Kyle Russell

13831056

Assignment two

## Preliminary

This investigation looks at how different structures, tuning techniques and parallelism in Oracle affect the performance of queries. More specifically we will be taking a baseline query as our control and running it on unstructured, indexed, clustered and hash-clustered structures then selecting the best structure to test our statement tuning techniques such as the use of hints and altering the table order then finally we will select the best tuning technique to perform the parallel query option on.

The context of the data is a supermarket database comprised of a Customer table containing customer names, DOB, gender etc., a Product table holding product names, description, and stock, price of the supermarkets products, a Despatch table with product shelf locations, available stock and finally a junction table: Sales holding sales transactions. In this experiment we are primarily interested in the relationship between the Customer and Sales tables which our query will be cantered on throughout.

SELECT c.Cust\_name, s.total\_price

FROM Customer c, Sales s

WHERE s.Cust\_id < 1000

AND s.Cust\_id = c.Cust\_id;

The query chosen as our control in this investigation is shown in figure one. Our query aims to select the names and total sale transaction value of those customers who have an ID less than 1000 which equates to 11% of the 9000 customers. The two tables Customer and Sales are in a one-to-many relationship and so consequently the query performs an equi-join on the Sales table to the Customer table such that the integer joining column is ‘Cust\_id’. The extra conditional used restricts the number of rows we fetch such that we aren’t retrieving every row, more specifically this range provides us with 11% of all 900,000 sales transactions and will allow flexibility when we want to select fewer/more rows. This range will give us a good baseline to work with as the partition of data we select is best optimized typically for cluster structures however I am also interested in narrower (< 5%) and wider (< 22%) partitions of the data to analyse the behaviour of indexes and hash-clusters which we will see in the structures section.

Figure : Baseline query

To ensure consistency and fair testing between experiments, the same data was used throughout by holding two replica data stores CustomerData and SalesData. As well as breaking up any natural ordering in the data. Before each test, the parent tablespace was restarted to prevent caching and as an extra measure to ensure the results are reliable and to eliminate CPU bursts and outliers, each test was ran at least 3 times and then averaged.

## Structures

Unstructured

For this experiment it is necessary to setup a scenario without any structures existing on our tables, more specifically these tables need to be absent of any indexes or cluster structures. This will provide us with a baseline that will allow direct comparisons such that the improvement through each stage should be clear. To setup the initial unstructured experiment the two tables Sales, Customer are created without any indexes or belonging to clusters (Query 4, 5) and then repopulated from their respective data stores, SalesData and CustomerData (Query 6, 7).

**Results**

Figure

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Low-range unstructured trials** | | | | | | | |
| **Trial** | **Count** | **CPU** | **Elapsed** | **Disk** | **Query** | **Current** | **Rows** |
| **First** | 3340 | 0.17 | 0.15 | 4282 | 8013 | 0 | 50084 |
| **Second** | 3342 | 0.16 | 0.15 | 4679 | 8009 | 0 | 50084 |
| **Third** | 3342 | 0.17 | 0.15 | 4679 | 8009 | 0 | 50084 |
| **Average** | **3341.33** | **0.17** | **0.15** | **4546.67** | **8010.33** | **0** | **50084** |

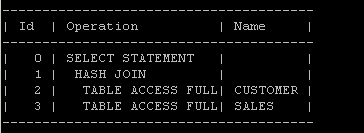
Figure

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mid-range unstructured trials** | | | | | | | |
| **Trial** | **Count** | **CPU** | **Elapsed** | **Disk** | **Query** | **Current** | **Rows** |
| **First** | 6719 | 0.19 | 0.21 | 4679 | 11380 | 0 | 100739 |
| **Second** | 6719 | 0.20 | 0.20 | 4679 | 11356 | 0 | 100739 |
| **Third** | 6717 | 0.19 | 0.19 | 4679 | 11356 | 0 | 100739 |
| **Average** | **6718.33** | **0.19** | **0.20** | **4679** | **11364** | **0** | **100739** |

Figure

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **High-range unstructured trials** | | | | | | | |
| **Trial** | **Count** | **CPU** | **Elapsed** | **Disk** | **Query** | **Current** | **Rows** |
| **First** | 13392 | 0.37 | 0.40 | 4321 | 18008 | 0 | 200822 |
| **Second** | 13392 | 0.39 | 0.37 | 4679 | 18006 | 0 | 200822 |
| **Third** | 13392 | 0.35 | 0.36 | 4679 | 18006 | 0 | 200822 |
| **Average** | **13392** | **0.37** | **0.38** | **4559.67** | **18006.67** | **0** | **200822** |

