Query Optimization – Teaching Service

Assignment 1 Report

Integrated Masters in Informatics and Computing Engineering

Database Technologies

Grupo A:
Carlos Albuquerque - up201706735@edu.fe.up.pt
Gonçalo Marantes - up201706917@edu.fe.up.pt
Tito Griné - up201706732@edu.fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

April 18, 2021

Summary

In the Database Administration field, knowing how to perform SQL queries and get the expected results is just the tip of the iceberg. There is a whole other world provided by database dictionaries that involve a much deeper knowledge of the inner workings of a database and its management system. As such, it is of paramount importance to understand and experiment with these concepts for a better learning process.

The goal of this report is to analyze and experiment with various indexes and integrity constraints given a simple, but loaded, relational database running on Oracle SQL. Through the use of different environments, each with its own indexes, we compare possible optimizations for the existing queries and reach conclusions on their advantages and disadvantages. We also dive into some detail on very important concepts such as Clustering Factors, Selectivity and Oracle's Adaptive Query Optimization.

The overall purpose of this project was achieved and we ended up with very interesting analysis and conclusions of the different scenarios used. Our knowledge on the above listed concepts helped us understand a lot of SQL's intricacies and to develop important skills in the Database Optimization subject.

Contents

List of Figures

List of Tables

0 Environment description

In order to analyze several SQL execution plans, as well as assess the impact of indexes in a relational database, the same database schema and data was copied three times, with prefixes *X*, *Y*, and *Z*. The goal was to create three experimentation environments.

- X: no indexes and integrity constraints;
- Y: standard indexes and integrity constrains, i.e. primary and foreign keys;
- Z: with the standard integrity constraints and the extra indexes we found convenient;

The same queries were run on these three environments and their performance was recorded and further analyzed.

0.1 Integrity Constraints

Integrity constraints, as the name implies, ensure data integrity and are used to apply business rules on the database tables. There are only a hand full of constraints available: Primary Key, Foreign Key, Unique, Not Null and Check.

For this report we have only used Primary and Foreign Keys to ensure integrity. The Primary Key constraint forces a column or a set of columns to uniquely identify a row of a table. This constraint is enforced through an unique index on the column/columns used. Therefore, the existence of primary keys can help with the optimization process.

The Foreign Key constraint ensures that an existing relationship between tables is valid. For example, if a Person has a Car, then the column on the Persons table that references the Cars table either has null values or values that reference existing Cars. This is called referential integrity and makes the usage of relational databases much simpler as there is no need to deal with special cases of non existing references. However, there are no indexes involved on a Foreign Key constraint, so it does not help with the optimization process.

0.2 Extra Indexes

For the third environment we have created several indexes along our exploration, so the SQL code, explain plan and description can be found in each question. It is important to note that all these queries were run in the same Z environment, i.e. all created indexes were present.

1 Question 1. Selection and Join

"Show the codigo, designacao, ano_letivo, inscritos, tipo, and turnos for the course 'Bases de Dados' of the program 275."

1.1 The SQL Query

```sql
SELECT ucs.codigo,
   ucs.designacao,
   ucs.curso,
   ocorrencias.ano_letivo,
   ocorrencias.inscritos,
   tipo.tipo,
   tipo.turnos
FROM   xocorrencias ocorrencias
   JOIN xtiposaula tipo
     ON tipo.codigo = ocorrencias.codigo
        AND tipo.ano_letivo = ocorrencias.ano_letivo
        AND tipo.periodo = ocorrencias.periodo
   JOIN xucs ucs
     ON ucs.codigo = ocorrencias.codigo
        AND ucs.designacao = 'Bases de Dados'
        AND ucs.curso = 275;
```

1.2 The answer

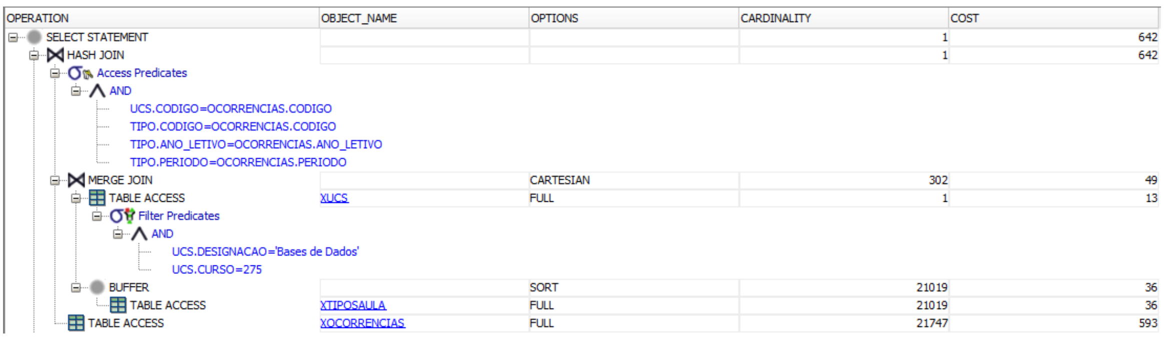
SQL | All Rows Fetched: 6 in 0.096 seconds

	CODIGO	DESIGNACAO	CURSO	ANO_LETIVO	INSCRITOS	TIPO	TURNOS
1	EIC3106	Bases de Dados	275	2003/2004	92	T	1
2	EIC3106	Bases de Dados	275	2003/2004	92	TP	4
3	EIC3106	Bases de Dados	275	2004/2005	114	T	1
4	EIC3106	Bases de Dados	275	2004/2005	114	TP	4
5	EIC3111	Bases de Dados	275	2005/2006	(null)	T	1
6	EIC3111	Bases de Dados	275	2005/2006	(null)	TP	6

Figure 1: Query 1 result

1.3 Execution Plans

1.3.1 X Environment



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	642
HASH JOIN			1	642
Access Predicates				
AND				
UCS.CODIGO=OCORRENCIAS.CODIGO				
TIPO.CODIGO=OCORRENCIAS.CODIGO				
TIPO.ANO_LETIVO=OCORRENCIAS.ANO_LETIVO				
TIPO.PERIODO=OCORRENCIAS.PERIODO				
MERGE JOIN		CARTESIAN	302	49
TABLE ACCESS	XUCS	FULL	1	13
Filter Predicates				
AND				
UCS.DESIGNACAO='Bases de Dados'				
UCS.CURSO=275				
BUFFER		SORT	21019	36
TABLE ACCESS	XTIPOSAULA	FULL	21019	36
TABLE ACCESS	XOCORRENCIAS	FULL	21747	593

Figure 2: Query 1 execution plan in X environment

Since the X schema provides no indexes whatsoever, most data accesses are done via full table scans that try to satisfy the `WHERE` clauses. First off, the DBMS filters the `XUCS` table with the conditions on the `WHERE` clause (`ucs.designacao = 'Bases de Dados' AND ucs.curso = 275`). It then moves on to merge the filtered `XUCS` table with `XTIPOSAULA` based on the `codigo` attribute of both tables. The result of this merge join is then hash-joined with the table `XOCORRENCIAS` based on the `JOIN ... ON` clauses.

Since the DBMS is forced to fully access tables, it is only normal that the overall cost is large.

