Queries and questions found on database forums on the Internet

Notes for the CS307 review

I search English or French languages sites, I'm sure that there is a lot of interesting cases to find on Chinese-language sites as well. The questions in this document are all real questions.

You usually find on forums tons of irrelevant answers, with many people willing to be helpful but not too knowledgeable themselves. From the people who advise to index every column to (a great favorite) people who are only concerned with execution plans (what the `explain` command returns). You can try `explain` on any of the queries we have done in labs, you usually get something that is far longer and far less legible than the original query. The real questions to ask when there is a problem with a query are what are volumes, what is indexed, and what are cardinalities - what is selective in a table. Not that execution plans are completely useless; they can sometimes reveal a trigger that nobody was aware of, or that what looked like a table was in fact a complex view (but all this can also be checked in the catalog). What an execution plan will show you is whether the index that you thought should have been used was used or not; the question is then "did finally the optimizer do a good choice?" and if not, why did it go wrong.

Test case 1

```
table forum :
- id
- thread (id of the first message)
- title
- msgdate

Wants to display the 15 last messages of the forum.
Size of the forum growing, performance issues.

Code:

SELECT f1.title,
       f2.thread,
       count(f2.id) as cnt,
       max(f2.msgdate) as last,
       max(f2.id) as last_id
FROM forum f1,
     forum f2
WHERE f1.id=f2.thread
GROUP BY f2.thread
ORDER BY last DESC
LIMIT 15
```

Comments

Note: this is obviously MySQL, every other product except SQLite would require **`title`** to be in the `GROUP BY` expression.

This query is using the old (but still valid) join syntax, and could be rewritten as

```
FROM forum f1
     join forum f2
     on f2.thread = f1.id
```

What is obvious is that there are no conditions at all, and that the query will scan `forum` (as `f1`) at every execution. From there, it must get the various rows for which the thread is the id of the current row. There is a 99% chance that the `id` column is a system-generated integer defined as primary key (`auto_increment` column in MySQL), and therefore indexed. However, when we search `forum` again as `f2`, the value of `id` isn't what we are looking for, but what we get as input, and we are looking for the corresponding values in `thread`. If column `thread` is NOT indexed, it means that not only we will be scanning `forum` once from the beginning to the end, but that for every line we'll have to scan it again to find the rows with a value in `thread` matching the current `id` value. If you see exactly what it means, you'll understand that performance will degrade very, very fast as the table grows, because the day when `forum` contains 10,000 rows we have to scan 100,000,000 rows before we start aggregating.

Let's assume that column `thread` is indexed. It's still pretty bad because as the table grows, the time required to scan it will grow proportionately, and the aggregate (that requires a sort) will only make matters worse.

The solution I would most easily imagine is to say that to find the 15 last posts we probably don't need to scan the forum since the first post 10 years ago. I would define a "reasonable time range", dependent on how active the forum is, based on how old the 15th most recently active thread is; if the 15th oldest most recent post is currently two weeks old, I'd say that I'm going back one month in time to have a decent safety margin.

So, to have the 15 most recent active threads, I would write something such as

```
select distinct thread
from forum
where msgdate >= suitable expression to compute current date minus one month
```

and use this query as `f1`. Note that I'm losing the title here, but as the id is the primary key I can join easily with table `forum` once more just to retrieve the title once I know what my 15 rows are. I must make sure that either the table is indexed on `msgdate`, or that the date is a partitioning key to avoid scanning the full table.

I can actually do better, and identify in my `f1` query what were the most recent active threads:

```
select thread, max(msgdate) as most_recent, max(id) as last_post
from forum
where msgdate >= suitable expression
```

```
group by thread
order by most_recent desc
limit 15
```

For counting the number of posts, though, I must still join to the full table because a recent post may refer to an old thread, so my query would read something like:

```
select f4.title, f3.thread, f3.cnt, f3.last, f3.last_id
from (select f1.thread, count(f2.id) as cnt,
             f1.most_recent as last,
             f1.last_post as last_id
      from (select thread, max(msgdate) as most_recent, max(id) as last_post
            from forum
            where msgdate >= suitable expression
            group by thread
            order by most_recent desc
            limit 15) f1
           join forum f2  -- to retrieve the count of the 15 threads
             on f2.thread = f1.thread
        group by f1.thread, f1.most_recent, f1.last_post) f3
     join forum f4 -- to retrieve the title of the 15 rows
       on f4.id = f3.thread
order by f3.last desc
```

This I think would be a robust solution that shouldn't degrade too much over time.

