Five Hints for Optimising SQL

Jonathan Lewis
JL Computer Consultancy
UK

Keywords:

SQL Optimisation Hints Subquery Merge push_pred Unnest push_subq driving_site

Introduction

Adding hints to production code is a practice to be avoided if possible, though it's easy to make the case for emergency patching and hinting is also useful as the basis of a method of generating SQL Plan Baselines. However, notwithstanding (and sometimes because of) the continuing enhancements to the optimizer, there are cases where the only sensible option for dealing with a problem statement is to constrain the broad brush strategy that the optimizer can take in a way that allows it to find a reasonable execution plan in a reasonable time.

This note describes in some detail the use and effects of five of the "classic" hints that I believe are reasonable strategic options to redirect the optimizer when it doesn't choose a path that you consider to be the most appropriate choice.

The Big Five

At the time of writing, a query against the view v$sql_hints on Oracle 12.1.0.2 reports 332 hints – but there are very few which we should really consider as safe for production code, and it's best to view even those as nothing more than a medium-term tool to stabilise performance until the optimizer is able to do a better job with our SQL.

The handful of hints that I tend to rely on for solving problems is basically a set of what I call "structural" queries though in recent years it has become appropriate to label them as "query block" hints. These are hints that give the optimizer some idea of the shape of the best plan without trying to enforce every detail of how it should finalize the plan. The hints (with their negatives where appropriate) are:

- Unnest / no_unnest — Whether or not to unnest subqueries
- Push_subq / no_push_subq — When to handle a subquery that has not been unnested
- Merge / no_merge — Whether to use complex view merging
- Push_pred / no_push_pred — What to do with join predicates to non-merged views
- Driving_site — Where to execute a distributed query

Inevitably there are a few other hints that can be very helpful, but a key point I want to stress is that for production code I avoid what I call "micro-management" hints (such as ***use_nl()***, ***index()***) – attempts to control the optimizer's behaviour to the last little detail; it is very easy

to produce massive instability in performance once you start down the path of micro-managing your execution plans, so it's better not to try.
The rest of this document will be devoted to describing and give examples of these hints.

The Optimizer's Strategy

You can think of the optimizer as working on a "unit of optimization" which consists of nothing more than a simple statement of the form:

```
select  list of columns
from    list of tables
where   list of simple predicates
```

To deal with a more complex query the optimizer stitches together a small number (reduced, preferably, to just one) of such simple blocks. So one of the first steps taken by the optimizer aims to transform your initial query into a this simple form. Consider this example:

```
select
        t1.*,v1.*,t4.*
from
        t1,
        (
        select
                t2.n1, t3.n2, count(*)
        from    t2, t3
        where exists (
                select
                        null
                from    t5
                where   t5.id = t2.n1
                )
        and     t3.n1 = t2.n2
        group by t2.n1, t3.n2
        )       v1,
        t4
where
        v1.n1 = t1.n1
and     t4.n1(+) = v1.n1
;
```

We have an inline view consisting of a two-table join with a subquery correlated to the first table, and from our perspective we have a "simple join" of three objects – t1, v1, and t4. Before it does anything else the optimizer will try to transform this into a five-table join so that it can join them in order one after the other. As part of that process it will generally attempt to eliminate subqueries in a processing known as unnesting.

Looking at the query as it has been presented author of the code may have been thinking (symbolically) of the underlying problem as:

```
( ( t1, ( ( t2, subquery t5 ), t3 ) ), t4 )
```

Take t1, join to it the result of applying the subquery to t2 and joining t3, then join t4.

The optimizer may decide to transform to produce the following:

```
( (  ( ( t1, t2 ), t3 ), {unnested t5} ), t4 )
```

Join t2 to t1, join t3 to the result, join the transformed t5 to the result, then join t4 to the result.