1.3.2 Y Environment

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	55
HASH JOIN			1	55
Access Predicates				
AND				
TIPO.CODIGO=OCORRENCIAS.CODIGO				
TIPO.ANO_LETIVO=OCORRENCIAS.ANO_LETIVO				
TIPO.PERIODO=OCORRENCIAS.PERIODO				
NESTED LOOPS			1	19
NESTED LOOPS			5	19
TABLE ACCESS	YUCS	FULL	1	13
Filter Predicates				
AND				
UCS.DESIGNACAO='Bases de Dados'				
UCS.CURSO=275				
INDEX	YOCORRENCIAS_PK	RANGE SCAN	5	1
Access Predicates				
UCS.CODIGO=OCORRENCIAS.CODIGO				
TABLE ACCESS	YOCORRENCIAS	BY INDEX ROWID	5	6
TABLE ACCESS	YTIPOSAULA	FULL	21019	36

Figure 3: Query 1 execution plan in Y environment

We can already see that the same query's cost is a lot smaller when basic indexes and constraints are present, even though the query operation steps are similar.

Firstly, the DBMS filters the `YUCS` table with the conditions on the `WHERE` clause (`ucs.designacao = 'Bases de Dados' AND ucs.curso = 275`) by doing a full table scan, just like before. It then moves on to merge the filtered `YUCS` table with `YOCORRENCIAS` based on the `codigo` key. Since there's an index for this attribute in the `YOCORRENCIAS` table, there is no need to access it directly and this is completed at a quite cheap cost. The join is performed via a `NESTED LOOPS` join, which is a general purpose method. The result of this operation is then joined with the `YOCORRENCIA` table, which is accessed `BY ROW ID` since we need some non index attributes from the `SELECT` clause.

Finally, result of this operation is then joined with the `YTIPOSAULA` table, which is read as a full table since there is no index on the `codigo` attribute for this table. The join of these two tables uses a `HASH JOIN` operation with the predicates found in the first `JOIN ... ON` clause.

1.3.3 Z Environment

```sql
DROP INDEX ZTIPOSAULA_IDX_CODIGO;
CREATE INDEX ZTIPOSAULA_IDX_CODIGO ON ZTIPOSAULA(codigo);
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	24
HASH JOIN			1	24
Access Predicates				
AND				
TIPO.CODIGO=OCORRENCIAS.CODIGO				
TIPO.ANO_LETIVO=OCORRENCIAS.ANO_LETIVO				
TIPO.PERIODO=OCORRENCIAS.PERIODO				
NESTED LOOPS			1	24
NESTED LOOPS			6	24
STATISTICS COLLECTOR				
HASH JOIN			1	19
Access Predicates				
UCS.CODIGO=OCORRENCIAS.CODIGO				
NESTED LOOPS			1	19
STATISTICS COLLECTOR				
TABLE ACCESS	ZUCS	FULL	1	13
Filter Predicates				
AND				
UCS.DESIGNACAO='Bases de Dados'				
UCS.CURSO=275				
TABLE ACCESS	ZOCORRENCIAS	BY INDEX ROWID BATCHED	5	6
INDEX	ZOCORRENCIAS_PK	RANGE SCAN	5	1
Access Predicates				
UCS.CODIGO=OCORRENCIAS.CODIGO				
TABLE ACCESS	ZOCORRENCIAS	FULL	5	6
INDEX	ZTIPOSAULA_IDX_CODIGO	RANGE SCAN	6	1
Access Predicates				
TIPO.CODIGO=OCORRENCIAS.CODIGO				
TABLE ACCESS	ZTIPOSAULA	BY INDEX ROWID	1	5
Filter Predicates				
AND				
TIPO.ANO_LETIVO=OCORRENCIAS.ANO_LETIVO				
TIPO.PERIODO=OCORRENCIAS.PERIODO				
TABLE ACCESS	ZTIPOSAULA	FULL	1	5

Figure 4: Query 1 execution plan in Z environment with codigo index

Right from the start, in figure 4, we can see that by simply adding an index (B-Tree by default) on the `codigo` column of the `ZTIPOSAULA` table we get a query that costs about half as much as the previous one. The decision to use a B-Tree index instead of a Bitmap one was mainly due to no performance improvement whatsoever. So in the end, we've decided to opt for a regular one.

The execution plan is a bit more complex, however the basis is the same as the previous one. Firstly, the DBMS filters the `ZUCS` table with the conditions on the `WHERE` clause (`ucs.designacao = 'Bases de Dados' AND ucs.curso = 275`) by doing a full table scan, just like before. It then moves on to merge the filtered `ZUCS` table with `ZOCORRENCIAS` based on the `codigo` key. Since there's an index for this attribute in the `ZOCORRENCIAS` table, this index is used to access other columns from the table. Then these tables are joined via a `NESTED LOOPS` join.

The result would then be joined via a `HASH JOIN` with the full `ZOCORRENCIAS` table. This actually doesn't happen due to a feature introduced in Oracle Database 12c called *Adaptive Query Optimization*. *Adaptive Query Optimization* is a set of capabilities that enable the optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better statistics. In this case, there is no need to join with the full `ZOCORRENCIAS` table, since all the tuples we need are already present from the previous table access `BY INDEX ROWID BATCHED`. The "resulting" table is then joined with the `ZTIPOSAULA` table using the `codigo` index we've created to access tuples `BY INDEX ROWID`.

Finally, the resulting table would be merged with the `ZTIPOSAULA` full table. This doesn't happen, for the same reason we've explained above. Since the `ZTIPOSAULA` was already accessed `BY INDEX ROWID` previously, then the optimizer changes execution strategy while running the query and understands that there is no need to join with the full table.

We can go the extra mile and add more indexes to the filtered columns in the `ZUCS` table.

```sql
DROP INDEX ZUCS_IDX_CURSO_DESIGNACAO;
CREATE INDEX ZUCS_IDX_CURSO_DESIGNACAO ON ZUCS(curso, designacao);
```

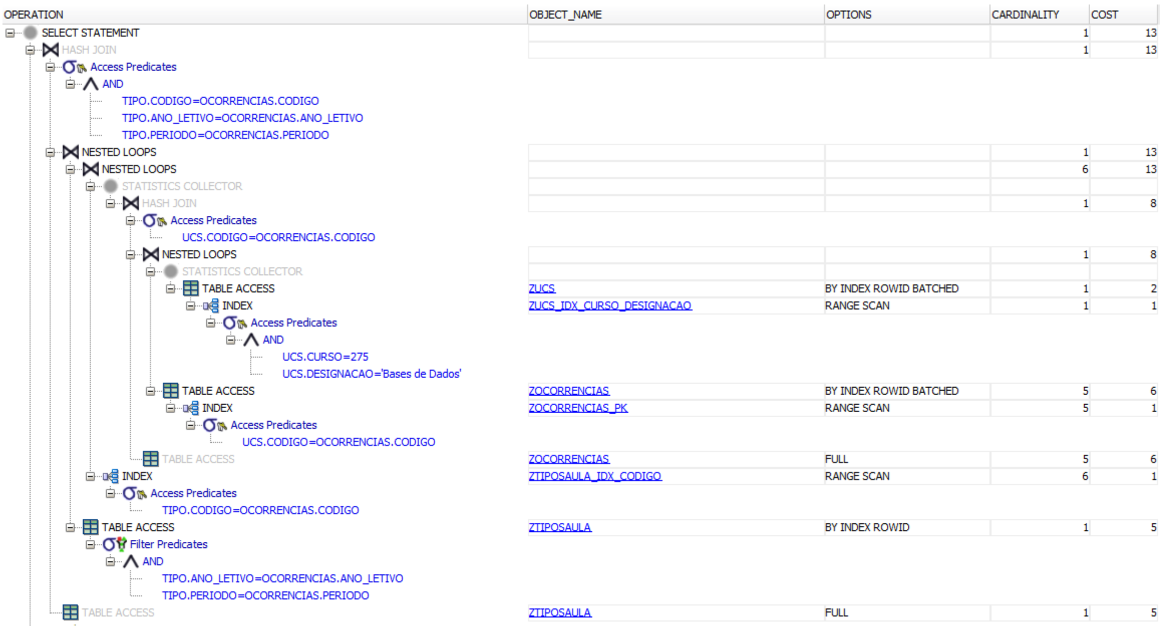


Figure 5: Query 1 execution plan in Z environment with more indexes

We can see from figure 5 that the only change regarding the previous version is that the `ZUCS` table is accessed `BY INDEX ROWID` using the newly created composite index `ZUCS_IDX_CURSO_DESIGNACAO`. This decreases the cost almost by half.

1.4 Execution Time

X Environment	Y Environment	Z Environment
0.066	0.029	0.022

Table 1: Query 1 average execution times in all environments (in seconds)

2 Question 2. Aggregation

"How many class hours of each type did the program 233 planned in year 2004/2005?"

2.1 The SQL Query

