Dr. Stefan Brass
School of Information Sciences
University of Pittsburgh

July 26, 2001

INFSCI 2711 "Database Analysis and Design" — Example II for Final Exam: Solutions —

General Remarks

- The average was 23.7 points out of 36. The maximum reached was 31.5 points. Every student got 7 bonus points, which moves the average to 30.7.
- Many students said that the time was not sufficient, that they had no idea how the questions would look like, and that query optimization was treated too shortly before the exam in class.
- In retrospecitive, I also regret the system which gives no credit for partially correct answers. If I do "multiple choice" again, I will make sure that only one answer in every exercise has to be checked.
- Of course, I will try to make the exam for this class better (easier).
- Students also said that they learnt a lot going over this exam again in class. So the questions were actually interesting.

Exercise 1 (SQL Errors) 6 Points

Suppose we want to find customers who have rented at least two times a "Star Wars" cassette. Let us assume that there are different such cassettes, but all contain the substring "Star Wars" in their title.

a)
```
SELECT C.NAME
FROM   CUSTOMER C, RENTED X, RENTED Y, VIDEO V
WHERE  C.CUST_NO = X.CUST_NO AND C.CUST_NO = Y.CUST_NO
AND    X.VID_NO = V.VID_NO AND Y.VID_NO = V.VID_NO
AND    V.TITLE LIKE '%Star Wars%'
```

☒ Wrong, Reason: This gives all customers who have rented at least one Star Wars Video.

`X.VID_NO=V.VID_NO` and `Y.VID_NO=V.VID_NO` imply that `X.VID_NO=Y.VID_NO`, so both "`RENTED`" events refer to the same Video-cassette. Also, there is nothing which would prevent `X` and `Y` to point to the same row in the table `RENTED`. In such cases, you would need an inequation like `X.START_DATE <> Y.START_DATE`. However, here this would not solve the problem, since you also need two tuple variables over `VIDEO`. A corrected version is:

```
SELECT C.NAME
FROM   CUSTOMER C, RENTED X, RENTED Y, VIDEO V, VIDEO W
WHERE  C.CUST_NO = X.CUST_NO AND C.CUST_NO = Y.CUST_NO
AND    X.VID_NO = V.VID_NO AND Y.VID_NO = W.VID_NO
AND    V.TITLE LIKE '%Star Wars%' AND W.TITLE LIKE '%Star Wars%'
AND    (X.START_DATE <> Y.START_DATE OR X.VID_NO <> Y.VID_NO)
```

b)
```
SELECT C.NAME
FROM   CUSTOMER C
WHERE  2 <= (SELECT COUNT(*) FROM VIDEO V
             WHERE V.TITLE LIKE '%Star Wars%')
```

☒ Wrong, Reason: This gives all customers if the database contains at least two “Star Wars” video cassettes.

The subquery is uncorrelated to the main query. Of course, we want to count in the subquery the number of Star Wars videos rented by the current customer in the outer query. But the subquery does not refer to `C` at all. A corrected version is:

```
SELECT C.NAME
FROM   CUSTOMER C
WHERE  2 <= (SELECT COUNT(*)
             FROM   RENTED R, VIDEO V
             WHERE  R.CUST_NO = C.CUST_NO
             AND    V.VID_NO = R.VID_NO
             AND    V.TITLE LIKE '%Star Wars%')
```

c)
```
SELECT   C.NAME
FROM     CUSTOMER C, RENTED R, VIDEO V
WHERE    C.CUST_NO = R.CUST_NO AND R.VID_NO = V.VID_NO
AND      V.TITLE LIKE '%Star Wars%'
GROUP BY C.CUST_NO, C.NAME
HAVING   COUNT(*) >= 2
```

☒ Correct

Listing also `CUST_NO` in the `GROUP BY` treats different customers with the same name as different (so if both have rented a “Star Wars” cassette once, their name is not printed. However, if this can really happen. the customer number should also be listed under `SELECT`.