Figure



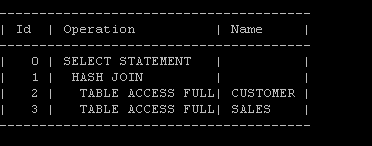


Figure : Low-range execution plan

Figure :Mid-range execution plan

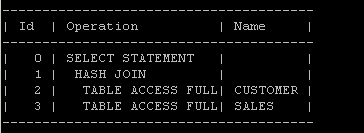


Figure : High-range execution plan

**Analysis**

The results from the unstructured experiments are quite interesting as we see quite a natural exponential progression between all three ranges as seen in figure 2 with all three having the same execution behaviour. More specifically there is quite a close 0.05s difference between the low and mid-range experiments shown in figure 3 and 4, while we have a noticeable 0.18s difference between the mid and high-range experiments. Although the number of rows returned are comparable in difference, the varying difference in elapsed time is not so linear and quite nicely captures the correlated behaviour of our queries elapsed time against the number of rows fetched. Prior to the trials I would have anticipated less of a difference between the mid and high-range tests as Oracles cost effective hash-join is very efficient for joining large partitions however we are still having to perform a full table scan due to the large amount of rows being fetched and this consequently we have a much higher CPU time resulting in higher elapsed time. Another observation can be seen in the variance of the parse statistics in each of the respective ranges tested where the low range experiment has a very stable elapsed time despite varying CPU times which is conversely the same for the mid-range experiments, while there appears to be a lot of variance in the CPU times and elapsed times in each trial and this is predominantly due to the substantially increased processing we are having to do and results in more volatile outcomes.

Indexed

To setup the indexed structure in our experiment we need to add indexes to our existing two joining tables that are involved in the experiment, Sales and Customer. More specifically we have a one-to-many relationship between Customer and Sales which the equi-join in our query manipulates and so we are interested in adding indexes on the joining columns of the two tables, Cust\_id column in Sales and the unique Cust\_id in Customer. Given that the optimizer chooses to utilize theses indexes on our tables we should see a performance bump compared to the unstructured experiment that is absent of any indexes, which is a fair assumption as our query does make good use of these indexed columns as we are searching on these columns and the index is not squashed by additional non-indexed filters in the query. For the initial setup an index is added on Cust\_id in Sales and a unique index is added on Cust\_id in Customer (Query 8, 9), additionally I’m also interested in seeing whether adding an index on a column Sale\_id in Sales (Query 10) that is not searched for in our query has any effect in performance, particularly if we see that the index is squashed in Sales.

**Results**

Figure

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Low-range indexed trials** | | | | | | | |
| **Trial** | **Count** | **CPU** | **Elapsed** | **Disk** | **Query** | **Current** | **Rows** |
| **First** | 3342 | 0.17 | 0.18 | 4273 | 7883 | 0 | 50084 |
| **Second** | 3342 | 0.16 | 0.17 | 4551 | 7879 | 0 | 50084 |
| **Third** | 3342 | 0.16 | 0.16 | 4551 | 7879 | 0 | 50084 |
| **Average** | **3342** | **0.16** | **0.17** | **4458.33** | **7880.33** | **0** | **50084** |

Figure

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mid-range indexed trials** | | | | | | | |
| **Trial** | **Count** | **CPU** | **Elapsed** | **Disk** | **Query** | **Current** | **Rows** |
| **First** | 6719 | 0.18 | 0.19 | 4318 | 11264 | 0 | 100739 |
| **Second** | 6719 | 0.15 | 0.18 | 2767 | 11244 | 0 | 100739 |
| **Third** | 6719 | 0.21 | 0.22 | 4565 | 11240 | 0 | 100739 |
| **Average** | **6719** | **0.18** | **0.20** | **3883.33** | **11249.33** | **0** | **100739** |

Figure

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **High-range indexed trials** | | | | | | | |
| **Trial** | **Count** | **CPU** | **Elapsed** | **Disk** | **Query** | **Current** | **Rows** |
| **First** | 13392 | 0.38 | 0.40 | 4282 | 18010 | 0 | 200822 |
| **Second** | 13392 | 0.34 | 0.35 | 4679 | 18006 | 0 | 200822 |
| **Third** | 13392 | 0.35 | 0.35 | 4679 | 18006 | 0 | 200822 |
| **Average** | **13392** | **0.36** | **0.37** | **4546.67** | **18007.33** | **0** | **200822** |