```sql
SELECT ucs.curso,
       tipo.ano_letivo,
       tipo.tipo,
       SUM(turnos * horas_turno) AS total_horas
FROM   xucs ucs
       JOIN xtiposaula tipo
         ON tipo.codigo = ucs.codigo
WHERE  ucs.curso = 233
       AND tipo.ano_letivo = '2004/2005'
GROUP  BY ucs.curso,
          tipo.ano_letivo,
          tipo.tipo;
```

2.2 The answer

SQL | All Rows Fetched: 3 in 0,6 seconds

	CURSO	ANO_LETIVO	TIPO	TOTAL_HORAS
1	233	2004/2005	P	581,5
2	233	2004/2005	TP	697,5
3	233	2004/2005	T	308

Figure 6: Query 2 result

2.3 Execution Plans

2.3.1 X Environment

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			50	5
HASH		GROUP BY	50	5
HASH JOIN			49	611
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
TABLE ACCESS	XUCS	FULL	13	504
Filter Predicates				
UCS.CURSO=233				
TABLE ACCESS	XTIPOSAULA	FULL	36	1671
Filter Predicates				
TIPO.ANO_LETIVO='2004/2005'				

Figure 7: Query 2 execution plan in X environment

Starting off with environment X, the first step is to filter both `XUCS` and `XTIPOSAULA` tables to reduce their sizes and reduce the cost of the join operation. Since there are no indexes at all, especially on `XUCS.CURSO` and `XTIPOSAULA.ANO_LETIVO`, the only way to filter the tables is by performing a Full Table Scan and check each row of the table.

Note that filtering `XTIPOSAULA` returns 1671 rows, while the whole table has 21019 rows, which makes the subsequent join much faster. However, this represents the hardest operation of the query, having a cost of 36.

When it comes to the join operation, there are some possibilities available. The system could have performed a Nested Loops Join or a Sort-Merge Join, but instead it chose a Hash Join. We believe the main reason is the filtered tables size. Since the tables are not small, to sort them might be too heavy and to have one of them totally in memory might be unfeasible. Hence, the Hash Join becomes the best solution with just a cost of 1 in this query.

Finally, since we need to `GROUP BY` our results, an Hash operation is executed to place all similiar `(CURSO, ANO_LETIVO, TIPO)` tuples in the same bucket and then summing the product of `TURNOS` by `HORAS_TURNO` in each of those buckets. A very interesting use for a hash function in our opinion!

2.3.2 Y Environment

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			50	5
HASH		GROUP BY	50	5
HASH JOIN			49	611
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
TABLE ACCESS	YUCS	FULL	13	504
Filter Predicates				
UCS.CURSO=233				
TABLE ACCESS	YTIPOSAULA	FULL	36	1671
Filter Predicates				
TIPO.ANO_LETIVO='2004/2005'				

Figure 8: Query 2 execution plan in Y environment

This plan is exactly the same as the above for environment X. The reason is that there are no indexes on `YUCS.CURSO` and `YTIPOSAULA.ANO_LETIVO`, which would replace the Full Table Scan with a Table Access by Rowid after performing a Full/Range Index Scan.

In addition, although `YUCS.CODIGO` has a unique index (created by the primary key constraint), since `YTIPOSAULA .CODIGO` does not have an index the join cannot be optimized. It still is cheaper to filter the tables and then join normally than to perform a join by indexes and only then fetch the rows from the tables by rowid.

2.3.3 Z Environment

```sql
DROP INDEX ZUCS_IDX_CURSO_CODIGO;
CREATE UNIQUE INDEX ZUCS_IDX_CURSO_CODIGO ON ZUCS(curso, codigo);
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			37	5
HASH		GROUP BY	37	5
NESTED LOOPS			36	611
TABLE ACCESS	ZTIPOSAULA	FULL	36	1671
Filter Predicates				
TIPO.ANO_LETIVO='2004/2005'				
INDEX	ZUCS_IDX_CURSO_C...	UNIQUE SCAN	0	1
Access Predicates				
AND				
UCS.CURSO=233				
TIPO.CODIGO=UCS.CODIGO				

Figure 9: Query 2 execution plan in a not fully optimized Z environment

Since we only need the columns `CURSO` (in the `SELECT` and `WHERE` clauses) and `CODIGO` (for the `JOIN`) from the table `ZUCS`, we decided to create a B-Tree index on both columns simultaneously, thus eliminating the need to access the table `ZUCS`. Furthermore, `CODIGO` is a primary key, which makes the pair `(CURSO, CODIGO)` always unique. Based on this information, we made the index unique, thus allowing for a quick Unique Scan on the index which greatly improved the performance of the query. Note that only a B-Tree index can be unique in Oracle SQL, so we did not even consider a non-unique Bitmap index as it would be certainly slower. Finally, the index is joined to `ZTIPOSAULA` through the column `CODIGO` using the Nested Loops Join. This is a good option because the index is very small, having only two columns and 83 rows that satisfy the condition `ZUCS.CURSO = 433`, so the Nested Loops is faster than other join operations.

Note: We believe the cost estimate for the Unique Scan operation is wrong. As stated before, there are 83 rows returned by that index access, which is a lot more than the cardinality of 1 estimated. That said, we ran a query that only used the index and achieved a cost of 2 and a cardinality of 83. Therefore, the estimated cost for the whole query should be 39.