If I decide that the original layout demonstrates the appropriate mechanism, my target is to supply the optimizer with just enough hints to lock it into the order and strategy shown, without trying to dictate every little detail of the plan. My hints would look like this:

```
select
        /*+
            qb_name(main) push_pred(v1@main)
            no_merge(@inline)
            no_unnest(@subq1) push_subq(@subq1)
        */
        t1.*,v1.*,t4.*
from
        t1,
        (
        select  /*+ qb_name(inline) */
                t2.n1, t3.n2, count(*)
        from    t2, t3
        where exists (
                select  /*+ qb_name(subq1) */
                        null
                from    t5
                where   t5.id = t2.n1
                )
        and     t3.n1 = t2.n2
        group by t2.n1, t3.n2
        )       v1,
        t4
where
        v1.n1 = t1.n1
and     t4.n1(+) = v1.n1
;
```

I've labelled the three separate select clauses with a query block name (qb_name hint), told the optimizer that the query block named "inline" should be considered as a separately optimized block (no_merge(@inline)), and the subquery inside that block called "subq1" should be treated as a filter subquery (no_unnest(@subq1)) and applied as early as possible (push_subq(@subq1)).

In some circumstances I might one more hint to tell the optimizer to consider a single join order: t1, v1, t4 using the hint /*+ leading(t1 v1 t4) */; but in this case I've told the optimizer to push the join predicate v1.n1 = t1.n1 inside the view (push_pred(@inline)) – which will make the optimizer do a nested loop from table t1 to view v1, resolving the view for each row it selects from t1.

Having captured 4 of the "big 5" hints in one sample statement, I'll now comments on each of them (and the final diriving_site() hint separately).

Merge / No_merge

This pair of hints apply particularly to "complex view merging", but can be used to "isolate" sections of a query, forcing the optimizer to break one large query into a number of smaller (hence easier) sections. I see two main uses for the hints (and particularly the no_merge) option – one is to help the optimizer get started when handling a query with a large number of table, the other is simply to block a strategy that the optimizer sometimes chooses when it is a bad move.

Consider, in the first case, a query involving 20 tables, with several subqueries. With such a long list it is very easy for the optimizer to pick a very bad starting join order and never reach a good join order; moreover, because of the multiplicative way in which the optimizer estimates selectivity it's very easy for the optimizer to decide after a few tables that the cardinality of the join so far is so small that it doesn't really matter which table to access next. In cases like this we might start by writing a simpler query joining the first four of five tables that we know to be the key to the whole query – once we have got the core of the query working efficiently we can "wrap" it into an inline view with a no_merge hint, and then join the rest of the tables to it, with some confidence that the optimizer will start well and that it can't go far wrong with the remainder of the tables so, for example

```
select  ...
from    t1, t2, t3, ..., t20
where   {various predicates}
and     exists {correlated subquery1}
and     exists {correlated subquery2}
and     column in {non-correlated subquery}
```

Might become

```
with v1 as (
        select  /*+ no_merge cardinality(2000) */ ...
        from    t1, t2, t3, t4, t5
        where   {various predicates{
        and     exists {correlated subquery1}
)
select  ...
from    v1, t6, t7, ..., t20
where   {join conditions to v1}
and     {other join conditions}
and     exists {correlated subquery2}
and     column in {non-correlated subquery}
;
```

I've written the example up using subquery factoring; in earlier versions of Oracle this relevant piece of code would have been written as an inline view, but the "with" clause can help to tidy the SQL up and make it easier to see the logic of what's being done – provided the practice isn't taken to such extremes that the query consists of large number of very small factors.

I've included a cardinality() hint in the factored subquery – it's not fully documented, and it's not commonly realised that it can be applied to a query block rather than to a table or list of tables. This query block usage is probably the safest example of using the hint – the table-related usage is badly understood and prone to mis-use.

As an example of the blocking a badly selected transformation, consider the following query (where I've already included qb_name() hints to name the two separate query blocks):

```
select  /*+ qb_name(main) */
        t1.vc1, avg_val_t1
from    t1,
        (
        Select  /*+ qb_name(inline) */
                id_parent, avg(val) avg_val_t1
        from  t2
        group by
                id_parent
        ) v1
where
        t1.vc2 = 'XYZ'
and     v1.id_parent = t1.id_parent
;
```

There are two basic strategies the optimizer could use to optimize this query, and the choice would depend on its estimate of how much data it had to handle. Whichever choice it makes we might want it to choose the alternative (without rewrite the query beyond hinting it). One option is for Oracle to execute the inline view to generate the aggregate data v1 and then join the result to t1; the other is to join t2 (the underlying table) to t1 and then work out an aggregation of the join that would give the same result.

