White Paper

embarcadero

Writing Optimal SQL

By Jonathan Lewis

Embarcadero Technologies

June 2010

This white paper is a companion piece to Embarcadero's live webcast event on June 10th, 2010. The slides for that event will be delivered to all LIVE attendees after the event. Much of the content contained in this white paper was first presented at the UKOUG Conference 2009 and is based on material from the "Writing Optimal SQL" full-day course that Jonathan Lewis has been presenting for the last five years.

Americas Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

INTRODUCTION

Before you can write efficient SQL you need to know what your data looks like and whether the storage mechanisms that you (or the database architect) have chosen for the data will allow you to find an efficient access path for the queries you want to write. It's also important to know whether the access paths that you think make sense are paths that the optimizer is actually able to choose.

Critically you have to know how much data you are going to collect from the database and whereabouts in the database you can find it. This means you need to be able to analyse the data by volume and by distribution – which is what the optimizer is trying to do, of course, when you use the ***dbms_stats*** package to collect statistics about the data.

Unfortunately the optimizer isn't always able to get a good picture of the data. It may work out (nearly correctly) that the "average" customer in your sales system places about 40 orders per year; on the other hand you may be aware that you don't have an "genuinely average" customer because 20 of your customers are responsible for 80% of your sales, 1,000 of your customers are responsible for the next 15% of your sales, and the remaining 80% of your customers are responsible for the last 5% of your sales.

Many of the bad plans generated by Oracle's query optimizer are the result of a difference of opinion between your knowledge of the data and Oracle's opinion of what the data looks like. (We won't pursue this point in this presentation, but sometimes the best way to get a good execution plan is to tell the optimizer what your data really looks like by writing little programs to create some "artificial but truthful" statistics, rather than using the ***dbms_stats*** package to gather statistics.)

GOOD INDEXING

If you know the data and understand what a given query is supposed to achieve you need to be sure that the optimizer can find an efficient path to your data. Typically this means ensuring you have a suitable set of indexes but designing a good set of indexes for each table is not a trivial problem, it requires a lot of knowledge of how the data will be used. (Indexing is not the only way to make sure that you can get at your data efficiently, of course, but after a system has gone into production it may be the only viable option for fixing performance problems,)

In the example in slide 9 we see a very simple query to list the definitions of all the indexes in the database. You could expand this query to include far more information, of course, such as the sizes (rows and blocks) of the table and its indexes, the number of distinct values and nulls in each column, the expressions used in

"function-based" indexes and so on. (There is a more sophisticated example – which could still be more refined - on my blog under https://jonathanlewis.wordpress.com/scripts). Even a query as simple as this, though, can highlight a number of problems with the indexing on a table – in particular problems that (a) waste resources and (b) give the optimizer an opportunity to choose the wrong index.

Foreign Keys: it is common knowledge that you need to **think about** creating indexes to match foreign key constraints if there is a threat of "foreign key locking" – but that doesn't mean that every foreign key has to be indexed, or that the indexes have to be exact matches to the constraint. If you never delete the parent row or update the parent key you don't need the index for reasons related of locking – although some foreign key indexes may be a good indexes in their own right for high-precision access.

If you have a foreign key that needs protection from the locking problem then the index need only ***start with*** the relevant columns; and if you create an index to protect a foreign key there's a fair chance that the leading column(s) of the index are very repetitive, so you need to consider using the ***"compress"*** option on the index to reduce its size. (Index compression may increase the CPU usage slightly but the benefit of reducing the size of the index – hence amount of I/O relating to the index – may be well worth the trade-off).

The site that supplied the example on slide 9 had a convention that foreign key index names include the letters FK (with PK for primary key and UK for unique key); so we see the "foreign key index" on (grp_id) is technically redundant because there is an index on (grp_id, role_id) that starts with the same column; similarly the indexes on (org_id) and (per_id) are technically redundant because there are indexes on (org_id, ap_id) and (per_id, ap_id) respectively. We do have to be just a little cautious about dropping the indexes though – looking at the on-site data I could see that two of them were hugely repetitious and so could probably be dropped very safely, but there was a possibility that the third (per_id) might be used in some cases where the optimizer would decide not to use the (per_id, ap_id) index because the larger leaf-block count or ***clustering_factor*** made it seem too expensive. There are cases where you might need to check the relative sizes of such statistics before dropping an index and then write a little script that adjusts the ***clustering_factor*** (in particular) of the larger index each time you collect stats on it.