As stated in the analysis for environment Y, this query could be further optimized with an index on `ZTIPOSAULA.ANO_LETIVO`:

```sql
DROP INDEX ZTIPOSAULA_IDX_ANO_LETIVO;
CREATE INDEX ZTIPOSAULA_IDX_ANO_LETIVO ON ZTIPOSAULA(ano_letivo);
```

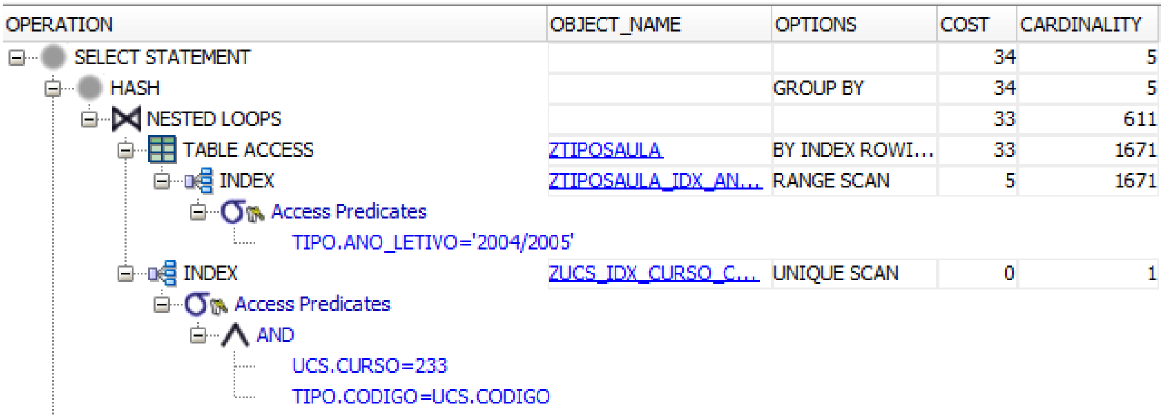


Figure 10: Query 2 execution plan in a fully optimized Z environment

As figure 10 shows, the `ZTIPOSAULA_IDX_ANO_LETIVO` index would allow for a Range Scan on the index with a subsequent table access By Rowid. This would lead to an overall cost of 34. However, since there is not much gain and there are other indexes that also use this column, we decided not to keep it for other questions in order to save space and time that would be otherwise wasted maintaining that index (assuming a fully operational database on a real world system). Although not better, a Bitmap index works similarly but costs a tiny bit more. Since there will be one used for question 5, it could be reused in a real world scenario, thus providing a better alternative than a completely new index.

2.4 Execution Time

X Environment	Y Environment	Z Env. (cost 37)	Z Env. (cost 34)
0.522	0.038	0.026	0.022

Table 2: Query 2 average execution times in all environments (in seconds)

3 Question 3. Negation

"Which courses (show the code) did have occurrences planned but did not get service assigned in year 2003/2004?"

3.1 Not In

Use not in.

3.1.1 The SQL Query

```sql
SELECT DISTINCT ocorrencias.codigo
FROM    xocorrencias ocorrencias
WHERE   ocorrencias.ano_letivo = '2003/2004'
        AND ocorrencias.codigo NOT IN (SELECT aulas.codigo
                                        FROM   xtiposaula aulas
                                               join xdsd dsd
                                                 ON aulas.id = dsd.id
                                        WHERE  aulas.ano_letivo = '2003/2004');
```

3.1.2 The answer

SQL | All Rows Fetched: 138 in 1.355 seconds

	CODIGO
1	EQ418
2	MEMT131
3	EIC4221
4	MEEC1053
5	MEM157
6	EIC4222
7	EIC4220
8	MDI1205
9	MEM181
10	CI027

Figure 11: Query 3.a result

3.1.3 Execution Plans

3.1.3.1 X Environment

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			657	1
HASH		UNIQUE	657	1
HASH JOIN		ANTI	656	10
Access Predicates				
OCORRENCIAS.CODIGO=CODIGO				
TABLE ACCESS	XOCORRENCIAS	FULL	593	1028
Filter Predicates				
OCORRENCIAS.ANO_LETIVO='2003/2004'				
VIEW	SYS.VW_NSO_1		63	1588
HASH JOIN		SEMI	63	1588
Access Predicates				
AULAS.ID=DSD.ID				
TABLE ACCESS	XTIPOSAULA	FULL	36	1588
Filter Predicates				
AULAS.ANO_LETIVO='2003/2004'				
TABLE ACCESS	XDSD	FULL	27	27385

Figure 12: Query 3.a execution plan in environment X

Given the execution plan, we can gather that first, a `FULL TABLE SCAN` is performed of the tables `XDSD` and `XTIPOSAULA`, with this last one only keeping the rows that satisfy the `WHERE` clause of having `ANO_LETIVO=2003/2004`. A `SEMI HASH JOIN` is used to combine both tables, only keeping rows that have a matching `ID` on both tables, with the result being stored in a view. A `FULL TABLE SCAN` is also performed for the `XOCORRENCIAS` table, again filtering for rows that obey the `WHERE` clause. It ends by performing an `ANTI HASH JOIN` on the `CODIGO` column, that is, a row is kept from `XOCORRENCIAS` if its `CODIGO` doesn't match any row in the view previously created.

Since tables `XTIPOSAULA` and `XDSD` are densely packed, less than 200 blocks, the operations on these tables are not very expensive. However, data from table `XOCORRENCIAS` is highly spread, occupying 2181 blocks when the number of rows is in the same range as `XTIPOSAULA`. The scans and joins on this table are very expensive and make up the bulk of the cost of this query.

3.1.3.2 Y Environment

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			86	1
HASH		UNIQUE	86	1
HASH JOIN		ANTI	85	10
Access Predicates				
OCORRENCIAS.CODIGO=CODIGO				
INDEX	YOCORRENCIAS_PK	FAST FULL SCAN	27	1028
Filter Predicates				
OCORRENCIAS.ANO_LETIVO='2003/2004'				
VIEW	SYS.VW_NSO_1		58	1588
HASH JOIN		SEMI	58	1588
Access Predicates				
AULAS.ID=DSD.ID				
TABLE ACCESS	YTIPOSAULA	FULL	36	1588
Filter Predicates				
AULAS.ANO_LETIVO='2003/2004'				
INDEX	YDSD_PK	FAST FULL SCAN	22	27385

Figure 13: Query 3.a execution plan in environment Y

In comparison with the previous execution plan, we can see that the indexes created on the primary keys had an effect on this execution plan. The primary key on table `YDSD` allowed for a `FAST FULL INDEX SCAN`, and, given that the column we were interested in was the primary key one, the values in the index were enough, and no table access was needed. Similarly, the primary key on table `YOCORRENCIAS`, with `CODIGO`, `PERIODO` and `ANO_LETIVO`, made it so no table access was needed in order to get the `CODIGO`, filter on `ANO_LETIVO`, and subsequently perform the `ANTI HASH JOIN` with the created view. The rest of the execution plan remains the same.

We can also see that now that we have indexes to access all tables, the effects of having data sparse among many blocks is far less significant and as such, operations on table `YOCORRENCIAS` are no longer the biggest cost factor in the query. In fact, the vast majority of the improvement on cost comes from operations on this table not having to perform `FULL TABLE SCANS`.

