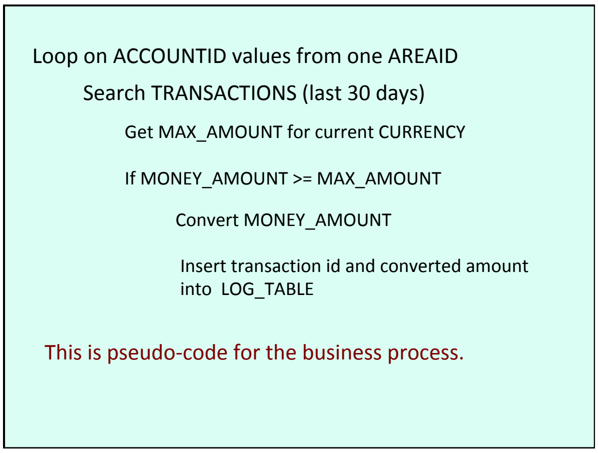
Example

This example was made up, but based upon a real bank procedure, the purpose of which was to flag big money transactions as suspicious, in order to be able to investigate more and see whether they weren't related to cases of money laundering.
The process is based on thresholds, the amount of which depends on the currency used for the transactions. Suspicious transactions are logged and the amounts converted to a common currency.



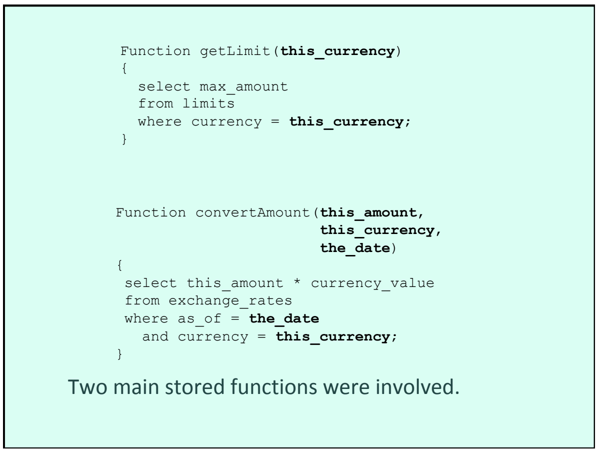
```
ACCOUNTS
     ACCOUNTID   NUMBER
     AREAID      NUMBER

TRANSACTIONS
       TXID             NUMBER
       TXDATE           DATE
       MONEY_AMOUNT     NUMBER
       CURRENCY         CHAR(3)
       ACCOUNTID        NUMBER

            LIMITS
                 CURRENCY      CHAR(3)
                 MAX_AMOUNT    NUMBER

             EXCHANGE_RATES
                  AS_OF               DATE
                  CURRENCY            CHAR(3)
                  CURRENCY_VALUE      NUMBER
```

2,000,000 rows



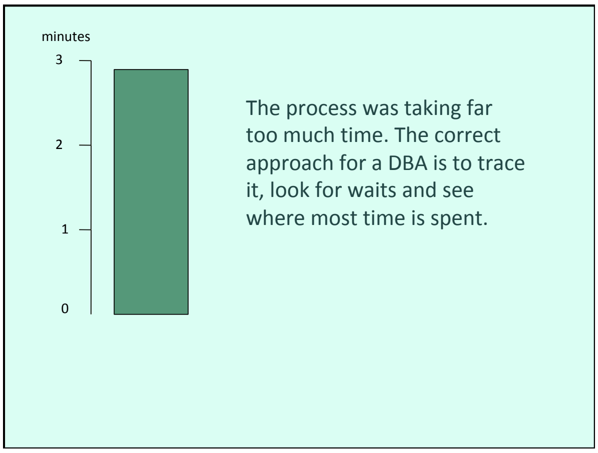
```
Loop on ACCOUNTID values from one AREAID
    Search TRANSACTIONS (last 30 days)
        Get MAX_AMOUNT for current CURRENCY
        If MONEY_AMOUNT >= MAX_AMOUNT
            Convert MONEY_AMOUNT
            Insert transaction id and converted amount
            into  LOG_TABLE
```

This is pseudo-code for the business process.