If I want to "join then aggregate" I would use the merge hint, if I wanted to "aggregate then join" I would use the no_merge hint. There are three different ways in which I could introduce the hint:

a) In the inline view itself I could simply add the hint "merge"
b) In the main query I could reference the view by view name "no_merge(v1)"
c) In the main query I could reference the inline query block name "no_merge(@inline)"

Note particularly the "@" symbol that I use to point a hint at a query block; and note that this is not needed when I reference the view name. (The reference by query block name is the more modern, preferred strategy.)

Push_pred / No_push_pred
Once we start dealing with non-mergeable views and have to join to them there are two strategies that we could use for the join; the first is to create the entire data set for the view and then use that in a merge join or hash join based on the join predicate, or we could "push a join predicate" into the view – in other words for each join value we could add a simple filter predicate to the view definition and create the view result based on that predicate. For

example, if (for convenience) we create a database view called avg_val_view with a definition matching the inline view we used in the previous example, we might see one of two possible execution plans for the following query:

```
select  t1.vc1, avg_val_t1
from    t1, avg_val_view
where   t1.vc2 = 'XYZ'
and     avg_val_view.id_parent = t1.id_parent
;
```

First – if the view is non-mergeable and we don't push the predicate, we can see the join predicate appearing at operation 1, as we do a hash join between table t1 and the entire result set from aggregating t2. This may be sensible, but it may be very expensive to create the entire aggregate:

```
-------------------------------------------------------------------
| Id | Operation            | Name         | Rows  | Bytes | Cost  |
-------------------------------------------------------------------
|  0 | SELECT STATEMENT     |              |     1 |    95 |    27 |
|* 1 |  HASH JOIN           |              |     1 |    95 |    27 |
|* 2 |   TABLE ACCESS FULL  | T1           |     1 |    69 |     2 |
|  3 |   VIEW               | AVG_VAL_VIEW |    32 |   832 |    24 |
|  4 |    HASH GROUP BY     |              |    32 |   224 |    24 |
|  5 |     TABLE ACCESS FULL| T2           |  1024 |  7168 |     5 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("AVG_VAL_VIEW"."ID_PARENT"="T1"."ID_PARENT")
   2 - filter("T1"."VC2"='XYZ')
```

So we may decide to add the hint /*+ push_pred(avg_val_view) */ to the query – we have to use the view-name method since we don't have a query block containing the view; if we were using the inline view from the previous query we could have used the "query block" format /*+ push_pred(@inline) */. The plan from pushing predicates is:

```
-------------------------------------------------------------------
| Id | Operation               | Name        | Rows | Bytes | Cost |
-------------------------------------------------------------------
|  0 | SELECT STATEMENT        |             |    1 |    82 |    7 |
|  1 |  NESTED LOOPS           |             |    1 |    82 |    7 |
|* 2 |   TABLE ACCESS FULL     | T1          |    1 |    69 |    2 |
|  3 |   VIEW PUSHED PREDICATE | AVG_VAL_VIEW|    1 |    13 |    5 |
|* 4 |    FILTER               |             |      |       |      |
|  5 |     SORT AGGREGATE      |             |    1 |     7 |      |
|* 6 |      TABLE ACCESS FULL  | T2          |   32 |   224 |    5 |
-------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("T1"."VC2"='XYZ')
   4 - filter(COUNT(*)>0)
   6 - filter("ID_PARENT"="T1"."ID_PARENT")
```

It would actually be a bad idea in this particular case, but if we could access the rows for a given id_parent in t2 efficiently this query could be much more efficient than the previous plan because it would only aggregate the small number of rows that it was going to need at each point with the smallest row size.

You might not that Oracle has cleverly introduced a filter as operation 4 to eliminate t1 rows where the aggregate would return a row with a zero when there was no matching data. It's details like this that typical programmers forget to think about when trying to transform SQL by hand.