Alternatively, one of you suggested a solution that is sometimes used (one of the guys behind an Oracle-themed website called orafaq.com is a friend and he told me that this is what they were doing), which is to maintain (with triggers) a table where you store thread, last message date, last post id and the count of posts. Every insert into the main table updates the values. That means that you only have, at the expense of more painful inserts, to query this table (where the last message date can be indexed, which makes retrieval of the last 15 values easy, as keys are ordered in an index - but maintaining the index will of course cause more pain when inserting into the main table) to find the 15 last values, then to join with the main table to retrieve the titles. If you have far more people viewing than posting, it can make sense even if it's not as clean a solution as the first one suggested.

Test case 2

Given is a mySQL table named "orders_products" with the following relevant fields:

products_id
orders_id

Both fields are indexed.

I am running the following query:

```
SELECT products_id, count(products_id) AS counter
FROM orders_products
WHERE orders_id
IN (SELECT DISTINCT orders_id
    FROM orders_products
    WHERE products_id = 85094)
AND products_id != 85094
GROUP BY products_id
ORDER BY counter DESC
LIMIT 4
```

This query takes extremely long, around 20 seconds. The database is not very busy otherwise, and performs well on other queries.

I am wondering, what causes the query to be so slow?

The table is rather big (around 1,5 million rows, size around 210 mb), could this be a memory issue?

Comments

No, it's not a memory issue. Why should it be?

There is nothing obviously wrong in the query, nor with indexing from what is said. The problem here is with the MySQL optimizer that (at least in some past versions) is/was known to be bad with `IN ()` subqueries. In such a case, the only reasonable approach is to try to write the query in a different but logically equivalent way.

An uncorrelated subquery such as this one can be also written either as a join or a correlated subquery. When rewriting it as a join, the only thing to be careful about is that `IN ()` always performs a `DISTINCT` (which is explicit in this query, so here you can directly paste the query into a join). The query can easily be rewritten as

```
SELECT op.products_id, count(op.products_id) AS counter
FROM (SELECT DISTINCT orders_id
     FROM orders_products
     WHERE products_id = 85094) p
    join orders_products op
     on op.orders_id = p.orders_id
    and op.products_id != 85094
GROUP BY op.products_id
ORDER BY counter DESC
LIMIT 4
```

(it identifies the 4 products that are most commonly ordered at the same time as product 85094)

This should normally provide very decent response times, unless product 85094 appears in most orders.

The correlated subquery version would write:

```
SELECT a.products_id, count(a.products_id) AS counter
FROM orders_products a
WHERE exists
   (SELECT null
    FROM orders_products b
```

```
    WHERE b.products_id = 85094
      and b.orders_id = a.orders_id)
 AND a.products_id != 85094
GROUP BY a.products_id
ORDER BY counter DESC
LIMIT 4
```

In that particular case, the correlated subquery would probably be a bad idea, because it fires a query for every row that is inspected. As you can expect most rows NOT to contain product 85094, that might be quite a lot. A correlated subquery is good when you have roughly identified a set of candidate rows and want to perform the final screening.

Test case 3

You'll probably find the following query scary (in fact it IS scary) but don't believe that this is an extraordinary and isolated case. I have seen tons of queries that were as complicated, or worse. This query (from what I understand from the French column and table names) is a HR query to validate requests from employees in what is probably a very big company when they want to take a day off.

First, a DISTINCT at the very beginning of a complex query is always a bad sign. There are many legitimate uses of DISTINCT - you have seen some in the previous cases. But a legitimate DISTINCT is usually hidden deep inside a subquery, applied to a single table or perhaps a join between two tables and to few columns. A DISTINCT such as this one means in 99.999% of the cases a "quick fix" for duplicate rows that the developer had no idea how to get rid of otherwise. It's usually a symptom of a missing join condition somewhere, or missing deeply nested DISTINCT constructs (and of sloppy programming).