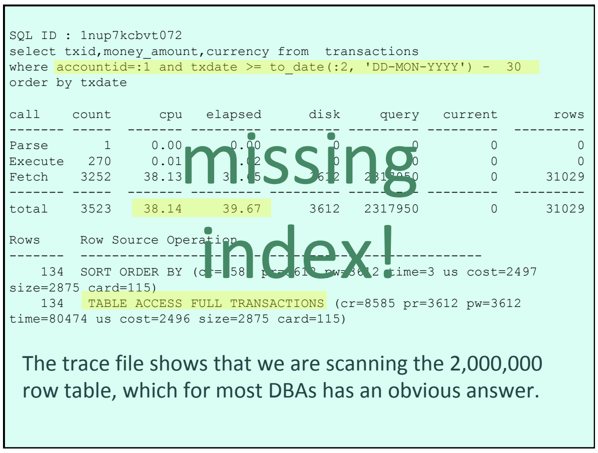


```
Function getLimit(this_currency)
{
  select max_amount
  from limits
  where currency = this_currency;
}


Function convertAmount(this_amount,
                       this_currency,
                       the_date)
{
 select this_amount * currency_value
 from exchange_rates
 where as_of = the_date
   and currency = this_currency;
}
```

Two main stored functions were involved.



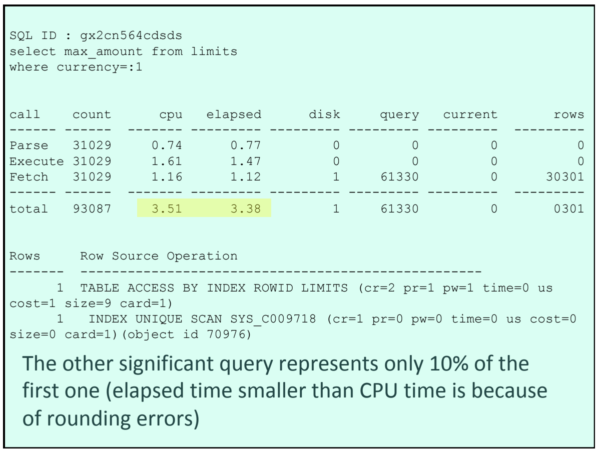
The process was taking far too much time. The correct approach for a DBA is to trace it, look for waits and see where most time is spent.



```
SQL ID : 1nup7kcbvt072
select txid,money_amount,currency from  transactions
where accountid=:1 and txdate >= to_date(:2, 'DD-MON-YYYY') -  30
order by txdate

call     count       cpu    elapsed       disk      query    current        rows
------- -----  ------- --------- --------- --------- ---------  ---------
Parse        1      0.00       0.00          0          0          0           0
Execute    270      0.01       0.02          0          0          0           0
Fetch     3252     38.13      39.65       3612    2317950          0       31029
------- -----  ------- --------- --------- --------- ---------  ---------
total     3523     38.14      39.67       3612    2317950          0       31029

Rows     Row Source Operation
-------  ---------------------------------------------------
    134  SORT ORDER BY (cr=8585 pr=3612 pw=3612 time=3 us cost=2497
size=2875 card=115)
    134   TABLE ACCESS FULL TRANSACTIONS (cr=8585 pr=3612 pw=3612
time=80474 us cost=2496 size=2875 card=115)
```

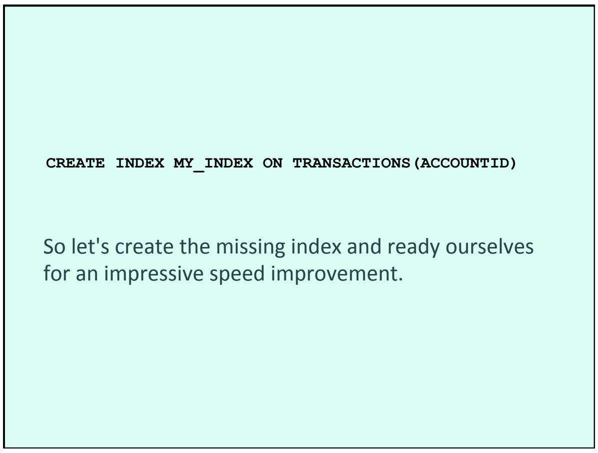
missing index!

The trace file shows that we are scanning the 2,000,000 row table, which for most DBAs has an obvious answer.



