Index Analysis Guide

A step-by-step guide on how to work out if each column in a query is suitable for an index

Ben Brumm

www.databasestar.com

Welcome!

Do you want to know the step-by-step process I use to work out if creating an index is a good idea?

You may have seen the flowchart about when to create an index, and what type.

This guide will take you through the steps that I use, along with that flowchart, to creating an index on the column.

So, let's get started.

Your Query

We create an index to make a query run faster, right?

So you'll need a query to start with here. We'll measure the impact of creating an index on a query.

I've found that in most cases I'm either:

- Writing a new query or updating a specific query (as in, I know what query I'm optimising for)
- Looking into an area of the application to try and optimise (and I don't know the exact query)

I won't go into too much detail on how to look into an area of an application to find out which query is performing slowly. There's a few methods you can use and some tools included in Oracle (such as sqltrace and tkprof).

For this guide, I've assumed you have a query in mind.

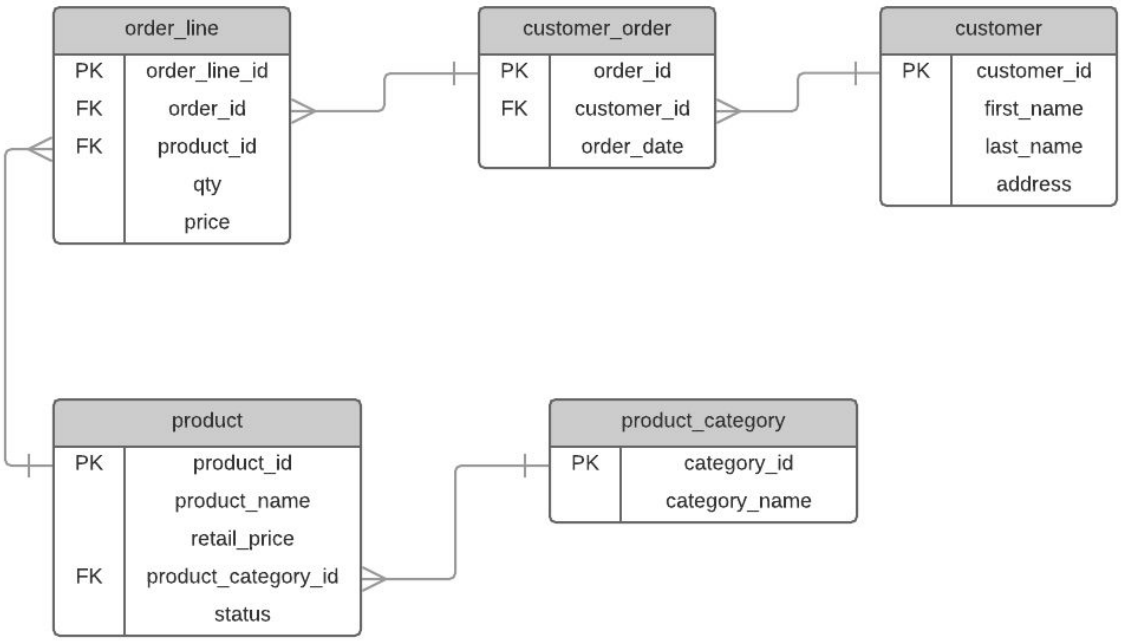
You might have gotten a query from the application that is performing poorly and you want to optimise it.

You might be working on a new feature and writing a brand new query. Or updating an existing query.

In any case, you'll have a query to work on.

This is where we will start.

For this guide, we'll use an example. The sample database I'll use looks like this:



The query we'll use for the example is this one:

```
SELECT
Co.order_date,
co.order_id,
co.customer_id,
SUM(ol.qty) AS num_products,
SUM(ol.price) AS order_value
FROM customer_order co
INNER JOIN order_line ol ON co.order_id = ol.order_id
WHERE co.order_date = '30-JUN-2017'
GROUP BY co.order_id, co.customer_id, co.order_date
ORDER BY co.order_date DESC, co.customer_id ASC;
```

ORDER_DATE	ORDER_ID	CUSTOMER_ID	NUM_PRODUCTS	ORDER_VALUE
30-Jun-17	4102	52	150	155.54
30-Jun-17	4460	266	39	52.67
30-Jun-17	7634	289	79	77.24
30-Jun-17	3046	387	121	116.89
30-Jun-17	4476	405	165	92.66

30-Jun-17	3504	422	146	136.18
30-Jun-17	4549	515	119	133.93
30-Jun-17	6503	535	88	86.3

It finds the order and customer details, the number of products, and the order value for a 12 month period.

So, we've got an example query here, and you have yours. Let's go to the next step.

Execution Plan

The first step I take on a query is to run an execution plan on the query.

This will show me how Oracle is running the query on the database.

You can do this in two ways:

1. Run an EXPLAIN PLAN FOR on your query; or
2. Click the button in your IDE.

My personal preference is the button in the IDE because it's easier. I'll show you both ways to get the execution plan in this guide.