In the same way, I feel always suspicious about user-defined functions, as they often query the same tables as the query where they are called. User-defined functions applied to rows that are returned aren't the worst case, because they are only computed for rows that have been selected. They are a far greater evil in the WHERE clause, because there they may be call for a lot of rows that will never be returned and they can hurt far more.

Analyzing such a query in detail may require a full day, and I wouldn't be shocked by a couple of days of work to rewrite it properly. I will just focus on the shaded part of the query, indicated in the French comments as the main subquery that presumably drives everything else.

This is an Oracle query, easily recognizable by calls to functions that only exist in Oracle, and by a syntax with (+) after column names that is an old Oracle syntax for outer joins (the column followed by (+) is the one that is outer joined and replaced by null when no match can be found).

```
select distinct        -- infos sur la demande
       dem_abs.cd_dem_abs,dem_abs.dat_demande,
       gestion_cge.getrepart(dem_abs.cd_dem_abs,'Y') repart,
       --gestion_cge.ctrl_modif(dem_abs.cd_dem_abs, '', '', 0, 0, -1, 1)
act_status,
       gestion_cge.ctrl_modif(dem_abs.cd_dem_abs,
         dem_abs.dat_deb,
         dem_abs.dat_fin,
```

```
          substr(ddat_deb, 9),
          substr(ddat_fin, 9),
          c.cd_collab, -1) act_status,
        to_char(dem_abs.cd_collab) cd_collab,
        to_char(dem_abs.cd_etat) cd_etat,
        lib_etat_demande.lib_etat lib_etat_demande,
        dem_abs.dat_deb,
        demi_jour_deb.lib_court lib_demi_jour_deb,
        substr(ddat_deb, 9) cd_demi_jour_deb,
        dem_abs.dat_fin,
        demi_jour_fin.lib_court lib_demi_jour_fin,
        substr(ddat_fin, 9) cd_demi_jour_fin,
        nb_jours,
        to_char(dem_abs.cd_verif_rh) cd_verif_rh,
        lib_verif_rh,
        dem_abs.commentaire_rh,
        -- imputation : si non null on a une imputation paye dans au moins 1
absence
        to_char(dem_abs.mois_imputation,
                'dd/mm/yyyy') mois_imputation,
        to_char(dem_abs.dat_modif,
                'dd/mm/yyyy HH24:MI:SS') dat_modif,
        to_char(dem_abs.cd_util) cd_util,
        -- infos collab
        c.prenom_collab,
        c.nom_collab,
        etat_collab.commentaire comm_collab,
        c.info_tps_partiel,
        c.partiel,
        c.email,
        -- infos approb
        to_char(etat_approb.cd_etat) cd_etat_approb,
        lib_etat_approb.lib_etat lib_etat_approb,
        to_char(dem_abs.cd_collab_approb) cd_collab_approb,
        approb.nom_collab||' '||approb.prenom_collab approb,
        approb.email mail_approb,
        etat_approb.commentaire comm_approb,
        -- infos client (pour afficher le nom de l'affaire req à part)
        dem_abs.resp_client,
        dem_abs.accord_client,
        dem_abs.tel_client,
        to_char(dem_abs.cd_affaire) cd_affaire,
        -- infos valid
        to_char(etat_valid.cd_etat) cd_etat_valid,
        lib_etat_valid.lib_etat lib_etat_valid,
        to_char(dem_abs.cd_collab_valid) cd_collab_valid,
        valid.nom_collab||' '||valid.prenom_collab valid,
        valid.email mail_valid,
        etat_valid.commentaire comm_valid,
        etat_approb.dat_modif dat_approb,
        etat_valid.dat_modif dat_valid,
        tab_rh.nb modif_rh ,
        col_modif.modif_collab,
        decode(afa.ref_mission, null, to_char(afa.cd_affaire),
                                afa.ref_mission) ref_mission,
        afa.nom_affaire,
        cli.lib_client
from (select da.*,
             tab_max_ligne.cd_ligne_collab_max,
             tab_max_ligne.cd_ligne_approb_max,
```

```
        tab_max_ligne.cd_ligne_valid_max,
        tab_abs.nb_jours,
        tab_abs.ddat_deb,
        tab_abs.ddat_fin,
        tab_abs.dat_deb,
        tab_abs.dat_fin,
        tab_abs.mois_imputation
 from  -- Sélectionner les cd_ligne adéquats : SOUS-REQUETE PRINCIPALE
      (select da.cd_dem_abs,
              max(heda_c.cd_ligne) cd_ligne_collab_max,
              max(heda_a.cd_ligne) cd_ligne_approb_max,
              max(heda_v.cd_ligne) cd_ligne_valid_max