Figure

Figure 14: Mid-range indexed execution plan

Figure 13: Low-range indexed execution plan

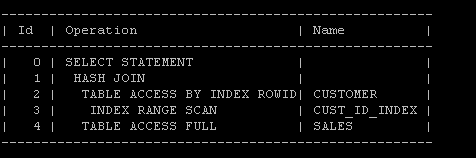
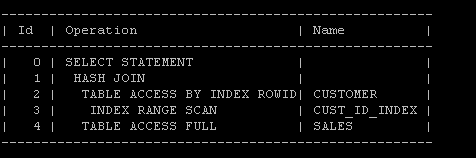
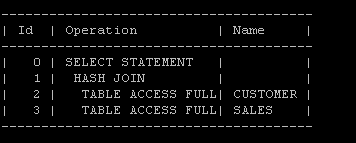


Figure 15: High-range indexed execution plan



**Analysis**

After running the trials for each range in the indexed structure experiment it can be seen that comparatively, there is very little variation in the results of the indexed structure and the unstructured and in the case of the low-range experiment we actually have a minor 0.02s increase in elapsed time as shown in figure 10. In contrast to the unstructured low and mid-range results, the indexed low and mid-range results also exhibit a similar trend as again there is very little variation in these two experiments. If we look at the low-range and mid-range execution plans in figures 13-14 it can be see than that we are actually squashing the index on Sales and performing a full table scan of our large Sales table in conjunction with a hash-join which was conversely the same for the unstructured plans in figures 6-8. The index is being squashed because in each range including the low range experiments, the number of rows being selected and joined is inefficient for a typical nested-loops and index range scan of Sales as there is just too many and it is more cost effective to perform a hash join which is very efficient for joining a large number of rows and running a full table scan so consequently we see very little variation in the experiment outcomes for the indexed structure compared to the unstructured because our ranges are too wide and we are resulting in hash-joins and full table scans. We are using the index on the Customer table in the low and mid-range experiment and so in some of the trials (more notably the mid-range trials of figure 11) we can see a minor performance increase because of the use of this index. Unsurprisingly the high-range query completely squashes both indexes and we have a full table scan occurring in both tables with a hash-join and this is simply because it is far more efficient to perform the hash-join and full table scan on such a large partition of the data as our indexes are useless in the case for such a wide search.

An additional experiment that I was interested to perform was adding an index on a non-searched for column, more specifically the Sale\_id column of Sales. This experiment however in all three range experiments with the extra index, revealed the same execution plan as shown in figures 13-15 and so with no changes in the execution plan we would see almost identical results with perhaps little variance. The assumption that this additional index would have any effect was that the index in Sales was being squashed regardless and perhaps this index on Sale\_id may have an effect, however even in extreme cases as the high-range experiments this is not the case because we are not searching for Sale\_id in our query, nor are we doing any filtering related to this column so the index is ignored.

Clustered

To setup the clustered experiment a cluster is created (query 11) where we use the Cust\_id as the cluster key column. For the physical attributes a high PCTUSED of 90 is specified to accommodate the frequent initial insertions and irregular updates, additionally we allocate at least 4 data blocks for each cluster key equating to the 25K size of each key. A cluster index is added onto the cluster (query 12) with the default INITRANS and MAXTRANS. Next the tables Customer and Sales are re-created belonging to the cluster created (Query 15-16) and then re-populated (Query 6-7) from the appropriate data stores. Since we are only interested in the two tables Sales and Customer and in particular their joining column Cust\_id, so the experiments that follow only manipulate the cluster structure created above without variation as our query only involves these objects specified and so consequently it is only necessary to test on this structure rather than adding unnecessary objects to the cluster.