To get the execution plan using the EXPLAIN command, add the words EXPLAIN PLAN FOR before your SELECT query:

```
EXPLAIN PLAN FOR
SELECT
Co.order_date,
co.order_id,
co.customer_id,
SUM(ol.qty) AS num_products,
SUM(ol.price) AS order_value
FROM customer_order co
INNER JOIN order_line ol ON co.order_id = ol.order_id
WHERE co.order_date = '30-JUN-2017'
GROUP BY co.order_id, co.customer_id, co.order_date
ORDER BY co.order_date DESC, co.customer_id ASC;
```

This will not run the query itself, but will prepare an execution plan and store the results in a table.

To then view the plan results, you can run this query:

```
SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY());
```

You'll then get an output like this:

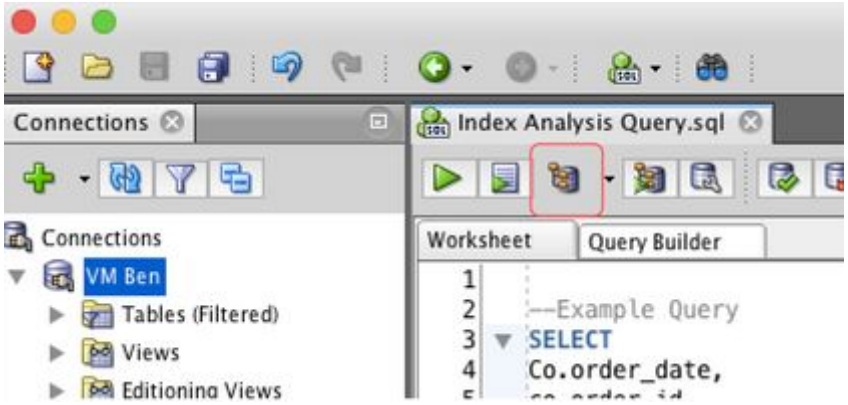
```
Plan hash value: 362070161

-------------------------------------------------------------------------------------------
| Id  | Operation           | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                |  1644 | 44388 |   107   (3)| 00:00:02 |
|   1 |  SORT GROUP BY      |                |  1644 | 44388 |   107   (3)| 00:00:02 |
|*  2 |   HASH JOIN         |                |  1644 | 44388 |   106   (2)| 00:00:02 |
|*  3 |    TABLE ACCESS FULL| CUSTOMER_ORDER |   164 |  2624 |    10   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| ORDER_LINE     |   100K|  1074K|    95   (2)| 00:00:02 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("CO"."ORDER_ID"="OL"."ORDER_ID")
   3 - filter("CO"."ORDER_DATE"=TO_DATE(' 2017-06-30 00:00:00',
              'syyyy-mm-dd hh24:mi:ss'))
```

Alternatively, there's a button inside the IDE that shows you the execution plan. In SQL Developer, it's here:



Click the button and the plan is shown.

Now, let's take a look at the plan. We can see here that there are a couple of TABLE ACCESS FULL entries, one of which is taking a large amount of the cost.

How can I improve this query?

Foreign Key Columns

The next thing to do is to look at the columns being used in the query.

Indexes are focused on columns, so this is where we will start.

Look at all of the columns being used, and the tables. Notice the columns in the SELECT clause, the WHERE clause, the GROUP BY clause, ORDER BY clause, and all columns mentioned.

Make a list of each of these columns in a spreadsheet. Something like this is a good place to start:

Table	Column
customer_order	order_date
customer_order	order_id
customer_order	customer_id
order_line	qty
order_line	price
order_line	order_id

I've listed all columns and tables used in the query. The table is useful as some columns exist in both tables (e.g. order_id).

The first step once we have a list of columns is to determine if any are foreign keys. This is because Oracle does not automatically create indexes on foreign keys (it does on primary keys), so foreign keys are a good candidate for indexing.

Are any of these columns foreign key columns?

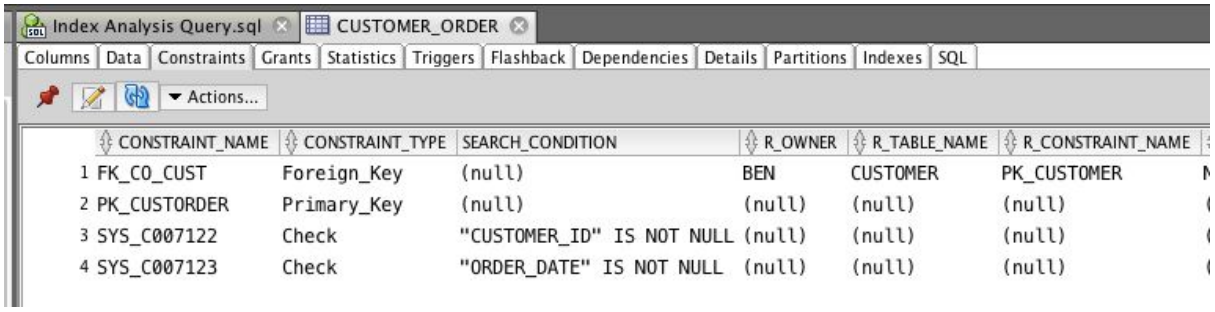
How do we find out?