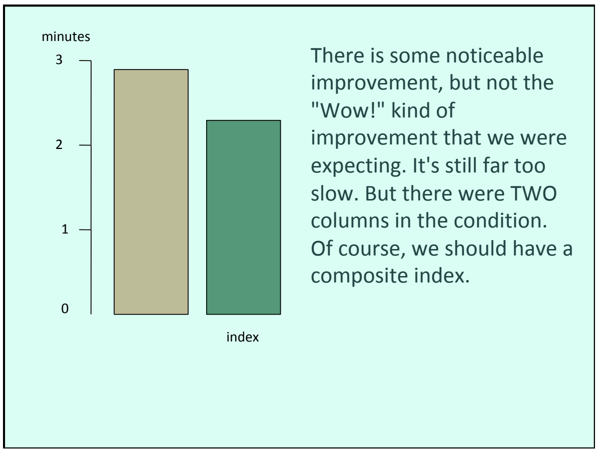
```
SQL ID : gx2cn564cdsds
select max_amount from limits
where currency=:1


call     count       cpu    elapsed       disk      query    current        rows
------ ------  ------- --------- --------- --------- ---------  ---------
Parse   31029      0.74       0.77          0          0          0           0
Execute 31029      1.61       1.47          0          0          0           0
Fetch   31029      1.16       1.12          1      61330          0       30301
------ ------  ------- --------- --------- --------- ---------  ---------
total   93087      3.51       3.38          1      61330          0        0301


Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS BY INDEX ROWID LIMITS (cr=2 pr=1 pw=1 time=0 us
cost=1 size=9 card=1)
      1   INDEX UNIQUE SCAN SYS_C009718 (cr=1 pr=0 pw=0 time=0 us cost=0
size=0 card=1)(object id 70976)
```

The other significant query represents only 10% of the first one (elapsed time smaller than CPU time is because of rounding errors)

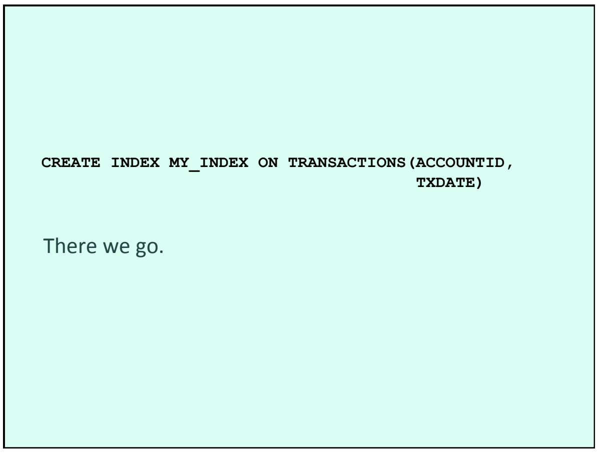


```
CREATE INDEX MY_INDEX ON TRANSACTIONS(ACCOUNTID)
```

So let's create the missing index and ready ourselves for an impressive speed improvement.

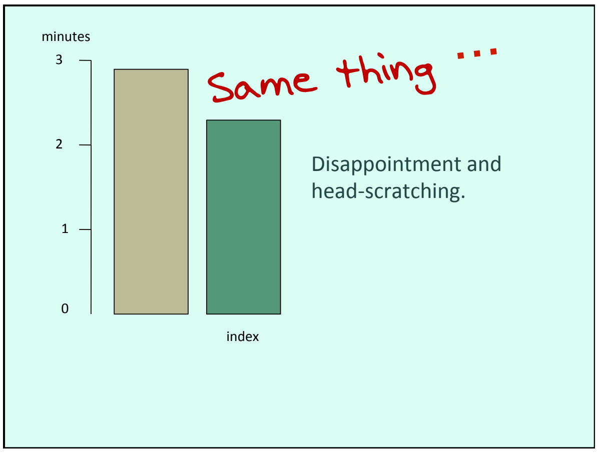


There is some noticeable improvement, but not the "Wow!" kind of improvement that we were expecting. It's still far too slow. But there were TWO columns in the condition. Of course, we should have a composite index.



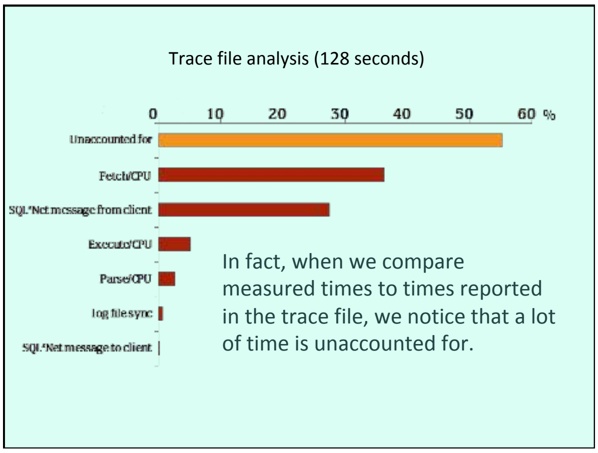
```
CREATE INDEX MY_INDEX ON TRANSACTIONS(ACCOUNTID,
                                      TXDATE)
```

There we go.