Unnest / No_unnest

The optimizer prefers joins to subqueries, and will generally try to transform a query to turn a subquery into a join (which often means a semi-join – for existence/in – or an anti-join – for not exists/not in). As the optimizer has improved with version many such transformations (or failures to transform) changed from being driven by rules to being driven by cost – and sometimes we want to override the optimizer because we know its costing calculation is bad. Most commonly we might want to write a query with a subquery – to show our intentions – but tell the optimizer to unnest the subquery: it's much safer to take this approach rather than to rewrite the query in unnested form ourselves – I've seen people do the rewrite incorrectly too many times to trust a user-created rewrite. For example:

```
select
        /*+ qb_name(main) unnest(@subq) */
        outer.*
from
        emp outer
where   outer.sal > (
                select
                        /*+ qb_name(subq) unnest */
                        avg(inner.sal)
                from    emp inner
                where
                inner.dept_no = outer.dept_no
        )
;
```

I've show the unnest hint here, and demonstrate the two possible forms – you can either use it in the main query block hint to point it at a give query block name (@subq), or you can use it without a "parameter" in the query block you want unnested. In effect the unnest hint causes Oracle to rewrite the query as:

```
select
        outer.*
from
        (
        select
                dept_no, avg(sal) av_sal
        from    emp
        group by
                dept_no
        )               inner,
        emp             outer
where
        outer.dept_no = inner.dept_no
and     outer.sal > inner.av_sal
;
```

You'll notice that this gives us an in-line aggregate view, so the optimizer could take (or be pushed) one more step into doing complex view merging as well, joining emp to itself before aggregating on a very mess set of columns.

Here's the plan if we unnest:

```
-------------------------------------------------------------------
| Id  | Operation            | Name    | Rows  | Bytes | Cost  |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT     |         |  1000 | 98000 |   114 |
|*  1 |  HASH JOIN           |         |  1000 | 98000 |   114 |
|   2 |   VIEW               | VW_SQ_1 |     6 |   156 |    77 |
|   3 |    HASH GROUP BY     |         |     6 |    48 |    77 |
|   4 |     TABLE ACCESS FULL| EMP     | 20000 |   156K|    36 |
|   5 |   TABLE ACCESS FULL  | EMP     | 20000 |  1406K|    36 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("ITEM_1"="OUTER"."DEPT_NO")
       filter("OUTER"."SAL">"AVG(INNER.SAL)")
```

Notice the appearance at operation 2 of a "view" names VW_SQ_1: there are a number of internal view names that appear in Oracle as it transforms queries – the fact that a view name starts with VW_ is a good clue that it's an internal one. Note, in this particular case that the main work done in the query is the two tablescans of EMP.

Here's the plan if we don't unnest:

```
-----------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost  |
-----------------------------------------------------------
|   0 | SELECT STATEMENT    |      |   167 | 12024 |   252 |
|*  1 |  FILTER             |      |       |       |       |
```

```
|   2 |   TABLE ACCESS FULL | EMP  | 20000 |  1406K|    36 |
|   3 |   SORT AGGREGATE    |      |     1 |     8 |       |
|*  4 |    TABLE ACCESS FULL| EMP  |  3333 | 26664 |    36 |
---------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("OUTER"."SAL"> (SELECT /*+ NO_UNNEST */
              AVG("INNER"."SAL") FROM "EMP" "INNER"
              WHERE "INNER"."DEPT_NO"=:B1))
   4 - filter("INNER"."DEPT_NO"=:B1)
```

The FILTER operation 1 tells us the nominally the optimizer will run the subquery once for every row in the emp table, but the optimizer costing (252) tells us that it thinks that really it will only run the query 7 time in total (7 * 36 = 252): once for the driver and six more times because there are only six departments in my emp table.

Push_subq / No_push_subq

Once we can control whether or not Oracle will unnest a subquery or run it as a filter we can then choose whether the subquery should run early or late. Historically the optimizer would always leave the subqueries to the very end of query operation – but recently the choice of timing has a costing component. "Pushing" a subquery means pushing it down the execution tree – i.e. running it earlier in the plan. To demonstrate this we need a minimum of a two-table join with subquery:

```
select
        /*+ leading(t1 t2) push_subq(@subq) */
        t1.v1
from    t1, t3
where   t1.n2 = 15
and     exists (
                select  --+ qb_name(subq) no_unnest push_subq
                        null
                from    t2
                where   t2.n1 = 15
                and     t2.id = t1.id
        )
and     t3.n1 = t1.n1
and     t3.n2 = 15
;
```

In this query I have a subquery where I've blocked unnesting, so it has to run as a filter subquery (in passing, I've use the alternative, less commonly known, format for hinting: the single-line hint/comment that starts with - - for a comment and - - + for a hint).