We can't say that the index on (role_id) is redundant, because it isn't matched by another index and we don't know if there are any locking problems to worry about; however, we might know (or we can discover) that the ***role_id*** is very repetitive, so we can recreate the index as compressed. Revisiting the other indexes that made the "foreign key indexes" redundant we can also decide that they can be compressed – two of them on just the first column (the ap_id is unique), and the third on both columns.

The last index to look at is a function-based index on "last update date" – truncated to the day. There are several thoughts you might have on this index. First, any one date might identify a huge amount of data scattered widely across the table – is the index even used and useful; second, since any date identifies a lot of rows it's the type of index that would benefit from compression; third, as you update data rows are going to be deleted from the "low" end of the index and inserted at the "high" end – this index is probably going to be fairly empty for a huge fraction of its range and hold tightly packed data in the last few days, so it's probably going to worth using the ***coalesce*** command on this index every few days. (For more thoughts on this particular class of index, and the problems associated with it see the items on "Index Explosion" on my blog).

Let's face it. Both the database developer and database administrator's role within the application lifecycle is expanding. Database tools that enable collaboration and communication with management, QA, development and partners can help everyone succeed within this connected environment. For example, are you able to pinpoint and communicate problems in a way that can quickly get management, development and QA on the same page? How quickly can you generate a report you'll need to prove an SLA? Can you place projects under a central version control system with just a few mouse clicks? How quickly can you compare and communicate ongoing schema changes between development and production? How about reverse engineering an application that a line of business drops on you unexpectedly? When evaluating database tools, consider how each tool will help you collaborate and communicate with internal and external stakeholders.

KNOWING THE DATA

After giving the indexes a quick sanity check and finding (we hope) some indexes which might make it possible for your query to run efficiently, we need to look at the data – how much there is, and how much work you would have to do to get it if you used a particular index. If you don't already know what your data looks like there are several simple queries that can help you find out. "Simple" doesn't mean that they will be quick and cheap to run, of course.

Slide 10 is an example of a query to find out how many rows there are for each value in a given column – with examples of the various ***sample*** clauses you could use on a large table. This is immediately helpful for a column with just a few values (up to 254, if you're thinking of histograms), but not much use in my little demonstration case where there are 10,000 distinct values.

But we can refine the query – it tells us, for example, that the values 2 and 3 appear 12 times so we might ask how many other values appear 12 times, and that's what the query on slide 11 tells us. With the aggregated results we can see that the worst case example is a value with 22 rows and perhaps we ought to optimize our query on

the basis of that 22 rows (or at least write our query to protect against the potential threats of picking that specific row.)

But the volume of data is not the only thing you need to know – we can refine our query again to determine something about how scattered the data is. Instead of counting the number of rows per distinct value and then summing we count the number of **blocks** per value and sum. In this case we see that the worst *"scattering"* means that there are two values where I have to visit 19 different blocks to collect the data. (The ***index()*** hint is a valid way of optimising the query shown because ***colX*** is part of the primary key ***t1_pk.***)

We can do similar analysis with indexes. Slide 13 shows the query used by the ***dbms_stats*** package to collect index statistics for a simple B-tree index, and I've extracted the ***sys_op_lbid()*** function (lbid = leaf block id) to write a query that uses the same *"double aggregation"* method I applied to the column data – first we count the number of index entries in each leaf block, then aggregate blocks with the same number of entries to get a picture of the internal storage efficiency of the index – which gives us an idea of how many leaf blocks we will have to read if we use that index. (It also gives us some clues about the *"health"* of the index, especially if we know how big the index keys are and, therefore, how many rows there should be per block if we allow for the typical 70% utilisation with binomial distribution that you get from *"random arrival"* B-tree indexes. The "smashed" index in slide 15, for example, should give us a clue that something about our processing is leaving a lot of empty space in the index, and prompt us to think about why that might be happening and whether we need to do anything about it.