There are two ways:

1. Look at the table definition in the IDE; or
2. Query the data dictionary to find out

Most IDEs (including SQL Developer) will show the foreign keys of a table in the table properties. To access these:

1. Expand your connection name, and the Tables list, on the left of the IDE.
2. Click once on the table name (or, if you have changed the setting in SQL developer to open the properties on a double click, then double click on the table name)
3. Open the Constraints tab on the new tab that appears:



We can see the first record here has a constraint named FK_CO_CUST. We can tell immediately by looking at this name that it's a foreign key, because I follow (and recommend) a standard naming convention that means foreign keys start with FK (and primary keys start with PK).

We can confirm this by seeing the Constraint Type is Foreign Key. In the R_CONSTRAINT_NAME column, this is the referring constraint name. This foreign key refers to constraint PK_CUSTOMER, which refers to the primary key on the Customer table.

Another way to find the foreign key columns of a table is to query the data dictionary.

You can run a query like this:

```
SELECT a.table_name, a.column_name, a.constraint_name, c.owner,
       c.r_owner, c_pk.table_name r_table_name, c_pk.constraint_name r_pk
  FROM all_cons_columns a
  JOIN all_constraints c
    ON a.owner = c.owner
   AND a.constraint_name = c.constraint_name
  JOIN all_constraints c_pk
    ON c.r_owner = c_pk.owner
   AND c.r_constraint_name = c_pk.constraint_name
 WHERE c.constraint_type = 'R'
   AND a.table_name IN (
```

```
        'CUSTOMER_ORDER',
        'ORDER_LINE'
    );
```

This will show something like the following output:

TABLE_NAME	COLUMN_ NAME	CONSTRAINT_ NAME	R_TABLE_ NAME	R_PK
CUSTOMER_ ORDER	CUSTOMER_ ID	FK_CO_CUST	CUSTOMER	PK_ CUSTOMER
ORDER_LINE	ORDER_ID	FK_OL_ORDER	CUSTOMER_ ORDER	PK_ CUSTORDER
ORDER_LINE	PRODUCT_ ID	FK_OL_PROD	PRODUCT	PK_ PRODUCT

The information is the same as what is shown in SQL Developer.

Now we know which columns are foreign keys, mark these columns as foreign keys in your spreadsheet:

Table	Column	FK?
customer_order	order_date	
customer_order	order_id	
customer_order	customer_id	Yes
order_line	qty	
order_line	price	
order_line	order_id	Yes

I've also marked the customer_order.order_id column as PK, because it's the primary key of this table. As a result, it will already have an index on it.

These might be good columns to create an index on. We'll look at all of the criteria for the columns later in this guide.

JOIN columns

Another factor to look at when analysing columns in a query is the columns that are used in JOIN statements.

Why are these columns useful to index on?

They are used to join tables, and Oracle looks up the value in each table and performs a match. The faster that Oracle can look up these values, the faster the query will run. So, an index on columns in the JOIN clause can help.

How can we tell which columns are being joined on?

Look for any JOIN keyword. This includes:

- JOIN (which is actually an Inner Join, as the Inner keyword is optional)
- INNER JOIN
- OUTER JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- And so on...

The join will include columns on each side of the join. Make a note in your spreadsheet of columns which columns are used in JOIN statements.

This is my example query:

```
SELECT
co.order_date,
co.order_id,
co.customer_id,
SUM(ol.qty) AS num_products,
SUM(ol.price) AS order_value
FROM customer_order co
INNER JOIN order_line ol ON co.order_id = ol.order_id
WHERE co.order_date = '30-JUN-2017'
GROUP BY co.order_id, co.customer_id, co.order_date
ORDER BY co.order_date DESC, co.customer_id ASC;
```

I would note co.order_id and ol.order_id as the join columns here.

Table	Column	FK?	Join?
customer_order	order_date		
customer_order	order_id	PK	Yes
customer_order	customer_id	Yes	
order_line	qty		

order_line	price		
order_line	order_id	Yes	Yes

They might be the same as the foreign key columns, and that's OK. Mark them down anyway.

One thing you might be thinking: what about queries that use a WHERE clause as a join?

So, something like this:

```
SELECT
co.order_date,
co.order_id,
co.customer_id,
SUM(ol.qty) AS num_products,
SUM(ol.price) AS order_value
FROM customer_order co, order_line ol
WHERE co.order_id = ol.order_id
AND co.order_date = '30-JUN-2017'
GROUP BY co.order_id, co.customer_id, co.order_date
ORDER BY co.order_date DESC, co.customer_id ASC;
```

Notice there is no JOIN keyword, but the columns are matched in the WHERE clause. This is an alternative method of joining tables.