Suppose that now we want to find the cassette which was rented most often. The answer does not have to be unique: If two cassettes were rented 150 times, and no cassette was rented more than 150 times, both of these cassettes should be printed.

Which of these solutions are correct, and which are incorrect?

d)
```
SELECT V.VID_NO, V.TITLE
FROM   VIDEO V
WHERE  V.RENT_COUNT = MAX
```

☒ Wrong, Reason: Syntax error: The aggregation function `MAX` must have an argument. Also, aggregation functions are not allowed under `WHERE` (except inside subqueries).

e)
```
SELECT V.VID_NO, V.TITLE
FROM   VIDEO V
WHERE  RENT_COUNT = (SELECT MAX(X.RENT_COUNT) FROM VIDEO X)
```

☒ Correct

f)
```
SELECT V.VID_NO, V.TITLE
FROM   VIDEO V
WHERE  NOT EXISTS(SELECT * FROM VIDEO X
                  WHERE X.RENT_COUNT > V.RENT_COUNT)
```

☒ Correct

This has confused many students (double negation is always complicated). `V` is the cassette which was rented maximally often if there is no cassette `X` which was rented more often. Suppose that `V` was rented 100 times. Then the subquery is

```
SELECT * FROM VIDEO X
WHERE X.RENT_COUNT > 100
```

And indeed, if this subquery returns a row, then `V` was not rented maximally often. Each cassette returned by this query was rented more often than `V`. If however, the result is empty, no cassette was rented more often than `V`, and `V` should be printed as result of the main query.

Exercise 2 (Disks and Buffering) 6 Points

a) The time needed to access a given sector on a disk consists of three components. Which of these components improve when the disk platters spin faster (e.g. 10000 rpm instead of 5800 rpm)? As always, more than one answer may be correct.

- ☐ Seek time
- ☒ Latency time
- ☒ Transfer time (from disk surface to disk cache/memory)
  Starts when the disk head is over the beginning of the sector/block
  and ends when the disk head is at the end of the block.

The main improvement is in the latency time, the improvement in the transfer time is only a side effect. The seek time does not improve, because the arm is moved by another motor. However, often disks with higher rotation speeds are simply better (more expensive), and also have a shorter seek time, but this does not follow from the higher rotation speed.

b) If you read blocks which are consecutively stored on the disk, how much data can you read per second?

- ☒ 1–20 MByte

Some textbooks say 5 MB/sec, but technology continually advances, so I figured that this range would be safe. However, one student showed me a press notice from Seagate (Dated October 26, 1998) that the 10000rpm Cheetah family drives set a new record for sustained throughput: 28 MBytes/sec. Also the latest models of the Barracuda family have sustained data transfer rates of 25.7 MBytes/sec. I apologize for being a bit outdated. Please notice that the transfer rates of the SCSI bus do not mean that a single disk can produce data at that speed. Usually there is more than one disk on the same bus.

c) Suppose your buffer cache has space for 200 database blocks of 2 KByte each. Let us assume that it contains some useful information, e.g. the root blocks of some indexes. Now you do a full table scan on a table which is 1 MByte long (and has no special declarations). Will this overwrite the entire contents of the buffer cache in Oracle?

- ☒ No. Only some blocks of the buffer cache are used in the full table scan.

We talked in class about sequential flooding of the buffer. With the normal LRU method, after this full table scan, the last blocks of this one table will be in the buffer. This is certainly not useful. Oracle avoids the problem by putting buffer frames used for a full table scan at the front of the LRU queue, so they will be immediately reused. Thus, the full table scan uses the same buffers again and again, and does to touch the other buffer frames in the LRU queue.

d) Suppose you increase the `DB_BLOCK_SIZE` from 2 KByte to 8 KByte. Will an average block access then need 4 times as much time?

- [✓] No, the average access to one 8 KByte block is faster than 4 independent accesses to 2 KByte blocks.