DRAWING THE PICTURE

Collecting information about the data is only the first step. You will need this information to help you work out a sensible path through the query but before you can apply the information you need a method for collating the query and the information you have collected. Inevitably it is hard to do something realistic on a Powerpoint slide so I've spread the detail over several slides and limited the content dramatically – in real-life you would start with a large sheet of paper and expect to make two or three attempts at sketching the query before you get a suitable layout.

My demonstration query joins five table with a three-table subquery. The requirement of the query is a realistic and potentially awkward one to meet. To turn this into a picture I have simply stepped through each table in the ***from*** clause, keeping an eye on the ***where*** clause at the same time, and drawing a box to represent each table, with a line representing a join between tables. For the subquery I've drawn a box with a dashed outline and created a picture of the subquery inside it, joining tables in the outer query to the tables in the subquery where there are correlation predicates. I've used *"crow's feet"* to represent the cardinality (one to one, one to many) of the

relationships between tables, and I've drawn and labelled an incoming arrow for each constant predicate on a table.

When complete, this diagram will also include details about any relevant indexes that exist (slide 18), the statistics about those indexes, and the statistics about the probably data volume and scatter (slide 19) and any operational information you might have (e.g. likely caching benefits) – but that's too much information for one slide. In many cases it's also more information than you will actually need to write out, especially if you are already very familiar with the data.

Once you've got all this information in a tidy graphical presentation you can pick a table (more or less at random, if you want to) and ask the following questions:

- how much data do I get if I start here,
- how do I collect that data (index or tablescan); and from that point onwards you keep asking the same three questions:
- which table do I visit next
- how do I get there (based on the information I have so far)
- how much data do I end up with

The ideal is to find a starting point which returns a small amount of data very cheaply and then visit the rest of the tables in an order that makes each visit cheap and keeps the result set small. To demonstrate the approach, let's consider the picture of the query and start on the ***suppliers*** table from the main query (ignoring the path presented on slide 21).

We start at ***suppliers*** because we know that we will collect just a few suppliers from Leeds and we can do it efficiently because we have a suitable index on the table. The next obvious place to go from ***suppliers*** is to the ***products*** they supply – there is an index to do this, and possibly the Leeds suppliers don't supply many different products in total, so the intermediate result set is still small.

We now have a choice, we could go to the ***order_lines*** table either through the foreign key index or by hash join and tablescan – but that choice will have to acquire a lot of rows that will be extremely randomly scattered through a very large table, so we decide to move into the subquery instead as this will (eventually) reduce the number of products before we go to the order lines. (Strategy note – given the choice, a join that reduces the volume of the data set will usually be better than a join that increases the volume of the data set, unless the join happens to be very expensive to operate at this step of the query).

So we go to the ***product_match*** table (which may increase or decrease the result set – maybe there are only a few products with alternative sources, maybe every product has a couple of alternatives), to the asubquery appearance of ***products***

table which grows the length of the rows in the intermediate result but not the number, to the ***suppliers*** table which allows us (probably) to all rows rows where the supplier is not from Leeds – perhaps most of the data we have collected so far.

Then we're back to the point where we have to visit ***order_lines*** table. It's possible, of course, that our knowledge of the data tells us that there are virtually no products which match the Leeds/Not Leeds requirements, and that hardly anyone ever buys them; so it's possible that the random walk through the ***order_lines*** table would be cheap enough to keep us happy at this point – but in a generic sales system we probably wouldn't want to do this join.
Having reviewed (a limited set of) the effects of starting with the main ***suppliers*** table let's consider the path shown in slide 21. We want all the orders in the last week from customers in London. In our system recent orders are the ones that are mostly likely to stay cached, and we have an index into ***orders*** by date so, even though we will pick up far more data than we really want, starting with ***orders*** will probably be quite efficient. We then need to discard redundant data as soon as possible, so we next join to ***customers*** to eliminate all but the London customers. From there we have to go to ***order_lines***, but because we are coming from *"recent orders"* the order lines we want are also likely to be well cached and well clustered so, even though this may be quite a lot of data, we should still be able to acquire it efficiently. The argument about the ordering of the rest of the tables is left as an exercise to the reader.