I've show the push_subq hint (run the subquery early) in two different ways – first at the top of the query referencing the query block that I want pushed, and then in the subquery itself where it doesn't need a parameter.

As you can see, the subquery is correlate to table t1 and I've told Oracle to examine only the join order t1 -> t3. The effect of the push_subq hint, therefore, is to tell Oracle to run the subquery for each row of t1 that it examines, and join any survivors to t3. The alternative is for Oracle to join t1 to t3 and then run the subquery for every row in the result. Depending on the data and indexes available either option might be the more efficient.

Here are the two plans – first if I don't push the subquery (note the FILTER operation):

```
-------------------------------------------------------------------------
| Id | Operation                     | Name  | Rows | Bytes | Cost  |
-------------------------------------------------------------------------
|  0 | SELECT STATEMENT              |       |    1 |    28 |   289 |
|* 1 |  FILTER                       |       |      |       |       |
|* 2 |   HASH JOIN                   |       |  173 |  4844 |   116 |
|* 3 |    TABLE ACCESS FULL          | T1    |  157 |  3140 |    57 |
|* 4 |    TABLE ACCESS FULL          | T3    |  157 |  1256 |    57 |
|* 5 |   TABLE ACCESS BY INDEX ROWID| T2    |    1 |     8 |     2 |
|* 6 |    INDEX UNIQUE SCAN          | T2_PK |    1 |       |     1 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter( EXISTS (SELECT /*+ QB_NAME ("SUBQ2") NO_UNNEST */ 0
              FROM "T2" "T2" WHERE "T2"."ID"=:B1 AND "T2"."N1"=15))
   2 - access("T3"."N1"="T1"."N1")
   3 - filter("T1"."N2"=15)
   4 - filter("T3"."N2"=15)
   5 - filter("T2"."N1"=15)
   6 - access("T2"."ID"=:B1)
```

Then if I push the subquery

```
-------------------------------------------------------------------------
| Id |Operation                     | Name  | Rows | Bytes | Cost  |
-------------------------------------------------------------------------
|  0 |SELECT STATEMENT              |       |    9 |   252 |   117 |
|* 1 | HASH JOIN                    |       |    9 |   252 |   115 |
|* 2 |  TABLE ACCESS FULL           | T1    |    8 |   160 |    57 |
|* 3 |   TABLE ACCESS BY INDEX ROWID| T2    |    1 |     8 |     2 |
|* 4 |    INDEX UNIQUE SCAN         | T2_PK |    1 |       |     1 |
|* 5 |  TABLE ACCESS FULL           | T3    |  157 |  1256 |    57 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("T3"."N1"="T1"."N1")
   2 - filter("T1"."N2"=15 AND  EXISTS (SELECT /*+ QB_NAME ("SUBQ2")
              PUSH_SUBQ NO_UNNEST */ 0 FROM "T2" "T2"
              WHERE "T2"."ID"=:B1 AND "T2"."N1"=15))
   3 - filter("T2"."N1"=15)
```

```
   4 - access("T2"."ID"=:B1)
   5 - filter("T3"."N2"=15)
```

Notice how the access to t2 has squeezed itself between t1 and t3, and is also indented one place as a clue that it is a subordinate action on t1, but the FILTER from the previous plan has disappeared. This plan is an example of a plan that doesn't follow the well-known "first child first / recursive descent" guideline – Oracle has hidden the FILTER operation and twisted the plan slightly out of its "tradiational" shape as a consequence.

Driving_site

The final hint is for distributed queries, and has no "negative" version. Sometimes the only way you can "tune" a distributed query is to minimise the time spent on network traffic, and this means dictating WHERE the query executes. The driving_site hints lets you make that choice. (Sometimes, having made that choice you also have to include a leading() hint to tell Oracle about the single join order you want it to consider – it's possible for the optimizer to do some very strange things with distributed queries, especially if the instance have different NLS settings).