Only one seek is needed here, whereas for reading four independent blocks, four seeks are required.

e) Suppose you do RAID Level 4 (striping with block-wise parity information). You use 5 disks (4 for the data, 1 for the parity blocks). You have mainly single-block read accesses, which distribute well over the 4 disks, so you can process nearly 200 read requests per second (assuming a single disk can process 50 requests per second). Now one of your 4 data disks fails. How many read requests per second can you now process?

- ☐ None. I must first replace the disk.
- ☐ Around 50.
- ☐ Around 100.
- ☐ Around 150.
- ☐ Unchanged: 200.

I apologize for this question. It turned out that not enough information is given to determine the number of block accesses possible after the disk failure. Only the first choice is certainly false: The purpose of the parity blocks is to be able to reconstruct the information if one of the data disks fails. Otherwise, the performance depends on which blocks are accessed:

- If blocks are randomly distributed, and not related to each other, the performance will be around 100: One block from each disk is requested, we first read the blocks from the three functional disks, and then we read the parity block and the blocks from the three remaining disks which build one parity group with the requested block from the damaged disk. From this information, we can reconstruct the requested block from the failed disk.
- If, however, we actually read a file stored in sequential blocks, than we automatically get one parity group at a time and can reconstruct the block from the failed block at no extra cost. However, in this case we read sequential blocks from the disks, and can deliver much more than 200 blocks per second.

f) Suppose the `CUSTOMER` table is quite large, and is accessed very often via the index on its key. Table and index would fit on one disk, but you have another empty disk. What would give you better performance?

☒ Put table and index on different disks.

Exercise 3 (Access Paths) 6 Points

a) Suppose you accidentally deleted an index. Will your application programs which previously used this index still run (maybe slower)?

☒ They will run, but probably slower.

The information in an index is redundant and physical data independence means that you do not have to change your application programs if the physical schema changes.

b) Consider the table `RENTED(`<u>`CUST_NO`</u>`, `<u>`VID_NO`</u>`, `<u>`START_DATE`</u>`, RETURNED_DATE)`. Suppose you have created a B-tree index with the command

```
CREATE INDEX I_RENTED_START_RETURNED ON
             RENTED(START_DATE, RETURNED_DATE)
```

Which of the following conditions can be evaluated using the index? Please check all correct answers.

☒ `START_DATE BETWEEN '15-OCT-99' AND '20-OCT-99'`
☐ `RETURNED_DATE >= '20-OCT-99'`
☐ `RETURNED_DATE - START_DATE > 4`
☒ `START_DATE = '15-OCT-99' AND RETURNED_DATE > '16-OCT-99'`
☐ `START_DATE = '15-OCT-99' OR RETURNED_DATE > '16-OCT-99'`

`START_DATE` is the main ordering criterion for the index entries, `RETURNED_DATE` becomes only important if two entries have the same `START_DATE`. Therefore, you can use this index like an index on `START_DATE` alone, but you cannot use it like an index on `RETURNED_DATE` alone. The condition `RETURNED_DATE - START_DATE > 4` cannot be supported by an index at all, at least not in Oracle. Only conditions which refer to a single attribute and compare it (with =, <, >, <=, >=, `BETWEEN`, and sometimes `LIKE`) with a constant can be supported by an index on that attribute. An index can also be used for joins, but there one table is accessed first, and then you know the attribute value for the current table from that relation, so it can be treated like a constant. Conditions which compare two attributes from the same relation cannot be supported by an index. If you need this, you would have to add the difference `RETURNED_DATE - START_DATE` as a new, redundant column to the table. It would make sense that a system allows also to index expressions on the attributes of a single tuple, but Oracle does not have this feature. The final selection (with `OR`) is wrong, since a full table scan is anyway needed for the right condition, so it would make to sense to use an index for the left condition. This would only give more block accesses.

c) Suppose you store `RENTED` in an index cluster, clustered by `CUST_NO`. Will this impose any limit on the table?