3.1.3.3 Z Environment

```sql
DROP INDEX ZTIPOSAULA_IDX_ANO_LETIVO_ID_CODIGO;
CREATE TABLE ZTIPOSAULA_IDX_ANO_LETIVO_ID_CODIGO ON ZTIPOSAULA(ano_letivo, id, codigo)
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			59	1
HASH		UNIQUE	59	1
HASH JOIN		ANTI	58	10
Access Predicates				
OCORRENCIAS.CODIGO=CODIGO				
INDEX	ZOCORRENCIAS_PK	FAST FULL SCAN	27	1028
Filter Predicates				
OCORRENCIAS.ANO_LETIVO='2003/2004'				
VIEW	SYS.VW_NSO_1		31	1588
HASH JOIN		SEMI	31	1588
Access Predicates				
AULAS.ID=DSD.ID				
INDEX	ZTIPOSAULA_IDX_ANO_LETIVO_I...	RANGE SCAN	9	1588
Access Predicates				
AULAS.ANO_LETIVO='2003/2004'				
INDEX	ZDSD_PK	FAST FULL SCAN	22	27385

Figure 14: Query 3.a execution plan in environment Z

Finally, with the index on table `ZTIPOSAULA` on `ANO_LETIVO`, `ID` and `CODIGO`, we have a good improvement, since the original table doesn't have to be accessed, having the index be enough to perform the filter from the where clause on `ANO_LETIVO`, match the `ID` with the `ZDSD IDs` and retrieve the `CODIGO` necessary for the subsequent `HASH JOIN`.

A normal B-Tree index was chosen since using a Bitmap index wouldn't make much sense given that the cardinality of `ID` and `CODIGO` is high as they are used to uniquely identify other tables. Also the fact that the index is in this particular order (ANO_LETIVO, ID, CODIGO) allows for all operations to be made sequentially, without having to scan the index multiple times, but instead use the resulting set from the previous operation. So we first access the index and match ANO_LETIVO with 2003/2004. We can then continue traversing the index matching the ID with the ID from ZDSD table. Finally, we just get the CODIGO that we want from the select statement to then match with ZOCORRENCIAS.

The cost improvement relative to environment Y is due to the fact that table ZTIPOSAULA doesn't have to be accessed to filter for ANO_LETIVO or retrieve the CODIGO.

3.1.4 Execution Time

X Environment	Y Environment	Z Environment
0.162	0.154	0.153

Table 3: Query 3.a average execution times in all environments (in seconds)

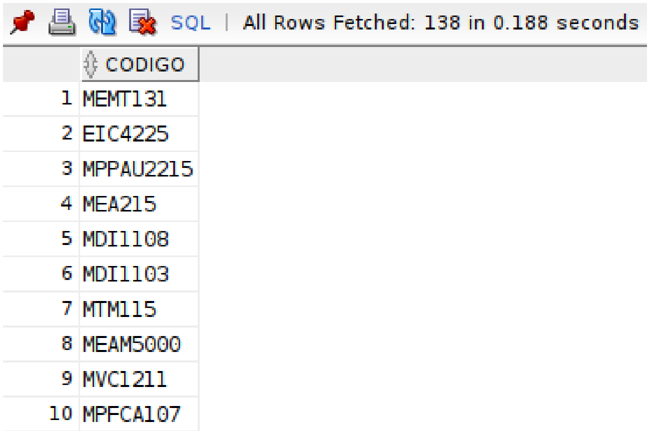
3.2 External Join and Is NULL

Use external join and is null.

3.2.1 The SQL Query

```sql
SELECT  codigo
FROM    (SELECT DISTINCT ocorrencias.codigo codigo,
                         aulas.id          id
         FROM   xocorrencias ocorrencias
                LEFT OUTER JOIN (SELECT aulas.codigo codigo,
                                        aulas.id     id
                                 FROM   xtiposaula aulas
                                        JOIN xdsd dsd
                                          ON aulas.id = dsd.id
                                 WHERE  aulas.ano_letivo = '2003/2004') aulas
                             ON ocorrencias.codigo = aulas.codigo
         WHERE  ocorrencias.ano_letivo = '2003/2004')
WHERE   id IS NULL;
```

3.2.2 The answer


SQL | All Rows Fetched: 138 in 0.188 seconds

	CODIGO
1	MEMT131
2	EIC4225
3	MPPAU2215
4	MEA215
5	MDI1108
6	MDI1103
7	MTM115
8	MEAM5000
9	MVC1211
10	MPFCA107

Figure 15: Query 3.b result

3.2.3 Execution Plans

3.2.3.1 X Environment

The execution path (see figure 16) shows that first, two FULL TABLE SCANs are performed for the tables XDSD and XTIPOSAULA, where the latter is done filtering on ANO_LETIVO being 2003/2004. The two are joined using a SEMI HASH JOIN, matching both IDs. The result is stored in a view that is joined with the results from a full table

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			657	1
VIEW			657	1
HASH		UNIQUE	657	1
FILTER				
Filter Predicates				
AULAS.ID IS NULL				
HASH JOIN		OUTER	656	1
Access Predicates				
OCORRENCIAS.CODIGO=AULAS.CODIGO(+)				
TABLE ACCESS	XOCORRENCIAS	FULL	593	1028
Filter Predicates				
OCORRENCIAS.ANO_LETIVO='2003/2004'				
VIEW			63	1588
HASH JOIN		SEMI	63	1588
Access Predicates				
AULAS.ID=DSD.ID				
TABLE ACCESS	XTIPOSAULA	FULL	36	1588
Filter Predicates				
AULAS.ANO_LETIVO='2003/2004'				
TABLE ACCESS	XDSD	FULL	27	27385

Figure 16: Query 3.b execution plan in environment X

access of the table XOCORRENCIAS, filtered as well on ANO_LETIVO being 2003/2004. This join is an OUTER HASH JOIN, specifically a LEFT JOIN, meaning that it keeps all the rows from the first set and associates the ID of the second set when there is a match on CODIGO, otherwise it's set to NULL. Finally, the resulting set is filtered by rows that have the ID as NULL and duplicates are removed using a UNIQUE HASH.

For the same reason as explained in question 3.a, the operations on table XOCORRENCIAS, namely the FULL TABLE SCAN with filtering and the HASH JOIN, are responsible for most of the estimated cost for this query, again, due to the data's high sparsity on this table.

3.2.3.2 Y Environment

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			86	10
VIEW			86	10
HASH		UNIQUE	86	10
FILTER				
Filter Predicates				
AULAS.ID IS NULL				
HASH JOIN		OUTER	85	10
Access Predicates				
OCORRENCIAS.CODIGO=AULAS.CODIGO(+)				
INDEX	YOCORRENCIAS_PK	FAST FULL SCAN	27	1028
Filter Predicates				
OCORRENCIAS.ANO_LETIVO='2003/2004'				
VIEW			58	1588
HASH JOIN		SEMI	58	1588
Access Predicates				
AULAS.ID=DSD.ID				
TABLE ACCESS	YTIPOSAULA	FULL	36	1588
Filter Predicates				
AULAS.ANO_LETIVO='2003/2004'				
INDEX	YDSD_PK	FAST FULL SCAN	22	27385

Figure 17: Query 3.b execution plan in environment Y

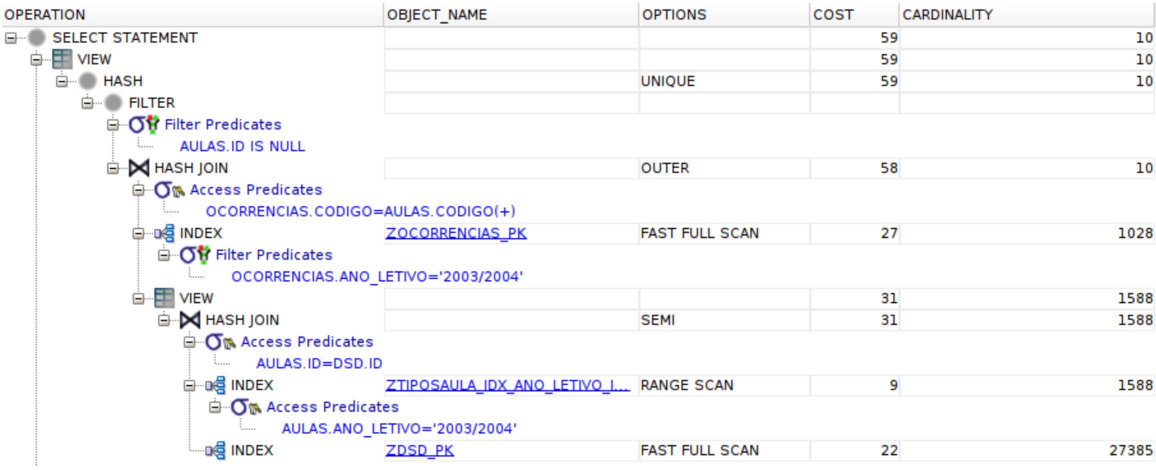
With the standard integrity constraints, the execution plan differs from the previous in two places. First the index on the primary key of YDSD can be used to retrieve the respective ID, and therefore not require any access to the actual table, doing only a FAST FULL INDEX SCAN instead. Secondly, and most significantly, with an index on CODIGO and ANO_LETIVO from the primary key of YOCORRENCIAS, it can also be used to both retrieve the CODIGO and perform the filter on ANO_LETIVO without ever accessing the table. The rest of the plan is the same as the previous.

Again, the biggest improvement coming from the integrity constraints is derived by not having to do a FULL TABLE SCAN on ZOCORRENCIAS.

3.2.3.3 Z Environment