It's important to note two variations here. First our knowledge of the caching and clustering of the ***orders*** and ***order_lines*** tables may be better than Oracle's, and it is this difference of opinion that may make Oracle choose a brute-force path rather than trying for the precision path we have identified; so we may have to hint the code, although we may find that it is sufficient (and generally more sensible) to correct the ***clustering_factor*** on the ***orders*** and ***order_lines*** indexes. Secondly, if we had an existing index (***id_customer, date_ordered***) on the ***orders*** table we might have chosen to start at the ***customers*** table and follow that index into the ***orders*** table because of the increased precision it would have given us; so this is a case where we might raise a flag to review the indexing strategy.

CASE STUDY

The example starting at slide 22 is a query (with camouflage) that came from an OLTP system that needed some performance improvements with minimum structural or code changes. It's an odd looking query, until you realise that its purpose is to provide a drop-down menu of contracts that a person is allowed to see as they tab across a screen into a particular field.

Checking the AWR report for the system, I found that the optimizer had generated 11 different execution plans for this query over the previous seven days – none of them particularly efficient. (The main problem in almost all cases was the tablescan of the ***transactions*** table.)

After creating the sketch of the query and filling in relevant information a couple of critical points stand out. There were 240 offices of different sizes – which is reflected in the variation in number of contracts per office – so Oracle's automatic stats collection job had generated a frequency histogram on the ***id_office*** column of the ***contracts*** table; thanks to bind variable peeking this means the plan could vary dramatically from day to day depending on who happened to be the first person to run the query in the morning.

The second critical piece of information (which was not available to the optimizer, of course) was that the standard query was for *"what happened since yesterday morning"* – which means the data that people want from the ***transactions*** table is only the most recent slice, which will end up being very well cached. So even if we start by visiting a lot of redundant transaction data, we won't be doing a lot of disk I/O to find it.

Another point to consider (and again this is not information that is available to the optimizer) is that the number of contracts that an office has handled will always increase with time – but the number of transactions per day is (virtually) unchanging because older contracts stop generating transactions.

So our first solution is: get rid of the histogram that's destabilising the system, and take the path ***transactions -> contracts -> transaction_types***. This starts with far more data than most users need and does an unreasonable amount of logical I/O, but it runs sufficiently swiftly and consistently to keep everyone happy.

There was a lot of resistance to changing indexes, but various changes in indexing could make this path more efficient, and could event introduce a better path.

Option 1 was to add the ***office_id*** column to the existing (primary key) index on the contracts table (***id***). This would give us an index-only path to collect the ***contracts*** infomration. This change would relatively low risk (compared to most index changes) because the (***id***) index was unique and single column – but it still needed careful testing for unexpected side effects in other queries.

Option 2 was to extend the (***created***) index on ***transactions*** to add into the index all the columns from the table that we needed so that we didn't have to visit the table at all – a change that would make a dramatic difference to buffer visits – but which might have more impact on other queries.

In the course of considering options, we considered and immediately rejected another path – starting and the contracts table and then accessing the transactions table, and enhancing indexes to make this as efficient as possible. But this is a non-scalable solution. Offices collect new contracts as time passes, and never drop them, so the work of identifying contracts and probing for (often non-existent) matching transactions in the past 24 hours would increase with time. To make this solution scalable we would need to add a *"contract terminated"*, or *"last transaction date"* column to the ***contracts*** table and include it in the driving index for the query.

At the end of the day, the client chose the safest course – no index changes, just hint the code and drop the histogram. It wasn't the fastest option, but it was fast enough and a stable solution because it depended on the (totally consistent) number of transactions per day, rather than a variable, and growing, quantity like the number of transactions per office.

CONCLUSION

When the optimizer fails to find a good execution plan for a query it's usually because its model of your data doesn't match reality. To solve such problems you need to do the job the optimizer is trying to do, and you need a mechanism that lets you do it easily. A graphical approach makes it easy to grasp the *"shape"* of a query and also allows you to present a large amount of information about the query in a very readable form.

If you can present the query, statistics and indexes in a readily comprehensible fashion it becomes much easier to recognise the best available execution path and decide if there are any structural changes (such as new or modified indexes) that should be introduced to allow a better path to become available.