       from demande_absence da,
            histo_etat_dem_abs heda_c,
            histo_etat_dem_abs heda_a,
            histo_etat_dem_abs heda_v
       where heda_c.cd_type_util=1
         and heda_c.cd_collab=da.cd_collab
         and heda_c.cd_dem_abs=da.cd_dem_abs
         and heda_a.cd_type_util=2
         and heda_a.cd_dem_abs=da.cd_dem_abs
         and nvl(heda_a.cd_collab, 0)=nvl(da.cd_collab_approb, 0)
         and heda_v.cd_type_util=3
         and heda_v.cd_dem_abs=da.cd_dem_abs
         and heda_v.cd_collab=da.cd_collab_valid
         and (da.cd_collab=188
              or da.cd_collab_approb=188
              or da.cd_collab_valid=188)
       group by da.cd_dem_abs) tab_max_ligne,
      demande_absence da,
      (select a.cd_dem_abs,
              a.cd_collab,
              sum(nb_jours) nb_jours,
              max(dat_mois_paye) mois_imputation,
              min(to_char(dat_deb, 'YYYYMMDD')||cd_demi_jour_deb) ddat_deb,
              min(dat_deb) dat_deb,
              max(to_char(dat_fin, 'YYYYMMDD')||cd_demi_jour_fin) ddat_fin,
              max(dat_fin) dat_fin
       from absence a,
            demande_absence da
       where a.cd_dem_abs=da.cd_dem_abs
         and (da.cd_collab=188
              or da.cd_collab_approb=188
              or da.cd_collab_valid=188)
       group by a.cd_dem_abs,
                a.cd_collab) tab_abs
 where tab_max_ligne.cd_dem_abs=da.cd_dem_abs
   and tab_abs.cd_dem_abs=da.cd_dem_abs
   and tab_abs.cd_collab=da.cd_collab
  /* aj ici les critères limitatifs PHP */
   and (da.cd_collab<>188
        or da.cd_collab=cd_collab_approb
        or da.cd_collab=cd_collab_valid)
   and ((da.cd_etat in (1, 2, 3) and cd_collab_valid=188)
        or (da.cd_etat=1 and cd_collab_approb=188))) dem_abs,
(select he.cd_dem_abs,
        decode(sum(decode(he.cd_etat, 1, 1, 0))-1, 0, 'N', -1, 'N', 'O')
               as modif_collab
 from histo_etat_dem_abs he,
      demande_absence da
```

```
      where he.cd_type_util=1
        and he.cd_dem_abs=da.cd_dem_abs
        and (da.cd_collab=188
             or da.cd_collab_approb=188
             or da.cd_collab_valid=188)
      group by he.cd_dem_abs) col_modif,
     -- la demande a-t-elle été modifiée en dernier par les RH ? (4= profil
sicom RH)
      (select da.cd_dem_abs,
              count(*) nb
       from utilisateurs u,
            demande_absence da
       where da.cd_util=u.cd_collaborateur
         and da.cd_collab!=da.cd_util
         and cd_profil_utilisateur=4
         and (da.cd_collab=188
              or da.cd_collab_approb=188
              or da.cd_collab_valid=188)
       group by da.cd_dem_abs) tab_rh,
      histo_etat_dem_abs etat_approb,
      histo_etat_dem_abs etat_valid,
      histo_etat_dem_abs etat_collab,
      collaborateur c,
      collaborateur approb,
      collaborateur valid,
      etat_dem_abs lib_etat_approb,
      etat_dem_abs lib_etat_valid,
      etat_dem_abs lib_etat_demande,
      demi_jour_conge demi_jour_deb,
      demi_jour_conge demi_jour_fin,
      verif_rh,
      affaire afa,
      client cli
where tab_rh.cd_dem_abs(+)=dem_abs.cd_dem_abs
  and dem_abs.dat_deb > '01/01/2007'
  and col_modif.cd_dem_abs(+)=dem_abs.cd_dem_abs
  and afa.cd_affaire(+)=nvl(dem_abs.cd_affaire, 0)
  and afa.cd_client=cli.cd_client(+)
  and etat_approb.cd_dem_abs=dem_abs.cd_dem_abs
  and etat_approb.cd_type_util=2
  and approb.cd_collab(+)=nvl(etat_approb.cd_collab, 0)
  and etat_approb.cd_ligne=dem_abs.cd_ligne_approb_max
  and etat_valid.cd_dem_abs=dem_abs.cd_dem_abs
  and etat_valid.cd_type_util=3
  and valid.cd_collab=etat_valid.cd_collab
  and etat_valid.cd_ligne=dem_abs.cd_ligne_valid_max
  and etat_collab.cd_dem_abs=dem_abs.cd_dem_abs
  and c.cd_collab=etat_collab.cd_collab
  and etat_collab.cd_type_util=1
  and etat_collab.cd_ligne=dem_abs.cd_ligne_collab_max
  and substr(dem_abs.ddat_deb, 9)=demi_jour_deb.cd_demi_jour
  and substr(dem_abs.ddat_fin, 9)=demi_jour_fin.cd_demi_jour
  and dem_abs.cd_verif_rh=verif_rh.cd_verif_rh
  and lib_etat_demande.cd_etat=dem_abs.cd_etat
  and lib_etat_approb.cd_etat=etat_approb.cd_etat
  and lib_etat_valid.cd_etat=etat_valid.cd_etat
order by (sysdate-dat_deb) ASC,
         (sysdate-dat_demande) ASC
```