Personally, I prefer the method that uses the JOIN keyword (and many others do as well). This is because JOINs are standard SQL, and if you use a JOIN then you can easily tell the difference between matches for joining purposes and criteria for filtering records.

Also, if you have many tables, it's easier to use the JOIN syntax and harder to forget to join them.

So, I would suggest modifying your query to use the JOIN keyword.

If you really don't want to, then you can mark these two columns down as JOIN keywords in your spreadsheet.

WHERE keywords

The next criteria to check for our analysis is the WHERE clause.

We'll look for all columns used in the WHERE clause.

There could be a lot of these. And the columns could be the same as the foreign key or join columns, but that's OK.

Take a look at your query and notice any columns in WHERE clauses. They could be simple clauses like WHERE x=y, or more complicated clauses using subqueries and LIKE.

```
SELECT
co.order_date,
co.order_id,
co.customer_id,
SUM(ol.qty) AS num_products,
SUM(ol.price) AS order_value
FROM customer_order co
INNER JOIN order_line ol ON co.order_id = ol.order_id
WHERE co.order_date = '30-JUN-2017'
GROUP BY co.order_id, co.customer_id, co.order_date
ORDER BY co.order_date DESC, co.customer_id ASC;
```

I would make a note of co.order_date in my spreadsheet. The "co" is an alias of "customer_order" in this query.

Table	Column	FK?	Join?	WHERE?
customer_order	order_date			Yes
customer_order	order_id	PK	Yes	
customer_order	customer_id	Yes		
order_line	qty			
order_line	price			
order_line	order_id	Yes	Yes	

Now we've got a list of our columns and whether they are foreign keys, used in JOINs, or used in WHERE clauses.

What next?

We'll work out the Cardinality and Selectivity of each of them, which will be helpful in determining if an index will be useful.

Cardinality

Cardinality is a term in database development that represents the number of unique rows in a table for a given column.

It's useful to know as Oracle uses it to determine if a query will use a specific index on a column.

The value for cardinality can range from 1 (if all values in the column are unique) up to thousands or millions, depending on the number of rows in the table.

How can you calculate cardinality?

You can run a query to find the count of distinct values in a column, like this:

```
SELECT COUNT(DISTINCT column_name) AS card_col
FROM table_name;
```

Replace your column_name and table_name with the column and table you want to use.

Is this stored in the database? When you create an index, the cardinality is stored as a property of the index. But it is not stored on the table itself.

So, for our exercise, we should run this SELECT query on all of the columns we have identified.

```
SELECT COUNT(DISTINCT order_date) AS card_col
FROM customer_order;
```

Result:

731

We will get a single number back from this query. Enter that number into your spreadsheet for that column.

Table	Column	FK?	Join?	WHERE?	Cardinality
customer_order	order_date			Yes	731
customer_order	order_id	PK	Yes		
customer_order	customer_id	Yes			
order_line	qty				
order_line	price				
order_line	order_id	Yes	Yes		

Repeat the query for each of the columns and enter the value here.

You could write one query for each table to speed things up:

```
SELECT
```

```
COUNT(DISTINCT order_date) AS card_orderdate,
COUNT(DISTINCT order_id) AS card_orderid,
COUNT(DISTINCT customer_id) AS card_custid
FROM customer_order;

SELECT
COUNT(DISTINCT qty) AS card_qty,
COUNT(DISTINCT price) AS card_price,
COUNT(DISTINCT order_id) AS card_orderid
FROM order_line;
```

Record your numbers into the spreadsheet, which should now look something like this:

Table	Column	FK?	Join?	WHERE?	Cardinality
customer_order	order_date			Yes	731
customer_order	order_id	PK	Yes		10,000
customer_order	customer_id	Yes			10,000
order_line	qty				19
order_line	price				192
order_line	order_id	Yes	Yes		9,999

As mentioned, the cardinality is important as it can help Oracle determine if an index will be used.

Selectivity

Selectivity is another measure of columns that Oracle uses to determine if an index will be helpful.

Selectivity is a calculation that represents the ratio of the number of unique values to the overall table.

It ranges from almost 0 (which represents almost every value is the same) up to 1 (which represents every value is unique).

If a column has high selectivity, then an index will usually help.

How can we calculate selectivity?

It's defined as:

Number of Distinct Records/Number of Total Records

Number of Distinct Records is also the same as Cardinality.

So, a query to determine this could be:

```
SELECT
COUNT(DISTINCT column_name)/COUNT(column_name) AS selectivity_col
FROM table_name;
```

Replace your column_name and table_name with the column and table you want to use.

However, for our spreadsheet, it's useful to know how many values are in a column. We can then calculate selectivity from that.

