As I begin my journey, let me take you through three encounters. Three rapturous encounters.

Encounter 1 - A reporting application takes two and a half hours to compute results, only to return message that no data was present for the specified criteria's. The application gets under scanner, upon analysis by an Application Developer, he removes just one word from the query, and the output comes in half a second.

Encounter 2 - The Application Developer was given a query taking quite huge time on a table, despite presence of indexes. By modifying one index, the cost reduced to one-hundredth.

Encounter 3 - By adding just a small thing that appeared like a comment, the Application Developer improved query's time four-fold.

Above three scenarios are what actually happened to an Application Developer. Before I hop on to reveal the identity of that person, let me take you through the Table of Contents.

TOC over, now back to what we were talking about. Let me reveal the identity, that guy was me.

This article is based purely out of my own personal experiences. I am not a DBA, just a simple Application Developer who learned most of his PL-SQL on the job. I am not going to quote how things could have been; I am going to narrate how things actually turned out to be. It is my sincere hope that the things mentioned here infuse generous doses of positive energy into your applications and help in avoiding a few common mistakes committed by Application Developers.

I made use of Oracle Database 10g for most of my queries depending on where information was hosted, and often executed operations using front line interfaces such as TOAD and SQL Developer. The experiences described in this paper are a result of the things I did in a time frame spawning 9 months from March 2009 to December 2009. Things just kept on materializing one after the other, as if someone above was sending them for a reason. I don't know what was the exact cause, but as a human I felt I should share with others what all I received, and hence development of this paper took place. I don't know how much of this would be useful to you, but even if a small percentage of the things described here help in ramping up your application, my purpose would be resolved.

Towards the end of this paper, I have discussed a few good practices based on recommendations by my peers, hope that they enlighten up your experiences as a coder and make the end result stupendous as well as comely. A bit of “Gyaanbazi” is also there, based on few interesting things I had heard but never got the chance to actually try them.

There are cases when your application's performance degrades not because of fault in your code, but because of issues with how your database is configured. These things lie in the purview of a DBA, hence won't be touching them here.

Done with the Introduction, let’s take rest of the proceedings further.

1. Indexes

Probably the de facto thing in anything related to SQL tuning, and if you are wondering what they are, an index serves as a directory for referencing where your data is stored in the table. When you have to look for a specific material in a book, you don't have to browse through the entire book; you just open up the Index page and hop on to the desired location. Similar effect takes place when you have indexes on your data fetching queries in a table.

Kindly note, do not assume that indexes have to be looked upon by DBA, it is the responsibility of an Application Developer to plan his / her applications accordingly in such a way that indexes would be used optimally. He / she should prod over what columns are being used and what data is going to be retrieved. The DBA has to maintain the indexes as they grow in size.

On several occasions I encountered a FTS (Full Table Scan) in a query, simply because index wasn't present on one of the prominent columns being referred in the where clause of the query.

There are several types of indexes, like a B* Tree index (the most commonly used), Bitmap index, Function based index, Composite index, Partitioned index. I am referring the first one here, would be covering the rest in proceeding pages of this paper.

Keep the following points in mind with regard to indexes.
i) It isn't necessary that you need to have an index on all the columns being referred in the where clause of a query. Creating an index creates overhead and often their size go up in gigabytes depending upon the size of the table. Best approach in this case is to write the query, populate table with data (assuming code hasn't gone into production yet), and check for explain plan of the query. If the plan depicts an FTS, then go ahead with index creation.

ii) If you are making use of like% operator in your query, then your existing index would be surpassed, except in the case you are referring only the first character (e.g. `where first_name like 'A%'`). If you are discovering occurrence of a fixed set of characters in your query (e.g. `where file_dump like '%20090918%'`) and if they happen to start from a fixed location every time (e.g. from 50th position to the next 8th), you may consider using a function based index (discussed later), if their position varies every time then you would either have to restructure your application or be content with the FTS.

iii) If your index is storing a value that is present on more than 10% of the total number of rows in the table, you may consider dropping the index (as the overheads would be very high) or use a Bitmap index (described later).

2. Avoid unnecessary conversion from one data type to another in where clause of a query

Often I have seen queries having entries in their where clause specified as

```
select field1, field2, field3 from ch_tab
where to_date(in_date,'DD-Mon-YYYY') = trunc(sysdate);
```

Here `in_date` is a character field storing date in the form `yyyymmdd`, a normal B*Tree index is present on this field called `INX_IN_DATE`
The conversion of the field `in_date` should be avoided, as it would do the following
i) Convert `in_date` into type date, thus increasing cost.
ii) Bypass the existing index on `in_date`, as the value stored in its index would be no longer referred due to its date being converted.