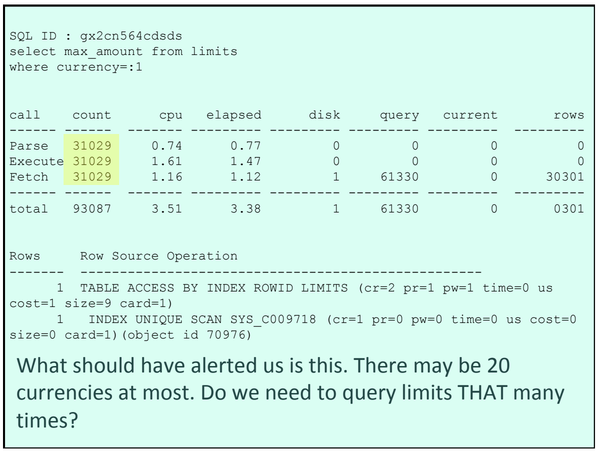
Same thing ...

Disappointment and head-scratching.



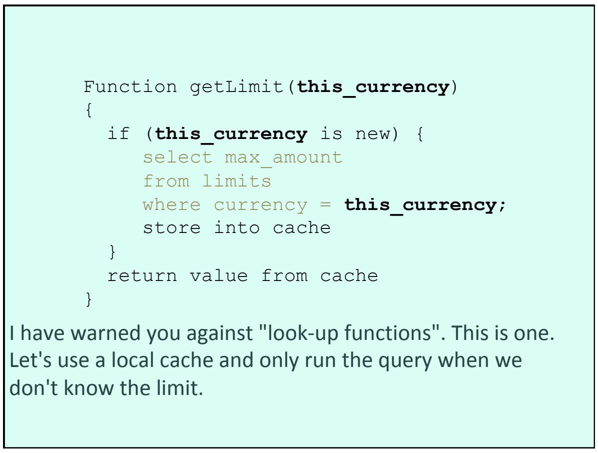
Trace file analysis (128 seconds)

In fact, when we compare measured times to times reported in the trace file, we notice that a lot of time is unaccounted for.



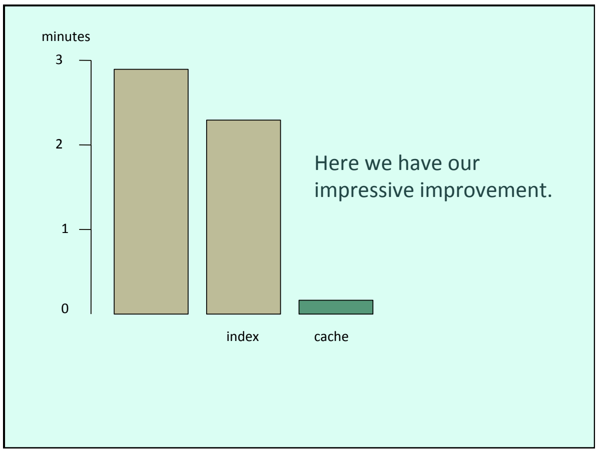
```
SQL ID : gx2cn564cdsds
select max_amount from limits
where currency=:1


call     count       cpu    elapsed       disk      query    current        rows
------ ------  ------- ---------- ---------- ---------- ----------  ----------
Parse    31029      0.74       0.77          0          0          0           0
Execute  31029      1.61       1.47          0          0          0           0
Fetch    31029      1.16       1.12          1      61330          0       30301
------ ------  ------- ---------- ---------- ---------- ----------  ----------
total    93087      3.51       3.38          1      61330          0        0301


Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS BY INDEX ROWID LIMITS (cr=2 pr=1 pw=1 time=0 us
cost=1 size=9 card=1)
      1   INDEX UNIQUE SCAN SYS_C009718 (cr=1 pr=0 pw=0 time=0 us cost=0
size=0 card=1)(object id 70976)
```

What should have alerted us is this. There may be 20 currencies at most. Do we need to query limits THAT many times?