So, to work out the number of values in each column, we can use a simple COUNT:

```
SELECT COUNT(column_name)
FROM table_name
```

You can make this easier by selecting the COUNT of several columns in the one query.

```
SELECT
COUNT(order_date) AS cn_orderdate,
COUNT(order_id) AS cn_orderid,
COUNT(customer_id) AS cn_custid
FROM customer_order;

SELECT
COUNT(qty) AS cn_qty,
COUNT(price) AS cn_price,
COUNT(order_id) AS cn_orderid
FROM order_line;
```

Enter the results from these queries into your table.

Table	Column	FK?	Join?	WHERE?	Cardinality	Count
customer_order	order_date			Yes	731	10,000
customer_order	order_id	PK	Yes		10,000	10,000
customer_order	customer_id	Yes			10,000	10,000
order_line	qty				19	100,000
order_line	price				192	100,000
order_line	order_id	Yes	Yes		9999	100,000

Now, add another column to the spreadsheet to calculate Selectivity. This is calculated as Cardinality/Count. Excel and other spreadsheets will let you calculate this.

Table	Column	FK?	Join?	WHERE?	Card.	Count	Sel.
customer_order	order_date			Yes	731	10,000	0.07
customer_order	order_id	PK	Yes		10,000	10,000	1.00
customer_order	customer_id	Yes			10,000	10,000	1.00
order_line	qty				19	100,000	0.00
order_line	price				192	100,000	0.00
order_line	order_id	Yes	Yes		9,999	100,000	0.10

You'll notice that order_line.qty and order_line.price have a selectivity displayed as 0. This is due to rounding: their actual selectivity is just above 0.

So, now we have a list of our columns with cardinality, selectivity, and record count. We're almost there!

Percentage of Records

Another calculation we can do on our data is to work out what percentage of the table is returned if we filter on a specific value of that column.

This can help determine if an index is used. It's similar to selectivity, but it caters for large and small tables in the same way.

Our query filters on the order_date column. There are 731 distinct values (a cardinality of 731) from a table size of 10,000. Selectivity was calculated as 0.07.

However, another measure of this column is that there are 731 different values, so by filtering on one of them, we are drastically reducing the number of records returned.

We can estimate the percentage of records returned by dividing the cardinality into 1.

For example, 1/cardinality. Or, 1/731 = 0.14%.

This means if we supply a single value for this column, we should return 0.14% of the table. A pretty small amount.

Now, we're filtering on just one value in the column. But we don't know the distribution of data. For this purpose, we can use the "percentage of records" calculation to eliminate the kinds of WHERE clauses that filter on columns that only have a few possible values.

So, add a new column to the spreadsheet called "Percentage of Records" or something similar (I've called mine "Pct" in the interests of space), and calculate the 1/cardinality for all columns marked Yes as being in the WHERE clause.

Table	Column	FK?	Join?	WHERE?	Card.	Count	Sel.	Pct
customer_ order	order_ date			Yes	731	10,000	0.07	0.14%
customer_ order	order_id	PK	Yes		10,000	10,000	1.00	
customer_ order	customer_ id	Yes			10,000	10,000	1.00	
order_line	qty				19	100,000	0.00	
order_line	price				192	100,000	0.00	
order_line	order_id	Yes	Yes		9,999	100,000	0.10	

Let's take a look at how to consider functions for your indexes.

Functions

Is there a function or expression being used on any of the columns? They can determine if an index is used or not.

Look at your query, and see if any functions or expressions are being used.

Examples of functions are:

- UPPER(column)
- SUBSTR(column)
- column * 1.1
- AVG(column)
- column + other_column

Update your spreadsheet if there are any functions or expressions being used.

Table	Column	FK?	Join?	WHERE ?	Card.	Count	Sel.	Pct	FN
customer_ order	order_date			Yes	731	10,000	0.07	0.14 %	

customer_ order	order_id	PK	Yes		10,000	10,000	1.00		
customer_ order	customer_id	Yes			10,000	10,000	1.00		
order_line	qty				19	100,000	0.00		Yes
order_line	price				192	100,000	0.00		Yes
order_line	order_id	Yes	Yes		9,999	100,000	0.10		

We've now finished gathering data from the columns! Let's take a look at what we have so far.

Analyse Columns

Now it's time to look at the data we've gathered for each of our columns and determine which index may be appropriate.

Look at each row in our spreadsheet, which represent a column in our query.

We need to indicate which columns could be used for an index. You can highlight them or add a new column. I'll highlight them in this example.

So, highlight any row in this spreadsheet that you put a Yes for:

- Foreign Key
- Join
- WHERE clause with a Percentage of Records over 15%.
- A high selectivity (over 0.8)

The selectivity of 0.8 is not an exact measurement, but it's a good place to start.

The 15% of "Percentage of Records" is suggested by Oracle. They actually suggest "15% of records returned by the query", but because we don't know our distribution of data at the moment, looking at it at a field level is OK.

We'll exclude the Primary Key for now as it already has an index.