- ☒ If the number of `RENTED` rows for a single customer becomes larger than was assumed in the `SIZE` calculation, performance might suffer.

d) Can it have any use to declare a composed index on all columns of a table? E.g. an index on `VIDEO(VID_NO, TITLE, RENT_COUNT)`? Please check all correct statements.

- ☒ If the key consists of all columns (not in the `VIDEO` example), we still need an index to enforce it (as for any other key).
- ☒ This avoids the TABLE ACCESS (BY INDEX ROWID) whenever the index is usable.

Note that SQL allows duplicate rows in tables. This is slightly more general than mathematical relations (which are sets of tuples/rows), and it is recommended to declare at least one index on every table to avoid this phenomenon (unless you really want duplicate rows and you know what you are doing). If there is no other key, you should choose all columns together as one composed key. Regarding the second answer: The index entries will in this case contain values for all attributes (plus the ROWID). So there is no need here to access the table.

e) Consider the table

```
RENTED(CUST_NO, VID_NO, START_DATE, RETURNED_DATE)
```

There will be many queries refering to currently borrowed cassettes. We want to keep information about all renting events in the past 12 months in the table `RENTED`. Video cassettes are usually borrowed only for a few days. One option would be to partition the table into

```
RENTED_ARCHIVE(CUST_NO, VID_NO, START_DATE, RETURNED_DATE)
   RENTED_OPEN(CUST_NO, VID_NO, START_DATE)
```

Here RENTED_OPEN will contain information only about the currently borrowed cassettes. When a cassette is returned, the entry in RENTED_OPEN is deleted and an entry is written into RENTED_ARCHIVE. What do you think of this solution? Please check all true statements.

- ☐ We will save significant storage space by not including the column RETURNED_DATE in RENTED_OPEN. In this way, we do not have to store the null value explicitly.
- ☒ RENTED_OPEN will be much smaller than RENTED, so a full table scan of RENTED_OPEN might be feasable, whereas a full table scan of RENTED might be prohibitively expensive.
- ☒ Indexes on RENTED_OPEN will be more effective than indexes on the entire table RENTED because they are smaller.
- ☒ If an index on RENTED_OPEN returns multiple ROWIDs, and we need to look up the corresponding rows, the chances that two ROWIDs are in the same block and that this block is still in the cache are higher than for indexes on the entire table RENTED.
- ☐ `CREATE VIEW RENTED_OPEN AS`
  `SELECT * FROM RENTED WHERE RETUNED_DATE IS NULL`
  has the same advantages for performance (efficient query evaluation).

We do not save storage space, because Oracle anyway does not store null values at the end of a row explicitly. RENTED_OPEN will indeed be much smaller than RENTED, since it contains only the events from the last few days as opposed to the last year. Therefore indexes will be smaller. Some students argued that if the table is too small, an index is not useful anymore. This was not what I meant. It is possible that a RENTED_OPEN is smaller than RENTED, but still not small. However, I have to agree that the index height grows only logarithmically with the table size, so that the difference is probably not big. My idea for the next question was that ROWIDs refer to fewer blocks, so the likelihood is greater that two ROWIDs refer to the same block. However, if no rows are deleted from RENTED, then new rows are inserted only at the end, so the rows for still borrowed cassettes are also stored close to each other (except if someone doesn't bring a cassette back for a longer time, then this row will be the only still open row in its block). Views only make query formulation simpler. Query evaluation gets even a bit more complicated.

f) Suppose you access a table by ROWID:

```
SELECT * FROM R WHERE ROWID = 'AAAAkPAA/AAAAADAAL'
```

How many block accesses does this query need? A single row will be not more than 200 Bytes long (shorter than a block). Rows in this table can be updated, and can grow (but still remain less than 200 Bytes). You cannot assume anything about PCTFREE.