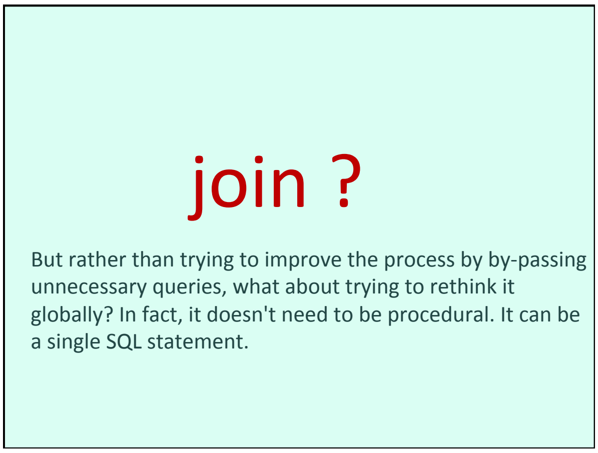


```
Function getLimit(this_currency)
{
  if (this_currency is new) {
     select max_amount
     from limits
     where currency = this_currency;
     store into cache
  }
  return value from cache
}
```

I have warned you against "look-up functions". This is one. Let's use a local cache and only run the query when we don't know the limit.

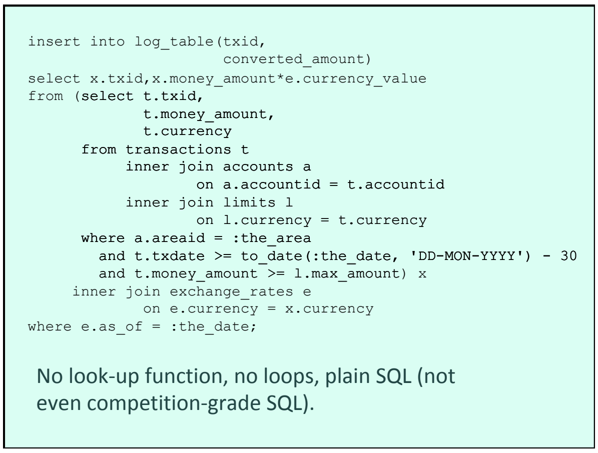


Here we have our impressive improvement.



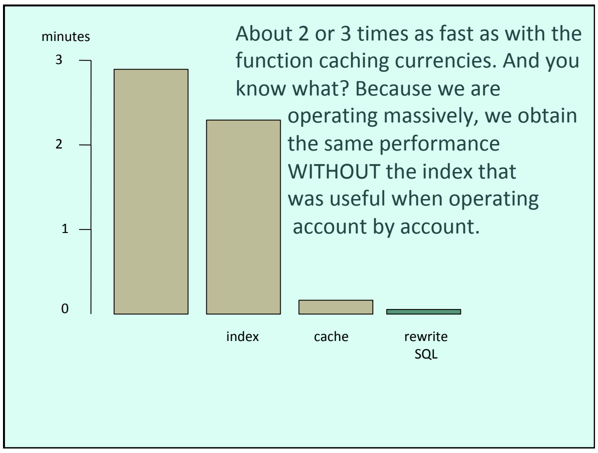
join ?

But rather than trying to improve the process by by-passing unnecessary queries, what about trying to rethink it globally? In fact, it doesn't need to be procedural. It can be a single SQL statement.



```
insert into log_table(txid,
                      converted_amount)
select x.txid,x.money_amount*e.currency_value
from (select t.txid,
             t.money_amount,
             t.currency
      from transactions t
           inner join accounts a
                   on a.accountid = t.accountid
           inner join limits l
                   on l.currency = t.currency
      where a.areaid = :the_area
        and t.txdate >= to_date(:the_date, 'DD-MON-YYYY') - 30
        and t.money_amount >= l.max_amount) x
     inner join exchange_rates e
             on e.currency = x.currency
where e.as_of = :the_date;
```

No look-up function, no loops, plain SQL (not even competition-grade SQL).



About 2 or 3 times as fast as with the function caching currencies. And you know what? Because we are operating massively, we obtain the same performance WITHOUT the index that was useful when operating account by account.