Now your table may look something like this:

Table	Column	FK?	Join?	WHERE ?	Card.	Count	Sel.	Pct	FN
customer_ order	order_date			Yes	731	10,000	0.07	0.14 %	

customer_ order	order_id	PK	Yes		10,000	10,000	1.00		
customer_ order	customer_id	Yes			10,000	10,000	1.00		
order_line	qty				19	100,000	0.00		Yes
order_line	price				192	100,000	0.00		Yes
order_line	order_id	Yes	Yes		9,999	100,000	0.10		

You might be wondering why I haven't highlighted the customer_id row, as that column is a foreign key?

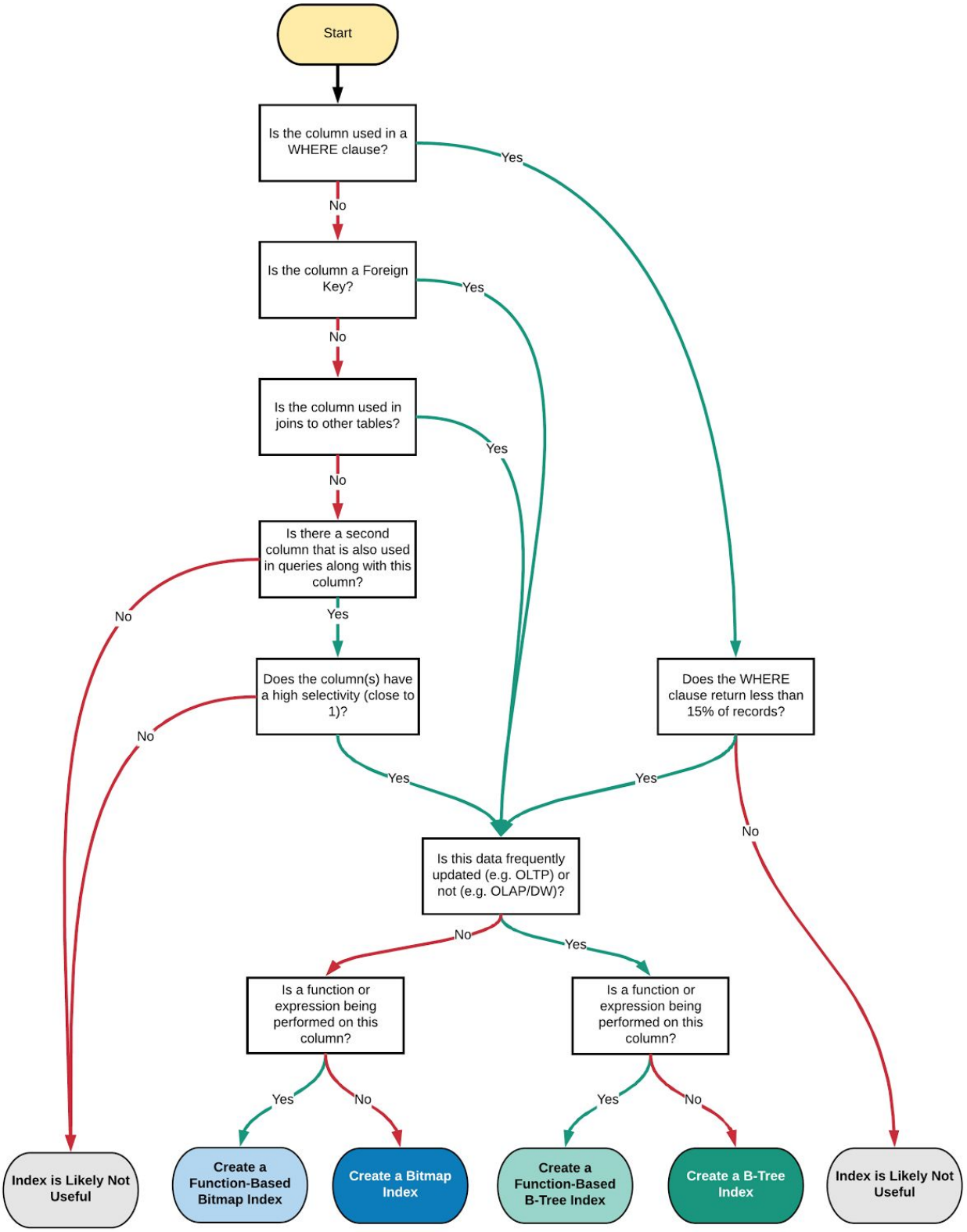
Well, the reason is that foreign key is a link to the primary key of the customer table. So, while it might be worth creating an index on that column to help with queries that join to the customer table, it won't help our query as we don't join to the customer table.

It's there for information purposes, but we don't need it for our query.

For the highlighted columns, which index would be appropriate on these columns?

That's where our flowchart comes in.