ABOUT THE AUTHOR

Jonathan Lewis has been working in the IT industry for nearly 25 years, and has been using the Oracle RDBMS for more than 20. For the last 16 years he has been working as a freelance consultant, often spending only one or two days at a time with any client to address critical performance problems. Jonathan is also renowned throughout the world (having visited 42 countries at the last count) for his tutorials and seminars. He was a founding member of the Oak Table network, and one of the first individuals to be contacted by Oracle University for their "Celebrity Seminar" events. He also writes regularly for the UKOUG magazine, and occasionally for other publications around the world. In the limited amount of time he has left over, Jonathan also publishes high-tech Oracle articles on his blog at jonathanlewis.wordpress.com.

Embarcadero Technologies, Inc. is the leading provider of software tools that empower application developers and data management professionals to design, build, and run applications and databases more efficiently in heterogeneous IT environments. Over 90 of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero's award-winning products to optimize costs, streamline compliance, and accelerate development and innovation. Founded in 1993, Embarcadero is headquartered in San Francisco with offices located around the world. Embarcadero is online at www.embarcadero.com.

How to …
.. design efficient SQL

Jonathan Lewis
www.jlcomp.demon.co.uk
jonathanlewis.wordpress.com

Who am I ?

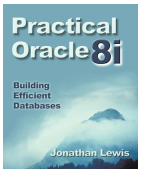
Independent Consultant.

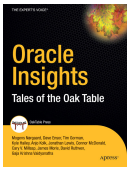
25+ years in IT
22+ using Oracle

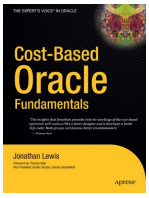
Strategy, Design, Review
Briefings, Seminars
Trouble-shooting

Jonathanlewis.wordpress.com
www.jlcomp.demon.co.uk

Member of the Oak Table Network.
Oracle author of the year 2006
Select Editor's choice 2007
Oracle *ACE Director*







Highlights

Know the data

Does a good execution path exist ?

Can the optimizer find that path ?

Knowing the data

How much data?

Where is it ?

Knowing the data - conflict

Your knowledge of the data

The optimizer's model of the data

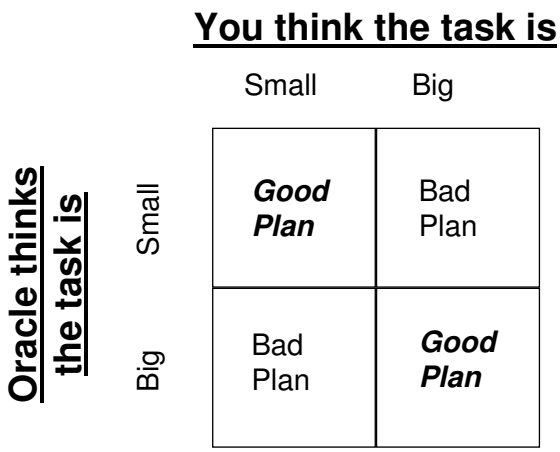
Strategy Choice

Lots of little jobs

- How many
- How little (how precise)

One big job

Common Outcomes



Optimizer problems

Correlated columns

Uneven data distribution

Aggregate subqueries

Non-equality joins

Bind variables

Know the metadata

```
select
       table_owner,
       table_name,
       index_name,
       column_name
from
       dba_ind_columns
order by
       table_owner,
       table_name,
       index_name,
       column_position
;
```

~~AP_GRP_FK_I~~	~~GRP_ID~~	
AP_GRP_ROLE_I	GRP_ID ROLE_ID	(compress)
AP_ORG_AP_I	ORG_ID AP_ID	(compress 1)
~~AP_ORG_FK_I~~	~~ORG_ID~~	
AP_PER_AP_I	PER_ID AP_ID	(compress 1)
~~AP_PER_FK_I~~	~~PER_ID~~	
AP_PK	AP_ID	
AP_ROLE_FK_I	ROLE_ID	(compress)
AP_UD_I	TRUNC(UPD_DATE)	(drop, compress, coalesce)

Know the data (a)

```
select
       colX,
       count(*) ct
from   t1
group by colX
order by colX
;
```

```
sample (5)
sample block (5)
sample block (5, 2)          -- 9i
sample block (5, 2) seed(N) -- 10g
```