```sql
DROP INDEX ZTIPOSAULA_IDX_ANO_LETIVO_ID_CODIGO;
CREATE INDEX ZTIPOSAULA_IDX_ANO_LETIVO_ID_CODIGO ON ZTIPOSAULA(ano_letivo, id, codigo);
```

With an additional standard index on table ZTIPOSAULA on ANO_LETIVO, ID and CODIGO, the table access done to this table is no longer necessary, the filtering on ANO_LETIVO can be done through the index, as well as the match



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			59	10
VIEW			59	10
HASH		UNIQUE	59	10
FILTER				
Filter Predicates				
AULAS.ID IS NULL				
HASH JOIN		OUTER	58	10
Access Predicates				
OCORRENCIAS.CODIGO=AULAS.CODIGO(+)				
INDEX	ZOCORRENCIAS_PK	FAST FULL SCAN	27	1028
Filter Predicates				
OCORRENCIAS.ANO_LETIVO='2003/2004'				
VIEW			31	1588
HASH JOIN		SEMI	31	1588
Access Predicates				
AULAS.ID=DSD.ID				
INDEX	ZTIPOSAULA_IDX_ANO_LETIVO_I...	RANGE SCAN	9	1588
Access Predicates				
AULAS.ANO_LETIVO='2003/2004'				
INDEX	ZDSD_PK	FAST FULL SCAN	22	27385

Figure 18: Query 3.b execution plan in environment Z

on `ID` with the `ZDSD` table. Since the column `CODIGO` is also contained in the index, it can be used to retrieve said column and, therefore, no access to the original table has to be performed.

It's important to note that, for the same reason as mentioned in 3.a, the ordering on the columns in the index is important to reach a better estimated cost, so the optimizer can perform all the necessary operations using the index and without needing to re scan it multiple times. The rest of the execution plan is the same as the previous.

3.2.4 Execution Time

X Environment	Y Environment	Z Environment
0.148	0.147	0.145

Table 4: Query 3.b average execution times in all environments (in seconds)

4 Question 4.

"Who is the professor with more class hours for each type of class, in the academic year 2003/2004? Show the number and name of the professor, the type of class and the total of class hours times the factor."

4.1 The SQL Query

```
SELECT docentes.nr,
       docentes.nome,
       dsd.fator * dsd.horas AS total_horas_semanais,
       tipo.tipo
FROM   xdsd dsd
       JOIN xdocentes docentes
         ON docentes.nr = dsd.nr
       JOIN xtiposaula tipo
         ON tipo.id = dsd.id
            AND tipo.ano_letivo = '2003/2004'
WHERE  dsd.fator IS NOT NULL
       AND dsd.fator * dsd.horas = (SELECT Max(sub_dsd.fator * sub_dsd.horas)
                                    FROM   xdsd sub_dsd
                                           JOIN xtiposaula sub_tipo
                                             ON sub_tipo.id = sub_dsd.id
                                                AND sub_tipo.ano_letivo = '2003/2004'
                                    WHERE  sub_dsd.fator IS NOT NULL
                                           AND sub_tipo.tipo = tipo.tipo)
ORDER  BY tipo.tipo;
```

4.2 The answer

SQL | All Rows Fetched: 7 in 0.034 seconds

	NR	NOME	TOTAL_HORAS_SEMANAIS	TIPO
1	210006	João Carlos Pascoal de Faria	3.5	OT
2	208848	Palmira Dias Oliveira Ferreira	18	P
3	211768	Joaquim Luís Bernardes Martins de Faria	18	P
4	232673	Manuel António Moreira Alves	18	P
5	230268	Adélio Miguel Magalhães Mendes	18	P
6	211342	Fernanda Maria Campos de Sousa	8	T
7	371086	Bárbara Rangel Carvalho	21	TP

Figure 19: Query 4 result

4.3 Execution Plans

4.3.1 X Environment

The query execution plan in figure 20 is mainly composed of full table accesses with access and filter predicates, as well as hash joins.

To begin with, the DBMS does a full table scan of the `XTIPOSAULA` table filtering by `ano_letivo` that must match `2003/2004`. It then `HASH JOIN`s the resulting tuples with the `XDSD` table, filtered by `NOT NULL` values of `fator`, based on the `id` column.

This operation is then repeated for the inner query and these two tables are joined based on the second's maximum value and their `tipo` column.

Finally, the result of this table is again joined with the `XDOCENTES` table based on the `nr` column, in order to fetch the professors' names. If we didn't need the professors' names then this query could be simplified by removing the join operation with the `XDOCENTES` table, reducing the cost by 5.

4.3.2 Y Environment

As we can see in figure 21, the execution plan is exactly the same. We can conclude that the query has a weak selectivity, hence the fact that the optimizer prefers to perform full table scans instead of using indexes. This is to be expected because the nature of this question is to find out which professors have the largest amount

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			46	134
SORT		ORDER BY	46	134
HASH JOIN			46	133
Access Predicates				
DOCENTES.NR=DSD.NR				
HASH JOIN			46	128
Access Predicates				
AND				
MAX(SUB_DSD.FATOR*SUB_DSD.HORAS)=DSD.FATOR*DSD.HORAS				
ITEM_1=TIPO.TIPO				
VIEW	SYS.VW_SQ_1		5	64
HASH		GROUP BY	5	64
HASH JOIN			2567	63
Access Predicates				
SUB_TIPO.ID=SUB_DSD.ID				
TABLE ACCESS	XTIPOSAULA	FULL	1588	36
Filter Predicates				
SUB_TIPO.ANO_LETIVO='2003/2004'				
TABLE ACCESS	XDSD	FULL	27356	27
Filter Predicates				
SUB_DSD.FATOR IS NOT NULL				
HASH JOIN			2567	63
Access Predicates				
TIPO.ID=DSD.ID				
TABLE ACCESS	XTIPOSAULA	FULL	1588	36
Filter Predicates				
TIPO.ANO_LETIVO='2003/2004'				
TABLE ACCESS	XDSD	FULL	27356	27
Filter Predicates				
DSD.FATOR IS NOT NULL				
TABLE ACCESS	XDOCENTES	FULL	939	5

Figure 20: Query 4 execution plan in X environment

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			46	134
SORT		ORDER BY	46	134
HASH JOIN			46	133
Access Predicates				
DOCENTES.NR=DSD.NR				
HASH JOIN			46	128
Access Predicates				
AND				
MAX(SUB_DSD.FATOR*SUB_DSD.HORAS)=DSD.FATOR*DSD.HORAS				
ITEM_1=TIPO.TIPO				
VIEW	SYS.VW_SQ_1		5	64
HASH		GROUP BY	5	64
HASH JOIN			2567	63
Access Predicates				
SUB_TIPO.ID=SUB_DSD.ID				
TABLE ACCESS	YTIPOSAULA	FULL	1588	36
Filter Predicates				
SUB_TIPO.ANO_LETIVO='2003/2004'				
TABLE ACCESS	YDSD	FULL	27356	27
Filter Predicates				
SUB_DSD.FATOR IS NOT NULL				
HASH JOIN			2567	63
Access Predicates				
TIPO.ID=DSD.ID				
TABLE ACCESS	YTIPOSAULA	FULL	1588	36
Filter Predicates				
TIPO.ANO_LETIVO='2003/2004'				
TABLE ACCESS	YDSD	FULL	27356	27
Filter Predicates				
DSD.FATOR IS NOT NULL				
TABLE ACCESS	YDOCENTES	FULL	939	5

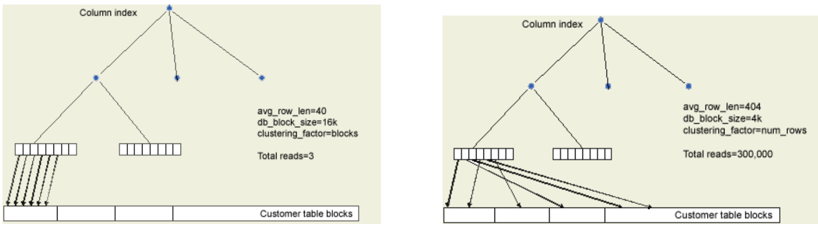
Figure 21: Query 4 execution plan in Y environment

of weekly class hours. So even thought there are joins on indexed columns it is better to simply perform full scans and filter tables then to access them via row ids.

But why is it better? To understand this, we need to look at how many blocks from the table will the query access using the indexes and how does that compare from the total number of blocks from the table. Provided the index enables us to read fewer blocks from the table than there are in total, then it is the right way to go. Soon as that index means that we'll have to re-read many blocks, than a full table scan is the right method. We can use the clustering factor for an indication of how likely the index is to be effective, with higher values meaning that it is going to be less effective. However, by default, this is a pessimistic estimate.

What is the clustering factor? It is an estimate that the database calculates when gathering statistics from a table. For each index, it walks down the entries and, for each entry, it checks if that entry is in the same block as the previous one. If not, then it increments the counter. If it is the same, then it just leaves it. So the minimum value for this factor is the number of blocks in a table, and the maximum value is the number of rows in that same table.

In conclusion, the higher the clustering factor, then the more scattered throughout the table the rows are relative to the order in the index.



(a) Clustered Index (b) Un-clustered index

Figure 22: Clustered and un-clustered data

As we can see from the images in 22, the un-clustered index requires more reads than there are blocks in the table. In this situation, it is better to perform full table scans.