If you haven't seen the Index Flowchart yet, here it is:


Start
Is the column used in a WHERE clause?
Yes
No
Is the column a Foreign Key?
Yes
No
Is the column used in joins to other tables?
Yes
No
Is there a second column that is also used in queries along with this column?
No
Yes
Does the column(s) have a high selectivity (close to 1)?
No
Yes
Does the WHERE clause return less than 15% of records?
Yes
No
Is this data frequently updated (e.g. OLTP) or not (e.g. OLAP/DW)?
No
Yes
Is a function or expression being performed on this column?
Yes
No
Is a function or expression being performed on this column?
Yes
No
Index is Likely Not Useful
Create a Function-Based Bitmap Index
Create a Bitmap Index
Create a Function-Based B-Tree Index
Create a B-Tree Index
Index is Likely Not Useful

On your table, for each column you have highlighted, take it through this flowchart. When you have identified which index may be appropriate (from the four boxes at the bottom of the flowchart), add that index type to the spreadsheet.

For example, let's look at the customer_order.order_date column:

- Is it used in a WHERE clause? Yes
- Does the WHERE clause return less than 15% of records? Yes
- Is the data frequently updated (OLTP) or not (OLAP/DW)? YEs
- Function or expression used? No
- Result: B-tree index

So, I would note that for the order_date column, a "b-tree" index could work.

What does "frequently updated" mean?

If you're working on an application database (a database that stores data for an application), it would likely get updated regularly. Data gets updated by users or by the system, new data is added, data may get deleted. This is called an OLTP system (Online Transaction Processing), and is more suited to B-tree indexes.

If you're working on a data warehouse, or a database that is used mainly for reading data (an OLAP system or Online Analytical Processing), then a Bitmap index is likely more suitable.

So, that's why this question is asked. It would be the same answer for all columns.

Now, back to our columns. Repeat the walkthrough of the flowchart for all highlighted columns. Your spreadsheet would look something like this:

Table	Column	FK	Join	WHERE	Card.	Count	Sel.	Pct	FN	Index
customer_order	order_date			Yes	731	10,000	0.07	0.14%		B-tree
customer_order	order_id	PK	Yes		10,000	10,000	1.00			
customer_order	customer_id	Yes			1,000	10,000	0.10			
order_line	qty				19	100,000	0.00		Yes	
order_line	price				192	100,000	0.00		Yes	
order_line	order_id	Yes	Yes		9,999	100,000	0.10			B-tree

We're almost ready to start creating an index.

But first, we should consider existing indexes, and if a composite index could be used.

Existing Indexes

Before we create a new index, we should check what indexes already exist.

This is because:

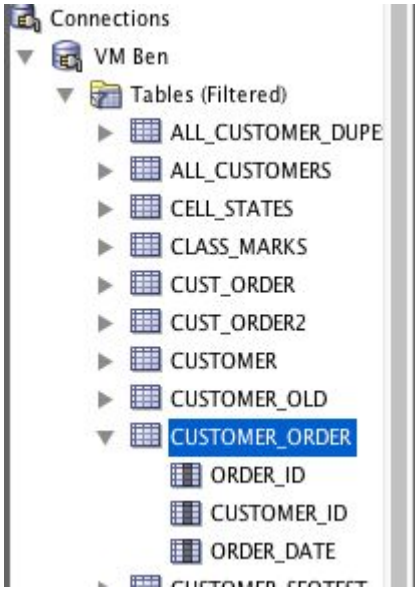
- There may be an index on the exact columns we are looking at, so creating another index won't work.
- There may be an index on some of the same columns we are looking at (e.g. a composite index exists and we're creating an index on a single column).

We want to avoid doubling-up or creating similar indexes. This is because every index will cause INSERTs and UPDATEs on the table to run slower. So we only want to have indexes on the table that will help other queries.

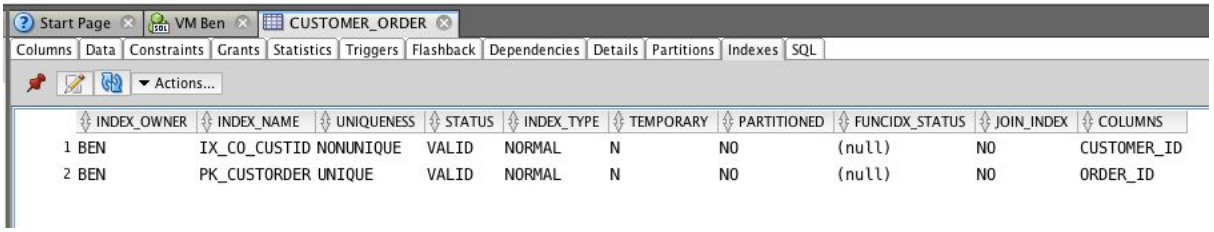
How can we find out what indexes already exist?

Just like with many of the methods we've seen here, we can either look at the table properties in the IDE, or query the data dictionary.

If you're using SQL Developer, you can look up the table properties by expanding the Tables section on the left sidebar and clicking on the table name.



Then, click on the Indexes tab.