☒ Normally one, at most 2 (0 if in cache)

Exercise 4 (Heap Files) 1+4+1=6 Points

a) Suppose it is an option to export the data of a table, recreate the table, and import all the data again. Please check all true statements:

- ☒ Afterwards there will be no migrated rows (at least until new updates are done).
- ☒ It is possible that the rows will be stored afterwards in fewer blocks, because no holes will remain from deleted rows.
- ☒ The ROWIDs of the rows might change during this step.

b) What would be a good INITIAL size and PCTFREE for the table RENTED? It is declared as follows:

```
CREATE TABLE RENTED(CUST_NO NUMBER(6) REFRENCES CUSTOMER,
                    VID_NO NUMBER(4) REFERENCES VIDEO,
                    START_DATE DATE,
                    RETURNED_DATE DATE NULL,
                    PRIMARY KEY(CUST_NO, VID_NO, START_DATE))
```

The CUST_NO values really contain 6 digits (they start with 100001), and the VID_NO values contain 4 digits (starting with 1001). DATE values need 7 data bytes. Note that when rows are inserted, RETURNED_DATE will be null. In Oracle, if the last column is null, no length byte is needed for this column. The table should be designed for 100 000 rows. It will initially be empty and grow through insertions (whenever a customer takes a video out of the shop) and updates (RETURNED_DATE is set when he/she brings the video back). DB_BLOCK_SIZE is 2048 Byte. Please show the main calculations.

Row length when RETURNED_DATE is null (without row directory entry): 20

3 Bytes for Row Header, 1 Byte for Length of CUST_NO, $4 = 6/2 + 1$ Bytes for data of CUST_NO, 1+3 Bytes for VID_NO, 1+7 Bytes for START_DATE.

Row length when RETURNED_DATE is later set (without row directory entry): 28

Now one Byte for the length of `RETURNED_DATE` and 7 Byte for its data are added. The length of a date column is always 7, but Oracle doesn't use the type information here.

`PCTFREE 29`

We assume that a block first will be filled with rows of 20 Byte each, leaving the space reserve. Then the rows will be expanded by 8 Bytes each, filling up the space reserve. If n rows are stored in the block, the total usable space is $n * 28$, and the space reserve is $n * 8$. So `PCTFREE` can be computed as $(n * 8)/(n * 28) = 8/28 = 0.29$. Note however, that 29% is based on the worst case, that the block is first completely filled with the short rows which then grow. If rows start growing before the block is filled, we don't need such a big space reserve. But some books say that we should avoid migrated rows at all costs, so assuming the worst case is probably right.

`INITIAL 3178K`

Note that the space reserve takes care of the 8 bytes growth, so we compute the needed number of blocks with a row length of 20 Bytes. In addition, 2 Bytes per row are needed for the row directory entry. The available space in each block is 2048 Bytes minus 90 Bytes for the block header and 568 Bytes for the space reserve ($(2048 - 90) * 0.29$). This gives 1390 Bytes per block. So we can store $1390/(20+2) = 63$ rows per block. For 100 000 rows, we need $100000/63 = 1588$ blocks, plus one block for the segment header. The initial size must be specified in Bytes (KB/MB), not in blocks. So we finally get $1589 * 2 = 3178$ KByte.

c) Suppose there are only insertions into a table (e.g. `RENTED`). No rows are ever deleted. In addition, all rows have the same length (which is not very big, less than 10% of the block size). Does the value for `PCTUSED` matter?

☒ Under these circumstances, it does not matter. `PCTUSED` is only important for insertions when rows have different lengths and for deletions.

However, `DB_BLOCK_SIZE * (100-PCTUSED-PCTFREE)/100` must still leave space for at least one row, in order to get blocks off the free list. If e.g. `PCTFREE` is `29`, and a row is `28` Bytes long, which is about 1.5% of the block size (without the header), `PCTUSED` may not be greater than `69`. It may be much smaller and this has no effect on the placement of rows in blocks under the above circumstances.