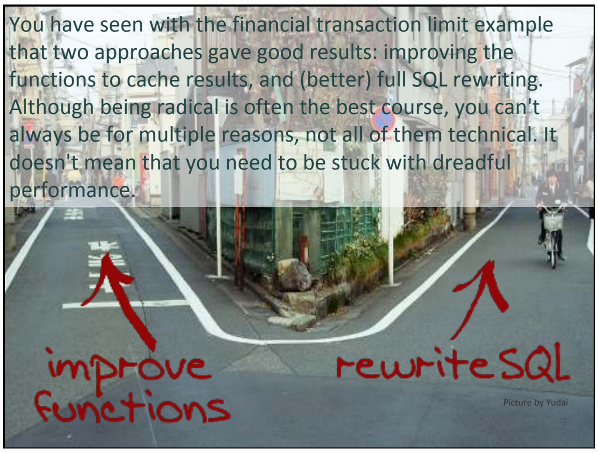
Lessons

Improving performance is not only adding indexes

Waits are useful for tuning, less so for refactoring

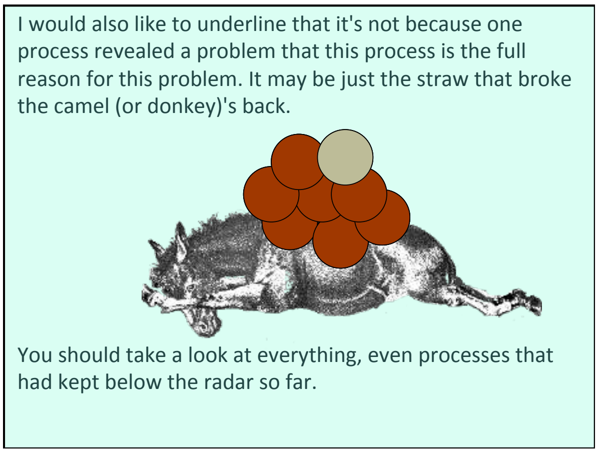
You can choose your approach

DON'T GET INTO THE WAY OF THE OPTIMIZER



You have seen with the financial transaction limit example that two approaches gave good results: improving the functions to cache results, and (better) full SQL rewriting. Although being radical is often the best course, you can't always be for multiple reasons, not all of them technical. It doesn't mean that you need to be stuck with dreadful performance.

Picture by Yudai



I would also like to underline that it's not because one process revealed a problem that this process is the full reason for this problem. It may be just the straw that broke the camel (or donkey)'s back.

You should take a look at everything, even processes that had kept below the radar so far.

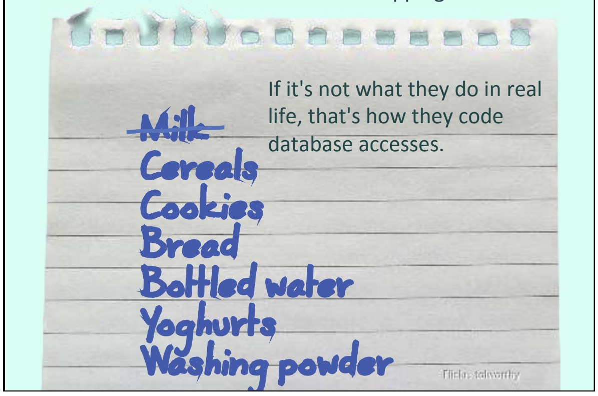
THE PROBLEM

The problem of many, many database applications can be explained with an analogy.

When they need to shop:

- Developers take their car
- Drive to the shop
- Look for a parking place
- Walk the aisles
- Queue for paying
- Look for their car
- Drive back home
- Store what they have bought

Then look for the next item on the shopping list.

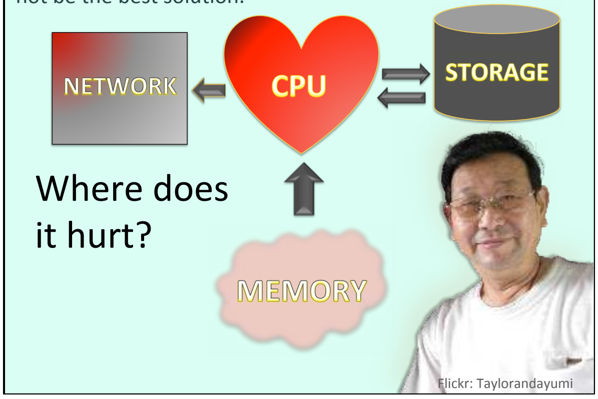


If it's not what they do in real life, that's how they code database accesses.