**Results**

Figure 16

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Low-range clustered trials** | | | | | | | |
| **Trial** | **Count** | **CPU** | **Elapsed** | **Disk** | **Query** | **Current** | **Rows** |
| **First** | 3342 | 0.10 | 0.09 | 466 | 4843 | 0 | 50084 |
| **Second** | 3342 | 0.08 | 0.08 | 502 | 4841 | 0 | 50084 |
| **Third** | 3342 | 0.07 | 0.08 | 502 | 4841 | 0 | 50084 |
| **Average** | **3342** | **0.08** | **0.08** | **490** | **4841.67** | **0** | **50084** |

Figure 17

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mid-range clustered trials** | | | | | | | |
| **Trial** | **Count** | **CPU** | **Elapsed** | **Disk** | **Query** | **Current** | **Rows** |
| **First** | 6719 | 0.15 | 0.18 | 960 | 9722 | 0 | 100739 |
| **Second** | 6719 | 0.16 | 0.17 | 1004 | 9720 | 0 | 100739 |
| **Third** | 6719 | 0.14 | 0.16 | 1004 | 9720 | 0 | 100739 |
| **Average** | **6719** | **0.15** | **0.17** | **989.33** | **9720.67** | **0** | **100739** |

Figure 18

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **High-range clustered trials** | | | | | | | |
| **Trial** | **Count** | **CPU** | **Elapsed** | **Disk** | **Query** | **Current** | **Rows** |
| **First** | 13392 | 0.31 | 0.33 | 1922 | 19399 | 0 | 200822 |
| **Second** | 13392 | 0.30 | 0.28 | 2008 | 19397 | 0 | 200822 |
| **Third** | 13392 | 0.35 | 0.34 | 2008 | 19397 | 0 | 200822 |
| **Average** | **13392** | **0.32** | **0.32** | **1979.33** | **19397.67** | **0** | **200822** |

Figure 19

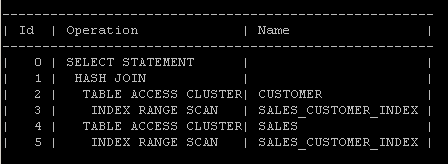
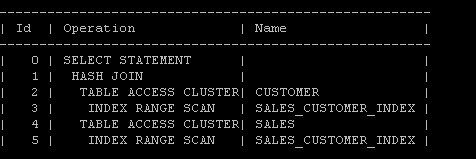


Figure 21: Clustered mid-range execution plan

Figure 20: Clustered low-range execution plan

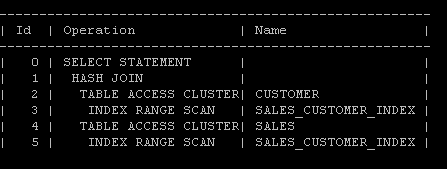


Figure 22: Clustered high-range execution plan

**Analysis**

The clustered results shown in figures 16-19 show a substantial improvement over the previous structures in particular with its next contender, the indexed structure where we see a 0.09s difference in low-range trials, 0.03s difference in mid-range trials and 0.05s difference in high-range trials. To put this into perspective the clustered mid-range trials are comparable to the indexed low-range trials as shown in figures 18 and 10. Additionally we can see a much more linear progression between each range as shown in figure 16 where the previous structures observed exponential progression particularly in the mid to high range comparisons. This is really indicative of a much more balanced structure whose performance is not one sided as the previous structures showed where the unstructured observed better performance in the scenario that we use a high and mid-range due to the hash-joins to accommodate the large partition of rows being joined and similarly the indexed structure will show better results with a much more narrow range to make good use of its index instead of being squashed.

The performance shown by the clustered structure on our query is attributed to our use of the clusters index where our query searches only on the clusters key column so that it is not squashed and we can see that in figures 20-22 we are using the cluster and operating on its index when needing to access either of the two tables and given our choice of ranges, the cluster can make efficient lookups appropriately.

Hash-Clustered

### Statement tuning

Hints

Table order

## Parallel query option

## Scripts

**Query 1: Low range experiment query**

SELECT c.Cust\_name, s.total\_price

FROM Customer c, Sales s

WHERE s.Cust\_id < 500

AND s.Cust\_id = c.Cust\_id;

**Query 2: Mid-range/Baseline experiment query**

SELECT c.Cust\_name, s.total\_price

FROM Customer c, Sales s

WHERE s.Cust\_id < 1000

AND s.Cust\_id = c.Cust\_id;

**Query 3: High-range experiment query**

SELECT c.Cust\_name, s.total\_price

FROM Customer c, Sales s

WHERE s.Cust\_id < 2000

AND s.Cust\_id = c.Cust\_id;

**Query 4: Create unstructured Customer table**

CREATE TABLE Customer

(

Cust\_id number(4) NOT NULL,

Cust\_name varchar2(30) NOT NULL,

Cust\_add varchar2(30),

Phone varchar2(10),

Email varchar2(40),

Join\_date date DEFAULT sysdate,

Points number(3) DEFAULT 0,

Password varchar2(32)