Consider the following query (I'll fill in the XXXX in the hint shortly):

```
select  /*+ driving_site (XXXX) */
        dh.small_vc,
        da.large_vc
from
        dist_home               dh,
        dist_away@remote_db     da
where
        dh.small_vc like '1%'
and     da.id = dh.id;
```

This query extracts a small amount of data from a table called DIST_HOMT in the local database, and joins it to some data in a table called DIST_AWAY in a remote database; producing a reasonably large number of medium-sized rows. There are basically two obvious plans:

a) nested loop – for each row in dist_home, query dist_away for matching data
b) hash join – create an in-memory hash table from the dist_home data, and then probe it with data from all the rows in dist_away.

The first plan will produce a large number of network round trips – so that's not very good; the second plan will pull a very large amount of data from the remote database if the query operates at the local database (it's only the columns we need, but it will be ALL the rows from the remote database).

Choosing the second plan but executing it at the remote database means we'll send a small parcel of data to the remote database, do the join there to produce a reasonable result set, then send it back to the local database. The network traffic will be minimised without causing an

undesirable increase in other resource usage. To make this plan happen all I needed to do in the query was change the XXXX in the driving_site() hint to reference a table alias from a table in the remote database, in this case driving_site(da).

Here's the execution plan:

```
-------------------------------------------------------------------------
| Id | Operation              | Name     | Rows | Bytes | Inst |IN-OUT|
-------------------------------------------------------------------------
|  0 | SELECT STATEMENT REMOTE|          |  216 | 48600 |      |      |
|* 1 |  HASH JOIN             |          |  216 | 48600 |      |      |
|  2 |   REMOTE               | DIST_HOME|  216 |  4320 |    ! | R->S |
|  3 |   TABLE ACCESS FULL    | DIST_AWAY| 2000 |   400K| TEST |      |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - access("A1"."ID"="A2"."ID")

Remote SQL Information (identified by operation id):
----------------------------------------------------
  2 - SELECT "ID","SMALL_VC" FROM "DIST_HOME" "A2" WHERE "SMALL_VC"
      LIKE '1%' (accessing '!')
```

Notice how the top line (id 0) includes the keyword REMOTE – this tells you that this is the plan from the viewpoint of the remote database/instance that will be executing it. Remember that from its viewpoint the database that we think is the local database is one that it thinks is remote – hence the REMOTE operation 2 which is addressing (our) local table DIST_HOME.

Other key points to note are the appearance of the Inst (instance) and IN-OUT columns. These tell you where each table is located - when a query executes remotely "our" database is tagged only by the name "!".

A nice feature of the execution plan for a distributed query is that you can see how the query has been decomposed for execution at the "remote" site. In this case the other database will be sending our database the query at operation 2 to pull the rows it wants from small_vc so that it can do the join at its site and send the result back to us.

The thing you generally don't want to see in more complex distributed queries is a separate query being generated for each remote table involved in the join – tables that live remotely should be joined remotely with just the join result being pulled back to the local database.

There is a special warning that goes with this hint – it isn't valid for the select statements in "create as select" and "insert as select". There seems to be no good reason for this limitation, but for CTAS and "insert as select" the query has to operate at the site of the table that is receiving the data. This means that you may be able to tune a naked SELECT to perform very well and then find that you can't get the CTAS to use the same execution plan. A typical workaround to this problem is to wrap the select statement into a pipelined function and do a select from table(pipelined_function).

Conclusion

There are literally hundreds of hints available, but as a general guideline there are only a few that are particularly useful and strategically sound. In this article I've listed the five hints that I've long considered to be the ones that are of most help and least risk. I have mentioned a couple of other hints in passing, and know that there are a couple of hints in the newer versions of Oracle that should eventually be added to the list; but the five I've mentioned give a sound basis to work from in understanding the benefits of using the hints that shape the optimizer's strategy for a query without trying to micro-manage it.

Contact address:

Name
JL Computer Consultancy
1 Saxonbury Gardens
Surbiton, Surrey, UK

Phone: +44(0)7973 188785
Email info@jlcomp.demon.co.uk
Internet: http://jonathanlewis.wordpress.com