Exercise 5 (Data Dictionary) 6 Points

a) Write an SQL query which lists tables with more than 2% of chained or migrated rows. (The data dictionary counts migrated rows as chained rows, it makes no distinction between the two.)

```
SELECT TABLE_NAME
FROM   TABS
WHERE  CHAIN_CNT/NUM_ROWS > 0.02
```

Note that SQL has no percent notation. Quite a lot of students wrote `2%` on the right hand side which is a syntax error.

b) Write an SQL query which lists for each table all columns on which a single-column index exists. The output should have the columns `TABLE_NAME`, `COLUMN_NAME`, and `INDEX_NAME`, and be sorted by `TABLE_NAME`. Only single-column indexes should be contained in the output.

```
SELECT TABLE_NAME, COLUMNH_NAME, INDEX_NAME
FROM   USER_IND_COLUMNS
WHERE  INDEX_NAME IN (SELECT   C.INDEX_NAME
                      FROM     USER_IND_COLUMNS C
                      GROUP BY C.INDEX_NAME
                      HAVING   COUNT(*) = 1)
```

An alternative solution is e.g.

```
SELECT X.TABLE_NAME, X.COLUMNH_NAME, X.INDEX_NAME
FROM   USER_IND_COLUMNS X
WHERE  NOT EXISTS(SELECT *
                  FROM   USER_IND_COLUMNS Y
                  WHERE  Y.INDEX_NAME = X.INDEX_NAME
                  AND    Y.COLUMN_NAME <> X.COLUMN_NAME)
```

c) Write an SQL query which lists tables such that the indexes on the table need together more disk space than the table itself.

```
SELECT T.SEGMENT_NAME
FROM   USER_SEGMENTS T
WHERE  SEGMENT_TYPE = 'TABLE'
AND    T.BLOCKS < (SELECT SUM(S.BLOCKS)
                   FROM   IND I, USER_SEGMENTS S
                   WHERE  I.TABLE_NAME = T.SEGMENT_NAME
                   AND    I.INDEX_NAME = S.SEGMENT_NAME
                   AND    S.SEGMENT_TYPE = 'INDEX')
```

Unfortunately, the exercise was not very precise whether the number of allocated blocks or the number of actually used blocks was meant. The data dictionary does not contain the number of actually used blocks for indexes (although the number of leaf blocks is a good approximation), so I have used the number of allocated blocks.

Exercise 6 (Query Optimization) 1+1+1+3=6 Points

a) Consider the following query:

```
SELECT *
FROM   CUSTOMER
WHERE  CUST_NO > 200000
AND    NAME = 'Smith'
AND    ADDRESS LIKE '%Pittsburgh%'
```

Which of the following access paths would the rule-based optimizer choose. Here, only one answer is correct (the one ranked highest among the possible ones):

- ☐ Full Table Scan
- ☐ Unique Index on `CUSTOMER(CUST_NO)`
- [✓ ☒] Index on `CUSTOMER(NAME)`
- ☐ Index on `CUSTOMER(ADDRESS)`

Note that the index on `CUSTOMER(ADDRESS)` cannot be used, since the pattern starts with a `%` (i.e. we don't know a prefix of the address).

b) Consider this query:

```
SELECT C.NAME, V.TITLE
FROM   CUSTOMER C, RENTED R, VIDEO V
WHERE  C.CUST_NO = R.CUST_NO
AND    R.VID_NO = V.VID_NO
AND    R.START_DATE = '20-OCT-99'
```

Suppose that the following access paths exist:

- Unique index on `CUSTOMER(CUST_NO)`
- Unique index on `RENTED(CUST_NO, VID_NO, START_DATE)`
- Unique index on `VIDEO(VID_NO)`
- Index on `RENTED(START_DATE)`

- `RENTED` and `VIDEO` are stored together in an index cluster, clustered by `VID_NO`.

Consider the QEP constructed by the rule-based optimizer which starts by accessing `RENTED`. Which table would the rule-based optimizer join first with `RENTED`?