~~Milk~~
Cereals
Cookies
Bread
Bottled water
Yoghurts
Washing powder

Flickr: tehworthy

Interestingly, improving EVERY point where you waited may not be the best solution.



Where does it hurt?

Flickr: Taylorandayumi

A good DBA approach to performance issues

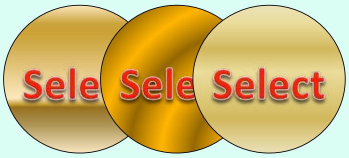
isn't

a good developer approach to performance issues.

You shouldn't be short-sighted in your approach and try to understand what the DBMS is getting and what it can do.

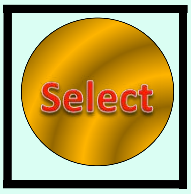
Seen by the DBMS

The DBMS only sees individual queries. It has no idea whatsoever about "flows".



Thousands of active users can be executing queries from the same program simultaneously.

Optimizer



Critically, the scope for the optimizer is ONE statement (and we have seen that optimizing, and getting stable performance, for one statement is hard enough)

```
for rec in (select deptno
            from emp
            where job = 'MANAGER')
loop
   select dname
   into v_deptname
   from dept
   where deptno = rec.deptno;
end loop
```

If you write your queries like this, the optimizer sees two independent queries that cannot be much improved. In effect, you are shoving a nested loop down the throat of the DBMS when a hash join might have been the smart way of retrieving data.

UNTUNABLE queries

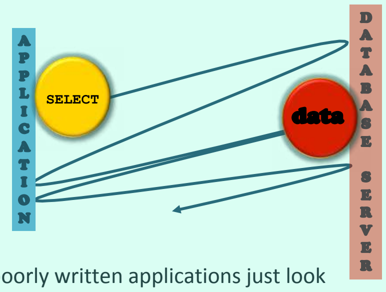
This is something that you see very, very often, and that Object Relational Mappers seem to be particularly apt at generating: tons of queries where the only search criterion is the primary key. The problem is, if that primary key is system-generated as it often is, how did you find its value in the first place? You are obviously running (at least) two queries when one could have done the job.

Successive statements

Another common mistake is when people execute in succession many statements that could have been run as one.

```
PROCEDURE p_hist_cust (p_txcode IN     customers.txcode%TYPE,
                       p_seqnum IN OUT customers.seqnum%TYPE)
IS
BEGIN
  /* Insert into the customer history table */
    INSERT INTO histcust
      SELECT * FROM customers
       WHERE customers.txcode = p_txcode;
  /* Change the fields in the history table */
    UPDATE histcust SET histcust.indact = 'N'
     WHERE histcust.txcode = p_txcode
       AND histcust.seqnum = p_seqnum;
  /* Assign a new sequence number */
    p_seqnum := NVL (p_seqnum,0) + 1;
  EXCEPTION
     WHEN others THEN
       util.p_raise_error( …);
END
```

You have here a real-life example. Why not inserting the right value directly instead of changing it afterwards? Can be done with a CASE ... END. Besides, select * is bad and trapping all errors dangerous.



Tons of poorly written applications just look like a tennis game between the application and the DBMS.
Bad enough locally, welcome to the cloud era.

Procedural logic

Far too few developers think in sets; and all the art of working with a database (and EVERYBODY is working with databases) is to switch from a procedural logic to a pure SQL logic.It's not because your procedural logic is embedded in a stored procedure that the sin is washed away.

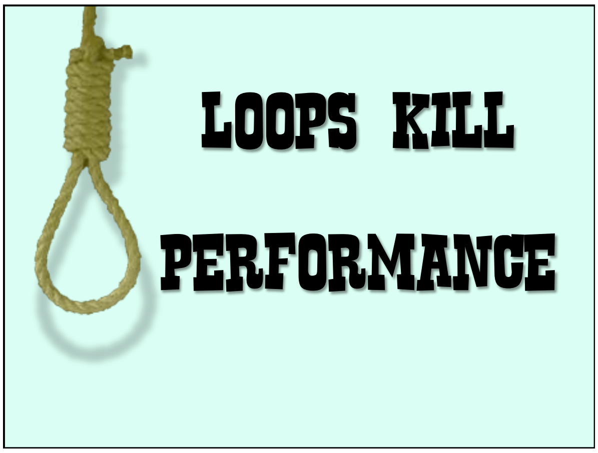
```
for rec in (select deptno
            from emp
            where job = 'MANAGER')
loop
   select dname
   into v_deptname
   from dept
   where deptno = rec.deptno;
end loop
```

I have given this example earlier. It may be a bit extreme but when you look carefully enough you meet the pattern very often. People love cursors. Almost every database programming course focuses on them.