)

PCTFREE 20

PCTUSED 80

TABLESPACE xjw9075\_ts

STORAGE (INITIAL 3M NEXT 500K);

**Query 5: Create unstructured Sales table**

CREATE TABLE Sales

(

Sale\_id Number(6) NOT NULL,

Cust\_id Number(4) NOT NULL,

Prod\_id Number(4) NOT NULL,

Desp\_id Number(4) NOT NULL,

Units\_sold Number(4),

Sale\_date date DEFAULT sysdate,

Total\_price Number(8, 2)

)

PCTFREE 5

PCTUSED 95

TABLESPACE xjw9075\_ts

STORAGE (INITIAL 47M NEXT 5M);

**Query 6: Populate Customer table**

INSERT INTO Customer SELECT \* FROM CustomerData;

**Query 7: Populate Sales table**

INSERT INTO Sales SELECT \* FROM SalesData;

**Query 8: Create index on Cust\_id in Sales**

CREATE INDEX sale\_custid\_index

ON Sales(Cust\_id)

TABLESPACE xjw9075\_ts;

**Query 9: Create unique index on Cust\_id in Customer**

CREATE UNIQUE INDEX cust\_id\_index

ON Customer(Cust\_id)

TABLESPACE xjw9075\_ts;

**Query 10: Create unique index on Sale\_id in Sales**

CREATE UNIQUE INDEX sale\_saleid\_index

ON Sales(Sale\_id)

TABLESPACE xjw9075\_ts;

**Query 11: Create cluster**

CREATE CLUSTER sales\_customer (Cust\_id NUMBER(4))

PCTUSED 90 PCTFREE 10 SIZE 25K

STORAGE(INITIAL 50M NEXT 1M);

**Query 12: Create index on cluster**

CREATE INDEX sales\_customer\_index

ON CLUSTER sales\_customer

INITRANS 2 MAXTRANS 5;

**Query 13: Hash-join hint on baseline**

SELECT /\*+ USE\_HASH(s, c) \*/c.Cust\_name, s.total\_price

FROM Customer c, Sales s

WHERE s.Cust\_id < 1000

AND s.Cust\_id = c.Cust\_id;

**Query 14: Merge-join hint on baseline**

SELECT /\*+ USE\_MERGE(s, c) \*/c.Cust\_name, s.total\_price

FROM Customer c, Sales s

WHERE s.Cust\_id < 1000

AND s.Cust\_id = c.Cust\_id;

**Query 15: Create Customer table in cluster**

CREATE TABLE Customer

(

Cust\_id number(4) NOT NULL,

Cust\_name varchar2(30) NOT NULL,

Cust\_add varchar2(30),

Phone varchar2(10),

Email varchar2(40),

Join\_date date DEFAULT sysdate,

Points number(3) DEFAULT 0,

Password varchar2(32)

)

CLUSTER sales\_customer(Cust\_id);

**Query 16: Create Sales table in cluster**

CREATE TABLE Sales

(

Sale\_id Number(6) NOT NULL,

Cust\_id Number(4) NOT NULL,

Prod\_id Number(4) NOT NULL,

Desp\_id Number(4) NOT NULL,

Units\_sold Number(4),

Sale\_date date DEFAULT sysdate,

Total\_price Number(8, 2)

)

CLUSTER sales\_customer(Cust\_id);

**Query 17: Create hash-cluster**

CREATE CLUSTER sales\_customer (Cust\_id NUMBER(4))

PCTUSED 90 PCTFREE 10 SIZE 25K

STORAGE(INITIAL 50M NEXT 1M)

HASH IS Cust\_id HASHKEYS 9000;

**Query 18: Create hash-cluster with double hash keys**

CREATE CLUSTER sales\_customer (Cust\_id NUMBER(4))

PCTUSED 90 PCTFREE 10 SIZE 25K

STORAGE(INITIAL 50M NEXT 1M)

HASH IS Cust\_id HASHKEYS 18000;

**Query 19: Change table order in baseline join**

SELECT c.Cust\_name, s.total\_price

FROM Sales s, Customer c

WHERE s.Cust\_id < 1000

AND c.Cust\_id = s.Cust\_id;

**Query 20: Get number of CPUs on server**

SELECT value

FROM v$parameter

WHER name LIKE 'cpu\_count'