☐ `CUSTOMER`
☒ `VIDEO`

The cluster leads to a higher-ranked access path for `VIDEO`.

c) Consider this query:

```
SELECT R.VID_NO
FROM   RENTED R
WHERE  R.CUST_NO = 123456
AND    R.START_DATE > '20-OCT-99'
```

Suppose that only the index enforcing the key constraint exists, i.e. a unique index on `CUST_NO`, `VID_NO`, `START_DATE`. How would a good QEP look like?

☐ Only a Full Table Scan is possible.
☐ Index scan evaluating `R.CUST_NO = 123456` and then a table access by ROWID.
☒ This query can be answered entirely out of the index. However, only `R.CUST_NO = 123456` would be used for accessing index entries. The table `RENTED` itself will not be accessed.

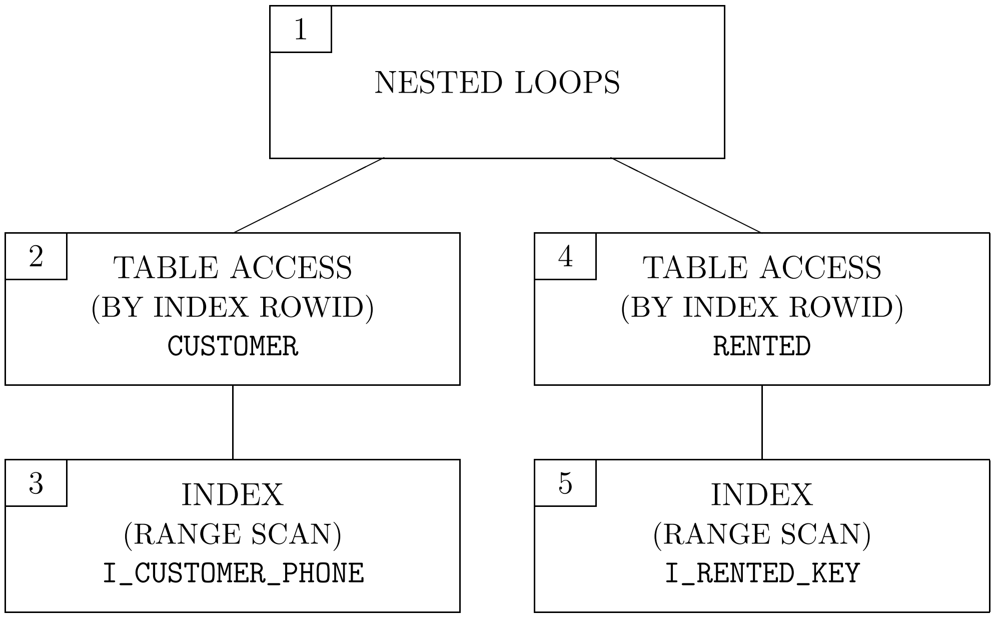
d) Consider the following query:

```
SELECT R.VID_NO, START_DATE
FROM   CUSTOMER C, RENTED R
WHERE  C.PHONE = '624-9404'
AND    R.CUST_NO = C.CUST_NO
AND    RETURNED_DATE IS NULL
```

The following indexes exist:

- Unique index `I_CUSTOMER_KEY` on `CUSTOMER(CUST_NO)`
- Index `I_CUSTOMER_PHONE` on `CUSTOMER(PHONE)`
- Unique index `I_RENTED_KEY` on `RENTED(CUST_NO, VID_NO, START_DATE)`

Please draw the Oracle QEP which the rule-based optimizer constructs starting with access to `CUSTOMER`. You can use the tree notation with interconnected boxes or use one line for every operation with indentation to clarify the structure. You do not have to assign numbers to every node.


1
NESTED LOOPS
2
TABLE ACCESS
(BY INDEX ROWID)
CUSTOMER
4
TABLE ACCESS
(BY INDEX ROWID)
RENTED
3
INDEX
(RANGE SCAN)
I_CUSTOMER_PHONE
5
INDEX
(RANGE SCAN)
I_RENTED_KEY