In this case I suggest the following approaches
i) Before this query is executed, convert the value of `trunc(sysdate)` into the type that is in compliance with how values are being stored in `in_date` and store in some variable. The modified code would look as

```
lv_date := to_char(sysdate,'yyyymmdd');
select field1, field2, field3 from ch_tab
where in_date = lv_date;
```

This would make use of index `INX_IN_DATE`, as well as preventing data type conversion in where clause of the query.
ii) Use a function based index on `to_date(in_date, 'DD-Mon-YYYY')` and if possible drop the existing index `INX_IN_DATE` so as to reduce overheads. But ensure that other applications / queries referencing `in_date` of the table `ch_tab` are referencing the column in_date in the manner specified in the above query only; else they would suffer performance lags.
iii) Change the data type of `in_date` from character to date. For this the existing column would have to be truncated, the data type would have to be changed, and then the column would have to be repopulated. The logic of inserting data into the table would have to be altered as well. The advantage of using this approach is that you wouldn't have to convert data from one data type to another, and in general is considered a good practice to store your dates in date data type instead of character.

Once I was working on an application having severe performance issues. I observed the following line in its where clause

```
where  to_date(to_char(dp_date,'YYYYMMDD'),  'YYYYMMDD')  =
'24-Mar-2009';
```

So what do you feel, how efficient was the above piece of code?

3. Data Partitioning

Partition is a word we all have heard of. From partitioning of the hard disk into drives other than C: to the partitioning of shelves in your cupboard, we have an idea of what it is. Be it software or real world, partitioning is implemented to divide existing set of things into segments so that it would be easier to organize stuff as well as retrieve it. Similar thing happens in Oracle as well, wherein you can divide your table, index, etc.

into partitions. Here we would discuss partitioning of a table, where you can partition your table based on data stored in a particular column.

Two popular techniques deployed for table partitioning include Partition by Range and Partition by Value.
i) Partition by Range - Is used when you wish to store data pertaining to a range of values e.g. for a table storing details of transactions made in a supermarket (table name `tab_daily_trans`), you can create partitions based on price of a transaction (stored in a column `totalcost` of type number), with one partition string data for `totalcost < 100`, another one storing `totalcost < 200`, and so on.

ii) Partition by Value - Is used when you wish to store data pertaining to a specific value. For example if you wish to store details of your customers based on the state they hail from, you can create different partitions as per state name and store details accordingly.

Once when working on an application having performance issues, I was involved with a table having 100 million entries but no partition. Value based Partitions were created based on a date field of the table specifying the date on which data was inserted into the table. The effect was that the select count(*) query that was taking 4:32 minutes on the non-partitioned table, took 0:31 minutes on the partitioned one. Other operations on the table got performance boost as well.

4. Function based indexes

I had referred the term in the first point of our discussion. What function based indexes do is that, they apply function onto a column and store its result in an index.

Consider the case of table `tab_employee`, where we are having an index INX_JOIN_DATE on the date field `join_date`. The query currently existing for finding out details of all employees who joined on the date specified in a date variable `lv_jdate` (it does not store the time component, only the date one) is

```
select first_name, last_name, join_date
from tab_employee
where trunc(join_date) = lv_jdate;
```

This would result in severe performance implications as the table grows with time, as the cost would be high due to `join_date` getting truncated every time and the existing index getting bypassed, resulting in a FTS.

In the following case you can create a function based index on `trunc(join_date)`, say INX_FN_JOIN_DATE . This would lead the optimizer to use the value stored in function based index, thus reducing the cost of truncating it every time and preventing FTS. If none of your queries is using the join_date field as it is in the where clause, you may consider removing INX_FN_JOIN_DATE . This would reduce the overhead of maintaining two indexes.

In case you do not wish to create a function based index, then you can alter the query accordingly.

```
select first_name, last_name, join_date
from tab_employee
where join_date >= lv_jdate
and join_date < lv_jdate +1;
```

I applied this approach to a table having 400 million records without any table partitioning, as a result the query which was taking 2:24 hours took 700 milliseconds.