COLX	CT
1	9
2	12
3	12
4	8
5	7
6	9
...	
...	
9997	1
9998	1
9999	1
10000	1

Know the data (b)

```
select
        ct, count(*)
from    (
        select
                colX,
                count(*) ct
        from    t1
        group by colX
        )
group by ct
order by ct
;
```

CT	COUNT(*)
1	9001
...	
6	37
7	78
8	94
9	117
10	112
11	126
12	99
13	97
14	86
15	49
16	32
...	
22	1

There are 99 values that appear 12 times

Know the data (c)

```
select
        blocks, count(*)
from    (
        select
                /*+ index(t1 t1_pk) */
                colX,
                count(
                    distinct substr(rowid,1,15)
                )       blocks
        from    t1
        group by colX
    )
group by blocks
order by blocks
;
```

BLOCKS	COUNT(*)
1	9001
...	
6	43
7	83
8	107
9	126
10	120
11	125
12	119
13	90
14	69
15	42
16	28
...	
19	2

There are 90 values that are scattered across 13 blocks

Know the data (d)

```
select /*+ index(t,"T1_I1") */
       count(*)                                          nrw,
       count(distinct sys_op_lbid(49721,'L',t.rowid)) nlb,
       count(distinct hextoraw(
              sys_op_descend("DATE_ORD") ||
              sys_op_descend("SEQ_ORD")
       ))                                                ndk,
       sys_op_countchg(substrb(t.rowid,1,15),1)       clf
from
       "TEST_USER"."T1"  t
where
       "DATE_ORD" is not null
   or  "SEQ_ORD" is not null
;
```

www.apress.com/resource/bookfile/2410

Know the data (e)

```
select
       keys_per_leaf, count(*) blocks
from   (
       select
              sys_op_lbid(49721,'l',t.rowid)       block_id,
              count(*)                             keys_per_leaf
       from
              t1  t
       where
              {index_columns are not all null}
       group by
              sys_op_lbid(49721,'l',t.rowid)
       )
group by       keys_per_leaf
order by       keys_per_leaf
;
```

Know the data (f)

Smashed Index

KEYS_PER_LEAF	BLOCKS
3	114
4	39
6	38
7	39
13	37
14	1
21	1
27	15
28	3
39	1
54	6
55	3
244	1
281	1
326	8

Normal Index

KEYS_PER_LEAF	BLOCKS
17	206
18	373
19	516
20	678
21	830
22	979
23	1,094
24	1,178
25	1,201
26	1,274
27	1,252
28	1,120
29	1,077
30	980
31	934
32	893
33	809
34	751
35	640
36	738
37	625
38	570
39	539
40	489

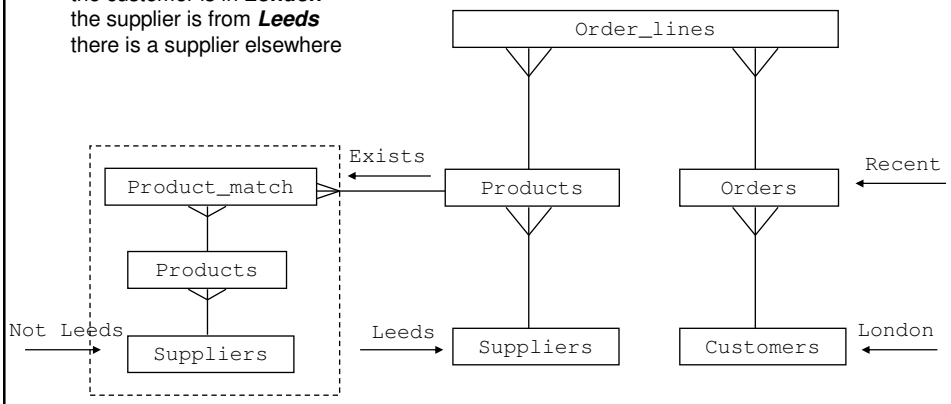
Draw the query - requirement

```
select   {columns}
from     customers        cus,
         orders           ord
         order_lines      orl
         products         prd1
         suppliers        sup1
where    cus.location     = 'LONDON'
and      ord.id_customer  = cus.id
and      ord.date_placed  between sysdate - 7 and sysdate
and      orl.id_order     = ord.id
and      prd1.id          = orl.id_product
and      sup1.id          = prd1.id_supplier
and      sup1.location    = 'LEEDS'
and      exists (
                  select   null
                  from     product_match    mch,
                           products         prd2,
                           suppliers        sup2
                  where    mch.id_product   = prd1.id
                  and      prd2.id          = mch.id_product_sub
                  and      sup2.id          = prd2.id_supplier
                  and      sup2.location != 'LEEDS'
         )
```