```sql
SELECT ui.index_name,
       ui.clustering_factor,
       ut.num_rows,
       ut.blocks
FROM   user_indexes ui
       JOIN user_tables ut
         ON ui.table_name = ut.table_name
WHERE  ui.table_name = 'YTIPOSAULA';
-- > INDEX_NAME   , CLUSTERING_FACTOR, NUM_ROWS, BLOCKS
-- > YTIPOSAULA_PK  3879              21019     126
```

As we can see the clustering factor is quite high in relation to the number of blocks. After some research, we've come the conclusion that the Oracle DBMS will probably do a full-table scan operation on the table rather than use an index if a significant percentage of the table is going to be accessed. This percentage can be calculated by dividing the clustering factor by the number of rows, in this case:

$$\frac{3879}{21019} = 18.45\%.$$

How does this clustering factor compare with another table, for example `YDOCENTES`?

```sql
SELECT ui.index_name,
       ui.clustering_factor,
       ut.num_rows,
       ut.blocks
FROM   user_indexes ui
       JOIN user_tables ut
         ON ui.table_name = ut.table_name
WHERE  ui.table_name = 'YDOCENTES';
-- > INDEX_NAME   , CLUSTERING_FACTOR, NUM_ROWS, BLOCKS
-- > YDOCENTES_PK  12                939       13
```

$$\frac{12}{939} = 1.28\%.$$

This clustering factor is rather low and quite close the the number of blocks. This means that the data is almost perfectly clustered and the optimizer will most certainly make use of this index.

Let's repeat this analysis with the final `YDSD` table.

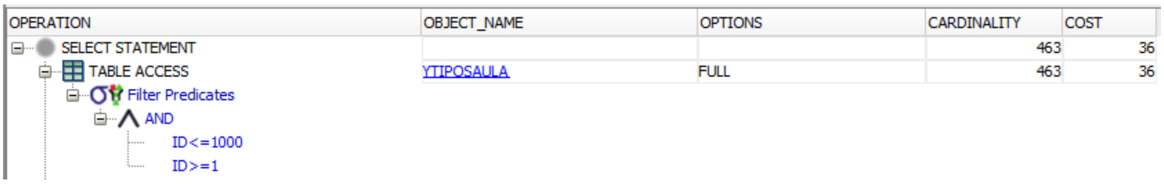
```sql
SELECT ui.index_name,
       ui.clustering_factor,
       ut.num_rows,
       ut.blocks
FROM   user_indexes ui
       JOIN user_tables ut
         ON ui.table_name = ut.table_name
WHERE  ui.table_name = 'YDSD';
-- > INDEX_NAME, CLUSTERING_FACTOR, NUM_ROWS, BLOCKS
-- > YDSD_PK     19715             27385     96
```

$$\frac{19715}{27385} = 71.99\%.$$

As we can see this is by far the worst clustering factor we've analyzed.

So in order to test this theory, we are going to query the `YTIPOSAULA` table using only indexed columns and see what the optimizer is telling us.

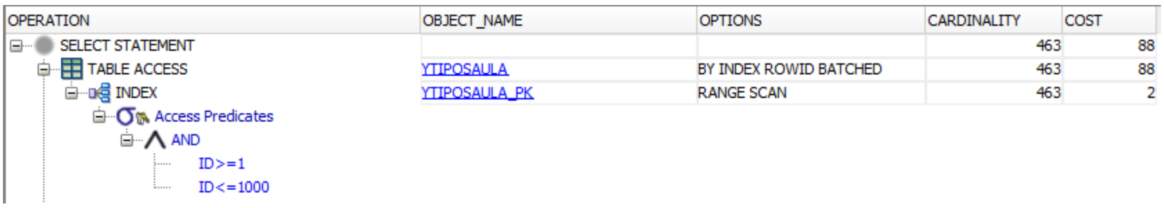
```sql
-- let the opmitizer choose on range scan
SELECT * FROM ytiposaula
WHERE id BETWEEN 1 AND 1000;
```



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			463	36
TABLE ACCESS	YTIPOSAULA	FULL	463	36
Filter Predicates				
AND				
ID<=1000				
ID>=1				

Figure 23: Range query test

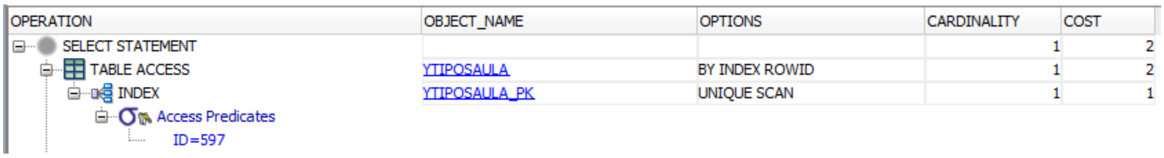
```sql
-- force index on range scan
SELECT /*+ index (ytiposaula (id) ) */* FROM ytiposaula
WHERE id BETWEEN 1 AND 1000;
```



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			463	88
TABLE ACCESS	YTIPOSAULA	BY INDEX ROWID BATCHED	463	88
INDEX	YTIPOSAULA_PK	RANGE SCAN	463	2
Access Predicates				
AND				
ID>=1				
ID<=1000				

Figure 24: Range query test force index use