I encountered another significant use of function based index by using the substr function. I was looking after a query based on a table having 100 million records having one of the conditions in the where clause as
`where dump_file like '%20090318%'`
Upon closer look at the field dump_file, I observed that its first 50 characters were same for all records, and so were the last 32 characters. This column was being used to store names of files that were being created onto the Operating System, and each row contained path of that file. All files were stored in the same directory, having similar naming conventions, with the only difference being in the dates that went through in the file.

I suggested two approaches for the following.
i) Create a function based index on `substr(dump_file,50,8)` which would store values of the next 8 characters starting from the 50th position. Accordingly modify the query as
`where substr(dump_file,50,8) = '20090318'`
ii) In case function based index can't be created on the above table due to its large size and ever happening transactions on it, then retain the existing normal index and in place of 20090318 append it with the characters that are common for all file names, thus making the query as
`where dump_file = 'C:\Program Files\ISO Soft\query dump 20090318 via terminal server.dmp'`
You can make use of a foreign key relationship if you want, storing the values for `dump_file` in another table.

5. Partitioned indexes

You can partition your table. In a similar manner, you can partition your index as well. Just like you get performance gains with your table, you get them with your index too.

We were having a reporting database, which used to pick up details of customers on daily basis from parent database and store in child one. There was a column in tables of parent database specifying when the row was last updated. We created a partitioned index on the date field for each day and accordingly used to load data on incremental basis making use of that partitioned index.

6. Index hint

It sometimes does happen, that you are having index on one of the prominent columns in the where clause of your query, yet it results in a FTS. Reason for this is attributed to the fact that the index is simply not used during query's computation. It can be resolved by making use of the Index Hint.

Hint is a directive to the compiler for performing query.

Index Hint specifies name of the index to be used while the query is being performed. An illustration of the same is

```
SELECT /*+ INDEX(INX_TB_DATE) */
field1, field2, field3
FROM
tab_tb_desc
where tb_date > trunc(sysdate);
--Here index on tb_date is INX_TB_DATE
```

Note that this problem normally doesn't occur in simple queries; it takes place in rare cases involving multiple joins with several other tables.

7. Processing in Parallel

If your DBA has set values of few system parameters such as parallel_max_servers, parallel_min_servers, parallel_cpu_count, etc. you can apply parallel processing onto your query by making use of the parallel hint.

For the query

```
select a.* from order_table a;
```

Cost - 26,000

If we wish the CPU to parallelize the process by applying 4 concurrent servers onto it, you may modify the query as

```
select /*+ PARALLEL(a,4) */
from order_table a;
```

Cost - 4,4189

Here ‘a’ signifies the table for which processing is to be done, and ‘4’ implies the number of concurrent parallel servers. You can increase the number of servers to decrease the cost further. However, keep in mind that setting the value too high could affect other processes running on your database as they might not be able to get CPU for processing and have would have to wait in order to get resources. Also, if the amount of servers as specified in the parameter isn’t currently available due to being engaged with other processes, then your query would run normally sans any parallel processing.

Once I stumbled upon a DBA's solution to a program having performance issues. The original program used to execute 4 different queries in succession and finally combined their outputs to give the desired result. The DBA modified the program by putting those 4 different queries in different procedures and modified the main procedure by calling

those 4 parts by submitting jobs. Those jobs, at their completion, used to insert some value in a table. In the main procedure, a timer was set which used to keep checking entries in that table as if the jobs had been completed or not. Once done, it used to proceed to the final part of combining them.

While the program was indeed novel, it did have its share of limitations. There is a limit to the number of jobs you can simultaneously run in Oracle. And if one of those 4 jobs failed, the final output would not be the desired one.

8. M-Views

Not every human being who weighs more than 200 kilograms is a sumo wrestler. Same thing can be said about Materialized Views (or Snapshots), if not handled properly.

M-Views should be created when your application has to retrieve same set of data repeatedly from a table. These can involve operations such as aggregations and pre-computation of results for various business users.

There are cases wherein M-Views are created as exact replicas of their parent tables and set on automatic refresh after a fixed interval. This is done to reduce load on their parent tables, so that any application that involves them for read-only purpose may use the M-Views instead. While this does serve the intended purpose, but as the M-Views grow in size, their cost of maintenance increases proportionately. And when too many applications start using these M-Views for varying set of data rather than the same set of data, the performance of the system is severely affected.

I was once involved with a query that took 2 hours on the parent table and 2.4 hours on the M-View. The execution time used to get delayed further as after every 45 minutes the M-View started getting refreshed, thus hogging up most of the system resources and so the application was left with no option but to wait till the system granted it enough resources.