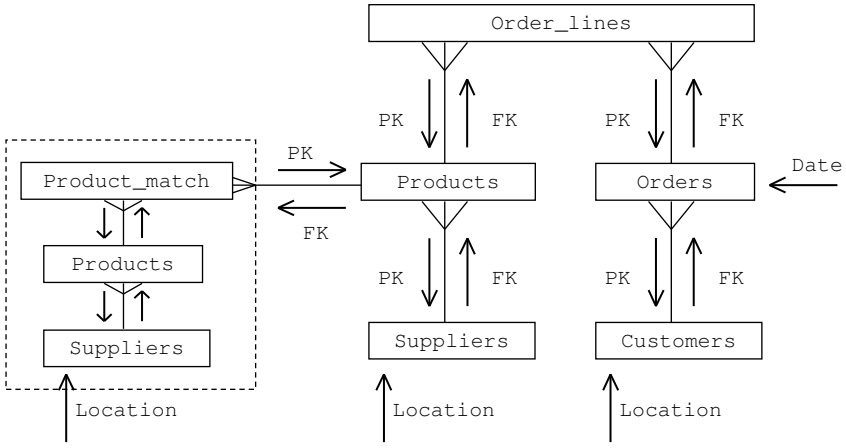
Orders in the ***last week*** where
the customer is in ***London***
the supplier is from ***Leeds***
there is a supplier elsewhere

Draw the query - outline

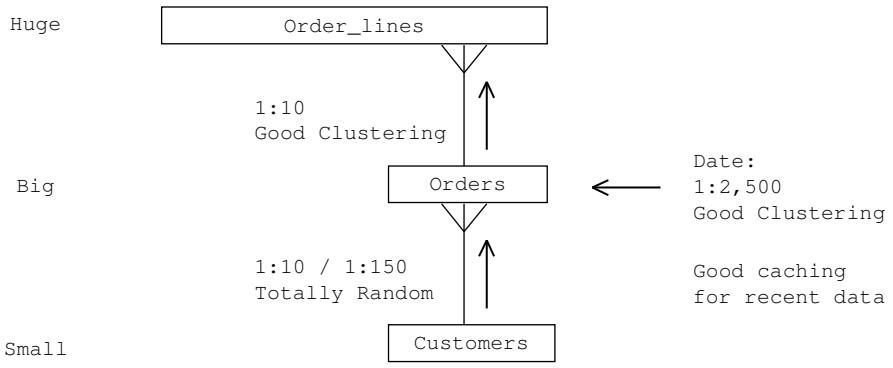
Orders in the ***last week*** where
the customer is in ***London***
the supplier is from ***Leeds***
there is a supplier elsewhere



Draw the query - indexes



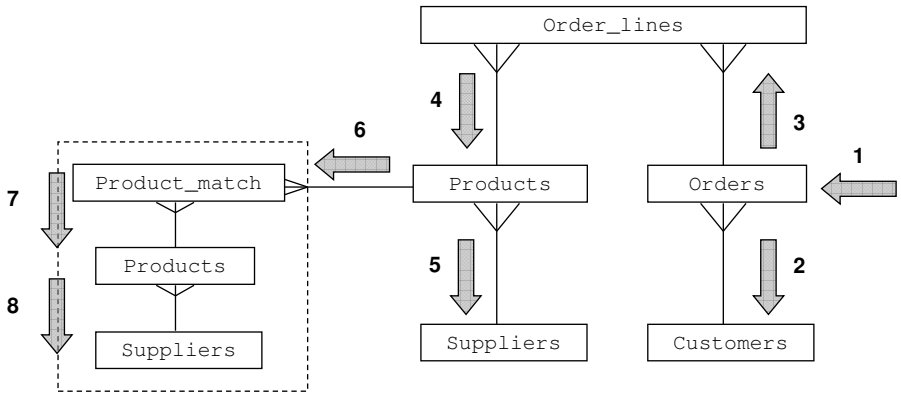
Draw the query - statistics



Sketch in paths - strategy

- Pick a starting point
  - How many rows will I start with
  - How efficiently can I get them
  - the first step may be inefficient (it only happens once)
- How do I get to next table
  - How many times do I make the step
  - How precise is the access path
  - How much data do I now have