You can also query the database by running this query:

```
SELECT i.index_name,
i.uniqueness,
c.column_name,
c.column_position
FROM all_indexes i
INNER JOIN all_ind_columns c ON i.index_name = c.Index_Name
WHERE i.table_name = 'CUSTOMER_ORDER';
```

INDEX_NAME	UNIQUENESS	COLUMN_NAME	COLUMN_ORDER
IX_CO_CUSTID	NONUNIQUE	CUSTOMER_ID	1
PK_CUSTORDER	UNIQUE	ORDER_ID	1

In either case, we have a list of indexes for a table.

Do we want to remove any of these existing indexes? Do we want to change any of them?

How do we know if changing or removing them will break other queries or cause them to go slow?

You would need to look at your code and find out which other queries run on this table, and perform the same kind of analysis task.

This might take some time, depending on how many other queries run on this table.

The good news is that you can easily turn indexes on and off, by using the INVISIBLE and VISIBLE commands.

Setting an index to INVISIBLE will keep the index in the system, but turn it off from being used by the optimiser. This will only affect our session, which is helpful for testing.

You can do this with the following commands:

```
ALTER INDEX indexname INVISIBLE;
```

To turn it back on:

```
ALTER INDEX indexname VISIBLE;
```

Composite Indexes

One final thing to consider before creating a new index is the number of columns.

Are there two or more columns used in your table that would be suitable for an index, such as in a WHERE clause or on a JOIN?

If so, consider using a composite index.

This can work well when both columns are being used in a WHERE clause or a JOIN. By themselves they may not be a highly selective filter, but when combined, they can be highly selective.

If you're using two or more columns, which column would come first?

Deciding which column comes first in a composite index is important, as it can impact the execution plan and if the index is used or not.

In short, the column that is used most often in filtering queries should come first. If there is no column that stands out as being the most used, then choose one that makes the query more selective (has a higher ratio of unique records).

Creating the Index

Let's create the index now.

Choose a column from the spreadsheet that seems to be a good solution for the index. This would be one you have marked in the column that says Index Type.

Now, create the index.

You can do this using the following commands.

B-tree index:

```
CREATE INDEX idx_name ON table(column);
```

Bitmap index:

```
CREATE BITMAP INDEX idx_name ON table(column);
```

If it's a function-based index, replace the word "column" with the function or expression on your column. For example:

```
CREATE INDEX idx_name ON table(UPPER(column));
```

Your index will then be created. It can take some time for the index to be created, depending on how much data is in the table, as Oracle has to add all records to the index. I've seen index creation statements take a few minutes.

Using the example in this guide, I can create an index on the order_date column:

```
CREATE INDEX idx_co_orderdate ON customer_order(order_date);
```

Now, it's time to test.

Run the explain plan for your original query and see the output.

For example, the new explain plan for our original query looks like this:

We can see that the overall cost has remained the same.

Why is that?

Well, we created an index on the order_date column, and Oracle has determined that it's not worth using that index on this column. It's likely due to the distribution of data and the WHERE clause we have specified.

So, let's turn this index off by setting it to INVISIBLE. And we'll create an index on the other column we identified: order_line.order_id

```
ALTER INDEX idx_co_orderdate INVISIBLE;

CREATE INDEX idx_ol_orderid ON order_line(order_id);
```

We can rerun the explain plan and see the result of our new index:

SQL | 1.332 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	BYTES	COST	CPU_COST	ID	TIME
SELECT STATEMENT				99	2673	39	24803594	0	1
SORT		GROUP BY		99	2673	39	24803594	1	1
NESTED LOOPS								2	
NESTED LOOPS				137	3699	38	2898271	3	1
TABLE ACCESS	CUSTOMER_ORDER	FULL		14	224	10	2620765	4	1
Filter Predicates									
CO.ORDER_DATE=TO_DATE(' 2017-06-30 00:00:00', 'syyyy-mm-dd hh24:mi:ss')									
INDEX	IDX_OL_ORDERID	RANGE SCAN		10		1	10171	5	1
Access Predicates									
CO.ORDER_ID=OL.ORDER_ID									
TABLE ACCESS	ORDER_LINE	BY INDEX ROWID		10	110	2	19823	6	1

We can see the overall cost has **gone from 107 down to 39**.

The steps taken have also been changed. We can see that an Index Range Scan has been performed, using IDX_OL_ORDERID.

The majority of the cost has been in the Nested Loops step where the results of both tables are joined.

We've seen that the second index we created has caused a big improvement in query performance. We can now drop the first index we created.

```
DROP INDEX idx_co_orderdate;
```

Conclusion

So, that's how you can analyse the columns for your existing query and work out whether an index will be beneficial.

It's useful to know before creating the index if it will be used by the optimiser, because you don't want to be creating indexes on everything and it can slow the database down if you have more indexes.

Being able to analyse data like this gets easier over time and it can help you write better SQL.

If you have any questions on indexes or on this process, feel free to email me at ben@databasestar.com

Thanks for reading!

Ben Brumm

www.databasestar.com