One thing to be kept in mind is, if the basic architecture of your database involving parent tables is weak (e.g. the table has 500 million records sans table partitioning), then you can't expect your M-Views to perform miracles.

9. Explain Plan

The Explain Plan acts like a map, it shows you what path would be followed in execution of a query, what estimated resources would be consumed, what would be the total cost, if there is a full table scan or an index scan, on what basis are tables being linked, and ultimately if the path should be followed or not.

Mark it as a habit, before you implement any query in production, check it’s explain plan thoroughly. If there is occurrence of a FTS in place of Index scan, change the query. If the cost can be brought down by trying a few permutations and combinations, try them. When you have the access path of how things are going to turn out, you can’t resist the urge of not to use it, can you?

Note, at times the explain plan does not give the exact answer, what may appear to be optimal in explain plan may turn otherwise during execution. However this isn't very common.

10. Usage of composite indexes

Composite index implies creating a single index on multiple columns. The catch with composite index is, if your query is referring say three columns (field1, field2, field3) on one table, you may create one composite index instead of creating three different indexes. This helps reducing overheads.

Now, the downside is, if you were to create another query on the table that were to refer one of those three columns, then the composite index would not used if you were referring field2 and field3, hence causing a FTS. If you were referring field1, then the composite index would be used as it is the first column being stored in the index.

Implement composite index as per your application's architecture. If I am storing data for all customers of an organization in a country and have to pick up records according to its state, I can consider creating a composite index with state name and the other field being referred.

11. The problem with too many indexes

A common misconception is that you need to have an index on all the columns that are referred in the where clause of your respective queries. While index do help in locating where your data is stored in a table, there are cases wherein size of the indexes exceed the size of the table. End result of which is that you end up storing less of your data and more of your data about data.

Once when working on a query that was fetching details of a particular customer, three of its columns was being referred and index existed on only one of the fields. That field was the customer for which details were to be fetched, that alone took out 10 records from 100 million entries, rest of the process involved filtering out records from those 10. Hence the overheads of creating index on the other two columns were prevented.

Please note that, the above case doesn't hold true in every case. Always test your query, whether it requires index or not. At times application restricting gives greater benefit in the long run rather than creating an additional index.

12. Nologging

Whenever you perform an operation such as insert and update, Oracle stores that information in Redo logs, so that in case the transaction gets aborted due to one reason or the other, you can restore it from the point of failure. If you wish to bypass those Redo logs, you can do so by specifying the keyword nologging in your query.

```
insert into tab_customer nologging
select -- rest of the query;
```

The above query is particularly useful when you have a stable system that doesn't crash and so you don't need to store any information in Redo logs. Even while creating indexes, putting up the nologging option can speed up your operations by as much as 30%.

13. Use order by and group by clauses only if needed

When you apply the order by clause to your query, it rearranges the dataset every time it fetches a particular record, hence increasing execution time. Same is the case with group by. You should use them only if your application actually needs them. Once when tuning a query, I noticed an order by clause that was rearranging the 2.2 million records being fetched by it. After talking to client, I got to know that sorted order of records wasn't of any use to them, so I dropped the order by clause, and the output that I used to get in 45 minutes took only 12 minutes.

14. Bulk collect

Leaving this topic as an exercise for the reader. All I would say is, the loading operations that were taking us one hour took less than 15 minutes after making use of bulk collect.

15. Bitmap index

If your column is storing only a handful of values, like customers who are active, suspended, or deactivated; then you can consider creating a Bitmap Index on the column. Couple this with Check Constraints (as discussed in point 9) for an optimum performance. Creating a normal B*Tree index in such cases often has larger overheads; a recommended practice is not to create it on a column in which same data is occurring on more than 10% of the data.

Remember, you insert data into the table once, but retrieve it multiple times. So be careful with how you insert data into your table.

16. DBMS_STATS.GATHER_STATS

This is used to update information about statistics such as indexes present in the table. There was a case where an application was taking more than 2 hours to compute results despite presence of indexes, we ran this command and the query gave us output in minutes.

The inference is, if you have indexes, make sure they area analyzed periodically either by setting them in some job or doing that in your procedure just before the resource-extensive query is called.

17. Deleting all the rows from a table

The general practice deployed for deleting records from a table is

```
delete from tablename
where -- condition goes here
```

This is effective when you have to delete one row as per some condition. But when you have to delete all the rows, the process becomes tedious, as it would first scan where a

row is, remove it, write its information into redo log, go to the next row, repeat the process.