Sketch in paths - analysis



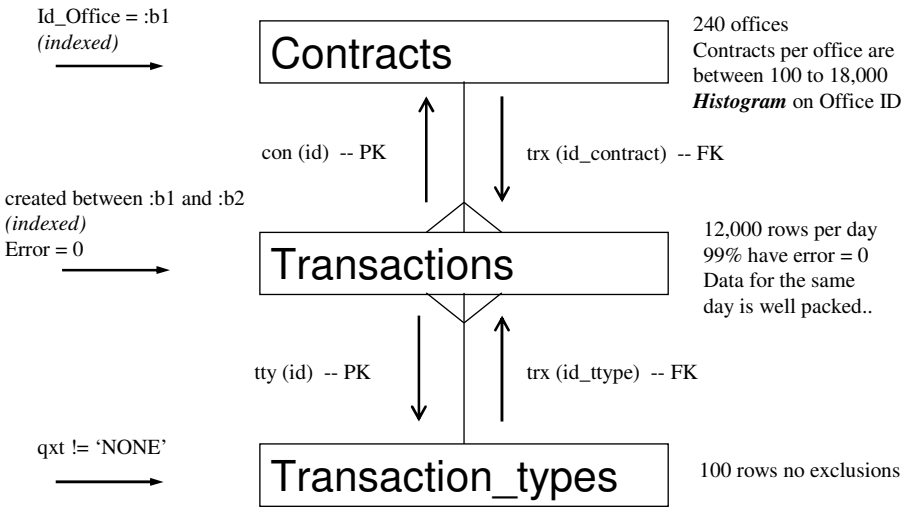
Case Study (a)

```
select
        distinct trx.id_contract
from
        transactions        trx,
        contracts           con,
        transaction_types   tty
where
        trx.id_ttype        =  tty.id
and     trx.id_contract     =  con.id
and     con.id_office       =  :b1
and     tty.qxt             <> 'NONE'
and     trx.created         between :b2 and :b3
and     trx.error           = 0
;
```

Case Study (b)

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT				14976
1	SORT UNIQUE		7791	304K	14976
2	FILTER				
3	HASH JOIN		7798	304K	14974
4	VIEW		9819	98190	1599
5	HASH JOIN				
6	INDEX RANGE SCAN	CON_OFF_FK	9819	98190	35
7	INDEX FAST FULL SCAN	CON_PK	9819	98190	1558
8	HASH JOIN		7798	228K	13374
9	TABLE ACCESS FULL	TRANS_TYPES	105	945	3
10	**TABLE ACCESS FULL**	**TRANSACTIONS**	**7856**	**161K**	**13370**

Case Study (c)



Case Study (d)

Get rid of the histogram on office_id !

Modified Indexes

```
contracts(id, office_id)
```

Expected plan:

```
hash join
  table access full transaction_types
  nested loop
    table access by rowid transactions
      index range scan transactions_idx
    index range scan contracts_idx
```


Case Study (e)

Modified Indexes

```
transactions(created, con_id, error, id_ttype)
contracts(id, office_id)
```

Expected plan:

```
hash join
  table access full transaction_types
  nested loop
    index range scan transactions_idx
    index range scan contracts_idx
```

Case Study (f)

Modified Indexes

```
contracts(id_office, id)
transactions(id_contract, created)
```

Expected plan:

```
hash join
  table access full transaction_types
  nested loop
    index range scan contracts_idx
    table access by rowid transactions
      index range scan transactions_idx
```


Summary

- Know the data
- Draw the picture
- Identify the problems
  - Bad indexing
  - Bad statistics
- Optimizer deficiencies
  - Structure the query with hints