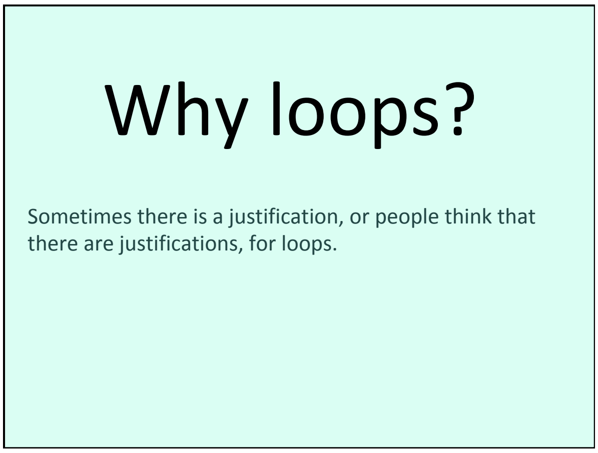
Nested loop
+ context switch / network latency

Is it the BEST we can do?

Loops are bad in a database environment. Very bad.

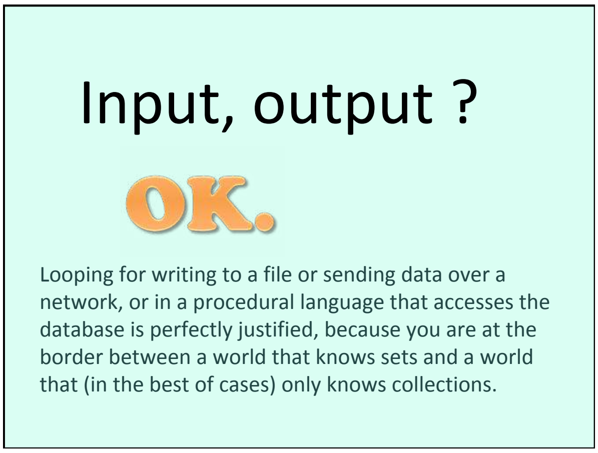


LOOPS KILL PERFORMANCE



Why loops?

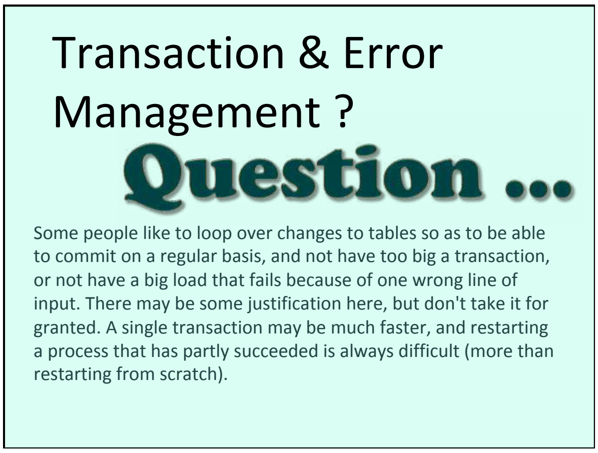
Sometimes there is a justification, or people think that there are justifications, for loops.



Input, output ?

OK.

Looping for writing to a file or sending data over a network, or in a procedural language that accesses the database is perfectly justified, because you are at the border between a world that knows sets and a world that (in the best of cases) only knows collections.



Transaction & Error Management ?

Question ...

Some people like to loop over changes to tables so as to be able to commit on a regular basis, and not have too big a transaction, or not have a big load that fails because of one wrong line of input. There may be some justification here, but don't take it for granted. A single transaction may be much faster, and restarting a process that has partly succeeded is always difficult (more than restarting from scratch).

`if … then …else` in the loop?

Try harder

There are also cases when people loop to be able to implement conditional logic in the loop. Cases when it's required are very rare. Most often, conditions can be reported inside the SQL statement with CASE ... END constructs and UNION ALL queries.

Otherwise …

No way!

These are about the only cases when the use of loops can (sometimes) be justified.

計利以聽，乃爲之勢，以佐其外。

Sun Tzu (6th century BC)
The Art of War, I,16

While heeding the profit of my counsel, avail yourself also of any helpful circumstances over and beyond the ordinary rules.