An effective solution in this case is to use the truncate command as follows.

```
truncate table tablename;
```

This command deletes all the records lying in your table without writing their information to redo logs. It doesn't check where your next row is lying, it just takes the entire table at one go. Effectively, it calls your drop table command and recreates the table structure (as it is a DDL command, no redo logs are maintained and commit isn't needed as well).

Note that truncate table doesn't work when you have to delete one / a few selected record(s). Also, if you wish to recover your transaction at some later stage, you can't do that.

18. Null values in an index

When interviewing candidates for the role of a PL-SQL developer, asking them the question "Can null values be stored in an index?" fetches three answers.
i) No, they can't be.
ii) Yes, they can be, are stored by default.
iii) Yes, they can be, by assigning a contemporary value to the one with null value.
The third answer is given by selected few, and they happen to be the ones who land up in your organization soon.

Depending upon your application's functionality, you may consider accommodating null values in an index. Once when working on a query involving details of their 11 million customers that was doing a FTS, I noticed an order by in the end of query pertaining to a field storing number of times a customer had been barred. 80% of the values for this field were null, since the customers had never been barred. So we modified the existing index to accommodate null values, result of which was that the cost reduced from 500,000 to 313.

Note that if it isn't possible to create an index as in the above case, you may consider dropping the order by clause or change the application logic to an incremental one based on as and when the customer gets barred.

19. Order of columns in where clause

Consider the following two queries for fetching details of a particular record with serial_id 2 having its entries on current date.

i)
```
select * from tab_rep
     where serial_no = 2
     and created_date = trunc(sysdate)
```

ii)
```
select * from tab_rep
     where created_date = trunc(sysdate)
     and serial_no = 2
```

The possible scenarios in this case would be
i) Indexes are present on serial_no and created_date.
ii) Index is present on none of the fields.
iii) Index is present on one of the two fields.

In the first case, the query would take equal time to compute. The optimizer doesn't follow linear fashion of first fetching details of serial_id = 2 and created_date = trunc(sysdate), it runs both at the same time and then combines their results. So irrespective of the order, the time would remain the same.

In the second case, the time would remain same due to the aforementioned reason. However, in large complex queries involving multiple joins and particularly with older versions of Oracle such as 8i, if index isn't present on one of the fields then the order varies at times, even though the cost is shown to be equal by the optimizer. You would have to try various permutations and combinations in this case.

In the third case, are you seriously considering deploying this query in your application?

20. Counting the number of rows in a table

While doing a count of the number of rows in a table

```
select count(1) from tab_rep;
```

As your table grows in size, the time of execution of the query follows suit. In the following case, I suggest to make use of the following

```
select num_distinct
from user_tab_columns
where table_name = 'TAB_REP'
and column_name = 'SERIAL_ID'
```

The above query gives you the number of distinct values being stored for the column `SERIAL_ID` in the table `TAB_REP`. When you refer the primary key as the column in where clause, it would effectively fetch you the number of rows in the table, hence you would get your result in much lesser time.

Note that if there is a gap between when your table was last analyzed and the time of running the query, then the output would obviously vary (you can refer the column `last_analyzed` in table `user_tab_columns`).

Once when doing a select count on a table with 300 million records, I got the output in 07:12 minutes. I analyzed the indexes; it took me 3:02 minutes, then ran the query on user_tab_columns, and got the correct output in less than a second.

A little bit of Gyaanbazi

1. Contention and Reverse Key Indexes

Your normal B*Tree indexes are crafted the way tress are created, depending on what values are there for your index column. For a column storing price of a product, there would be leaves determining ranges between `0-100`, `101-200`, and so on.

Now, when you have a sequence as primary key on which index is present, as your table grows, so does your index grows on the right. This leads to uneven distribution and your index keeps on increasing towards right, termed Contention.

To resolve this, create a Reverse Key Index. This would reverse the elements present in your data column and store it accordingly. Hence for records `311021`, `311022`, and `311023`, they would be stored as `120113`, `220113`, `320113`, thus making them store in a uniform manner, rather than towards a single side.

2. Index key compression

When you have a Composite Key Index, you can compress one of the columns in the index (termed Index key compression). This is particularly useful if one of the columns has repeated data.

For three columns `Country`, `State`, and `City`, you can compress the index for first column, as it would be common for a lot of records. You can compress the index for second column too, as that itself would be common for a number of records.