Comments

One noticeable feature of the core subquery is that it's a join between one table that appears at multiple places in the global query (`demande_absence`) - and three times the same table. The name of this second table starts with 'histo_', which suggests in French exactly the same thing as in English - historical data. Whenever you see "historical data", it usually means two things: 1) huge, and 2) it won't become any smaller as time goes by.

When you can, it's often good to limit the number of joins to the same table, that appears here under three different "identities", c, a, and v, all joined to da on `cd_dem_abs`, apparently differentiated by `cd_type_util` and a third condition that varies.

```
select da.cd_dem_abs,
       max(heda_c.cd_ligne) cd_ligne_collab_max,
       max(heda_a.cd_ligne) cd_ligne_approb_max,
       max(heda_v.cd_ligne) cd_ligne_valid_max
from demande_absence da,
     histo_etat_dem_abs heda_c,
     histo_etat_dem_abs heda_a,
     histo_etat_dem_abs heda_v
 where heda_c.cd_type_util=1
   and heda_c.cd_collab=da.cd_collab
   and heda_c.cd_dem_abs=da.cd_dem_abs
   and heda_a.cd_type_util=2
   and heda_a.cd_dem_abs=da.cd_dem_abs
   and nvl(heda_a.cd_collab, 0)=nvl(da.cd_collab_approb, 0)
   and heda_v.cd_type_util=3
   and heda_v.cd_dem_abs=da.cd_dem_abs
   and heda_v.cd_collab=da.cd_collab_valid
       and (da.cd_collab=188
            or da.cd_collab_approb=188
            or da.cd_collab_valid=188)
    group by da.cd_dem_abs
```

The idea is to first retrieve from the big table all the rows that are of interest to us, in a single pass:

```
select ..
from  demande_absence da
  join histo_etat_dem_abs heda
    on heda.cd_dem_abs=da.cd_dem_abs
   and ((heda.cd_type_util=1 and heda.cd_collab=da.cd_collab)  -- c
    or (heda.cd_type_util=2 and nvl(heda.cd_collab,0)=nvl(da.cd_collab_approb,0)) -- a
    or (heda.cd_type_util=3 and heda.cd_collab=da.cd_collab_valid)) --v
```

The select part is very easy

```
select da.cd_dem_abs,
       max(case heda.cd_type_util
            when 1 then heda.cd_ligne
            else null
           end) cd_ligne_collab_max,
       max(case heda.cd_type_util
            when 2 then heda.cd_ligne
            else null
           end) cd_ligne_approb_max,
```

```
max(case heda.cd_type_util
     when 3 then heda.cd_ligne
     else null
    end) cd_ligne_valid_max
```

I'm taking advantage of the fact that aggregate functions ignore null to only look in each aggregate to the rows I'm interested in.