3. Foreign key, not null, and check constraints

When you keep a foreign key constraint on one of the columns of a table, it makes the Optimizer easier to know what all values are going to reside in a table and directly refers values residing in the parent table, and if index is present there then that is referred directly.

Similarly when you create a not null constraint, the Optimizer gets to know that no null values would be residing in the table and hence any chances of FTS with is null / is not null are hence avoided (Null values aren't normally stored in an index). Doing a select count on two tables with exactly same structure and same data, but the first with not null constraint and second with no such constraint, would take lower time in the first case even when no null values are present (except if you accommodate null values in the index).

And when you have a check constraint consisting of what values would be going in a table, the Optimizer knows exactly what values it is going to encounter in the table.

Let’s go over to a few practices I learnt.

Few good programming practices

1. Case is better than multiple else-if's

Consider the following piece of code.

```
if var1 = 1 then --rest of the code
elsif var1 = 24 then --rest of the code
elsif var1 = 39 then --rest of the code
elsif var1 = 23 then --rest of the code
elsif var1 = 93 then --rest of the code
elsif var1 = 17 then --rest of the code
else --rest of the code
```

Here the compiler would first check if the value for `var1` is 1, if not it would go to the next iteration and check if its value is 24, if not it would go to the next iteration for 39 and so on.

The issue that arises is, the compiler is performing a lot of iterations just to check if the value of `var1` corresponds to what is desired or not. If the value of `var1` was 7, I would have to reach the last value, default one in else, undergoing 6 extra iterations.

When we write the same code using switch

```
switch(var1)
case 1 : --rest of the code
case 24 : --rest of the code
. . .
default: --rest of the code
```

Now for `var1` with value 7, the compiler would directly jump over to default area.

2. Try doing your PL-SQL computation in a single query itself, instead of multiple ones

If you have to update details of your employee, such as salary, tax, date of last payment, etc., then instead of writing multiple queries to perform that operation do it in a single query. As a good programming practice try doing your stuff in a single query only.

3. Too many commits

Once I encountered an application, wherein the user was fetching thousands of records one by one using a cursor, performing some iteration on the record, updating it, and saving it there only by doing a commit. This increases your program's overall execution time as it invokes the system to save what all has been done, and then goes back to your program.

From my personal experience, try to save all your data in a single commit, if the number of records isn't very high. This depends on how your database is configured. I save records up to less than 1 million in a single go. Till the data commit isn't performed, your data is saved onto a buffer. If the buffer limit exceeds you are bound to get errors such as Rollback Segment Too Small.

4. Select Count

Consider the following two queries on a table tab_users.

i) `select count(*) from tab_users;`

ii) `select count(1) from tab_users;`

Both queries depict the same amount of cost when you run the Explain Plan. However, it is considered a good practice to go with the second query. That is because when you do `count(*)`, it takes into account all the columns residing in the table. With `count(1)`, it only takes either the first column or the primary key, hence reducing the amount of data that is transferred onto the buffer.

5. Length of a procedure / function

Its often debated over how long should your function be, while some say it shouldn't exceed 20 lines, other have their own versions. Keeping that aside, it is a good practice not to make your procedure / function too long. They should be fragmented into smaller sets of functions / packages and the entire procedure should be kept in a single package.

6. Boolean geometry - de Morgan laws

Remember de Morgan rules that you had probably studied in your mathematics course at school / college? They implied how to break down a complex statement into a simpler one that gave the same output. The very same set of rules can be applied to your statements as well.

```
if ( !var1 OR !var2 OR !var3 OR !var4)
```

Can be simplified as

```
if(!(var1 AND var2 AND var3 AND var4) )
```

7. Selecting all the columns in a table

Quite often we go as

```
insert into tab_destination
select * from tab_source;
```

This practice should be avoided, as you never know when and how your table structure may get changed, a column could be added as well as a column be deleted (though that is rare). As a good practice always mention the column names wherever possible.

```
insert into tab_destination
(df1, df2, df3, df4)
select sf1, sf2, sf3, sf4 from tab_source;
```

Final Words

Before I wind up the article, here is what I did in the scenarios mentioned above.

Encounter 1 - Removed trunc from a date field being mentioned in the query, as no function based index was present on it.

Encounter 2 - Modified the existing index by accommodating null values in it, as 80% of the values being referred here were null.

Encounter 3 - Added parallel hint with 4 processors.

That's all I have. Thanks for going through my article, hope it was of use to you. In case I have more comely experiences in future, would share them with you as well. :-)

-Yaju Arya.