I doubt very much that this alone will salvage the query, but this is a technique that can often be applied. It also has the advantage of showing better where the joins are occurring and where we should care about indexes. It's obvious that `cd_dem_abs` should be in first position of an index in the `heda` table. If not, this index should be created. A column such as `cd_type_util` probably requires no index, as cardinality must be very low. `cd_collab` is also a good candidate for indexing (possibly in the same index as `cd_dem_abs`) but the `nvl()` function (same as `coalesce()`) would kill its usage in one of the cases. Perhaps that the column could be made mandatory with a default value of 0, which would remove the need for the `nvl()`. This would of course have to be further studied and discussed with people who know the data.

As I have already pointed out, table `demande_absence` appears several times in the global query (at least five or six), in the aggregate but also elsewhere. It means that both aggregate and detail are required, which suggests using a window function, such as

```
max(case heda.cd_type_util
      when 1 then heda.cd_ligne
      else null
     end) over()
```

that would return the aggregate value with every row. One point to consider is how the rows from `demande_absence` are selected:

```
(da.cd_collab=188
 or da.cd_collab_approb=188
 or da.cd_collab_valid=188)
```

It would be interesting to know if any or all of these columns are indexed. If one of them ISN'T indexed, then there is no other way than scanning the table, and no index will be used. Rather than indexing everything, which can always badly impact inserts and updates, I would try to scan the table only once (you can scan hundreds of thousands of rows in les than half a second). I would use a common table expression (`with (...)`) to get the rows once and for all.

The syntax in this query is pretty old. It's quite possible that it was initially written 30 years ago, then more joins and columns were added as requirements were evolving. Obviously nobody ever attempted to use new features that might have been useful - until performance becomes unacceptable.

Test case 4

Two tables passenger and travelmate :

```
title= Mr, Mrs, Ms, Miss
passenger (client_id, title, surname, first_name,email ,tel
travelmate (tm_id, client_id, tm_title, tm_surname, tm_first_name, tm_birthyear)

Travelmate can be an adult or a child.
If adult, tm_title should be populated.
If child, tm_birthyear should be populated.

Any idea?
```

This is a case that many people would think of solving with a trigger, making insert or update fail if both `tm_title` and `tm_birthyear` are null or not null at the same time.

It's actually a design problem, and the proper way to "fix" constraints would be to have a "parent" table

`travelmate(tm_id, client_id, tm_surname, tm_firstname, tm_type)`

with `tm_type` being 'A' for adult and 'C' for child,

and two "child" tables:

```
adult_travelmate(tm_id, tm_title)
```

and

```
child_travelmate(tm_id, tm_birthyear)
```

The primary key of both tables remains `tm_id`, which is also a foreign key referencing `travelmate(tm_id)` (this kind of 'inheritance' model is about the only case where a relationship between tables with a cardinality (1:1) is fine). In such a model, all columns are made mandatory.

The downside is of course that instead of inserting into one table, one has to insert into two tables each time. However, don't forget that a trigger would have a cost too, and so it's not as bad in comparison as it may look.

On the plus side, one can expect many queries that will only need one smaller table (for instance, "average number of adults and children with each passenger" only requires the parent table, while "average age of children" only requires `child_travelmate`). If you remember that the cost of execution of a query is how many blocks or pages it hits (which simply means how many bytes have to be inspected and processed to get the result set), in such a situation we are hitting tables that may only contain rows that we want (only travelmates without much detail, only adults, only children), with far fewer columns, which means that every block will only contain rows we are interested in, and far more rows because they are smaller. So in the end we'll need much fewer blocks. Additionally, the model is easy to extend if one day you want to consider people travelling with pets, or consider separately travelmates requiring special assistance (wheelchair).