```sql
-- simple index access
SELECT * FROM ytiposaula
WHERE id = 597;
```



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	2
TABLE ACCESS	YTIPOSAULA	BY INDEX ROWID	1	2
INDEX	YTIPOSAULA_PK	UNIQUE SCAN	1	1
Access Predicates				
ID=597				

Figure 25: Equality query test

As we can see, if we use `hints`, like in figure 24, to force the index on a range scan the performance is much worse. So it is normal that the Oracle DBMS prefers a full table scan like figure 23 suggests.

It is important to note that the order of the composite index in `YDSD` matters to this query, because Oracle reads an index starting with its leftmost column - `YDSD_PK: (NR, ID)`. Since we are trying to access the `YDOCENTE`'s `NR` through the `YDSD`'s `ID` then it might be a good idea to swap the column order on this index.

```sql
-- remove primary key
ALTER TABLE YDSD
DROP CONSTRAINT YDSD_PK;
-- create new primary key with new column order
ALTER TABLE YDSD
ADD CONSTRAINT YDSD_PK PRIMARY KEY (ID, NR);
```

According to figure 26, now we can see that the new `YDSD_PK` index would be used, however the *Adaptive Query Optimization* feature notices that this index is not very useful. Hence if we remove the grey leafs, than the execution plan is exactly the same. First `YTIPOSAULA` is fully read and filtered, and then joined via a `HASH JOIN` with the `YDSD` table. So in the end, there's not a single change in the actual execution plan.

SELECT STATEMENT			46	134
SORT		ORDER BY	46	134
HASH JOIN			46	133
Access Predicates				
DOCENTES.NR=DSD.NR				
HASH JOIN			46	128
Access Predicates				
AND				
MAX(SUB_DSD.FATOR*SUB_DSD.HORAS)=DSD.FATOR*DSD.HORAS				
ITEM_1=TIPO.TIPO				
VIEW	SYS.VW_SQ_1		5	64
HASH		GROUP BY	5	64
HASH JOIN			2567	63
Access Predicates				
SUB_TIPO.ID=SUB_DSD.ID				
NESTED LOOPS			2567	63
NESTED LOOPS				
STATISTICS COL				
TABLE ACCE	YTIPOSAULA	FULL	1588	36
Filter Predicates				
SUB_TIPO.ANO_LETIVO='2003/2004'				
INDEX	YDSD_PK	RANGE SCAN		
Access Predicates				
SUB_TIPO.ID=SUB_DSD.ID				
TABLE ACCESS	YDSD	BY INDEX ROWID	2	27
Filter Predicates				
SUB_DSD.FATOR IS NOT NULL				
TABLE ACCESS	YDSD	FULL	27356	27
Filter Predicates				
SUB_DSD.FATOR IS NOT NULL				
HASH JOIN			2567	63
Access Predicates				
TIPO.ID=DSD.ID				
TABLE ACCESS	YTIPOSAULA	FULL	1588	36
Filter Predicates				
TIPO.ANO_LETIVO='2003/2004'				
TABLE ACCESS	YDSD	FULL	27356	27
Filter Predicates				
DSD.FATOR IS NOT NULL				
TABLE ACCESS	YDOCENTES	FULL	939	5

Figure 26: Query 4 explanation plan with composite index column swap

4.3.3 Z Environment

```sql
DROP INDEX ZTIPOSAULA_IDX_ANO_LETIVO;
CREATE INDEX ZTIPOSAULA_IDX_ANO_LETIVO ON ZTIPOSAULA(ano_letivo);
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			46	124
SORT		ORDER BY	46	124
HASH JOIN			46	123
Access Predicates				
DOCENTES.NR=DSD.NR				
HASH JOIN			46	118
Access Predicates				
AND				
MAX(SUB_DSD.FATOR*SUB_DSD.HORAS)=DSD.FATOR*DSD.HORAS				
ITEM_1=TIPO.TIPO				
VIEW	SYS.VW_SQ_1		5	59
HASH		GROUP BY	5	59
HASH JOIN			2567	58
Access Predicates				
SUB_TIPO.ID=SUB_DSD.ID				
TABLE ACCESS	ZTIPOSAULA	BY INDEX ROWID BATCHED	1588	31
INDEX	ZTIPOSAULA_IDX_ANO_LETIVO	RANGE SCAN	1588	5
Access Predicates				
SUB_TIPO.ANO_LETIVO='2003/2004'				
TABLE ACCESS	ZDSD	FULL	27356	27
Filter Predicates				
SUB_DSD.FATOR IS NOT NULL				
HASH JOIN			2567	58
Access Predicates				
TIPO.ID=DSD.ID				
TABLE ACCESS	ZTIPOSAULA	BY INDEX ROWID BATCHED	1588	31
INDEX	ZTIPOSAULA_IDX_ANO_LETIVO	RANGE SCAN	1588	5
Access Predicates				
TIPO.ANO_LETIVO='2003/2004'				
TABLE ACCESS	ZDSD	FULL	27356	27
Filter Predicates				
DSD.FATOR IS NOT NULL				
TABLE ACCESS	ZDOCENTES	FULL	939	5

Figure 27: Query 4 execution plan in Z environment

After quite a few experiments, including various index types, i.e. B-tree and Bitmap, we've come to the conclusion that optimizing this query with indexes is not straight forward. The only small improvement we could achieve was by simply adding an index to the `ano_letivo` column that is being filtered in the `WHERE` clause, as figure 27 suggests.

4.4 Execution Time

X Environment	Y Environment	Z Environment
0.036	0.037	0.036

Table 5: Query 4 average execution times in all environments (in seconds)

5 Question 5.

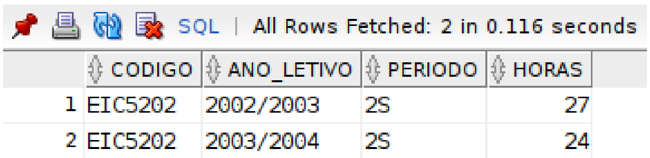
"Compare the execution plans (just the environment Z) and the index sizes for the query giving the course code, the academic year, the period, and number of hours of the type 'OT' in the academic years of 2002/2003 and 2003/2004."

1. With a B-tree index on the type and academic year columns of the `ZTIPOSAULA` table;
2. With a bitmap index on the type and academic year columns of the `ZTIPOSAULA` table;

5.1 The SQL Query

```
SELECT aulas.codigo,
       aulas.ano_letivo,
       aulas.periodo,
       ( aulas.horas_turno * aulas.turnos ) horas
FROM   ztiposaula aulas
WHERE  aulas.tipo = 'OT'
       AND ( aulas.ano_letivo = '2002/2003'
             OR aulas.ano_letivo = '2003/2004' );
```

5.2 The answer



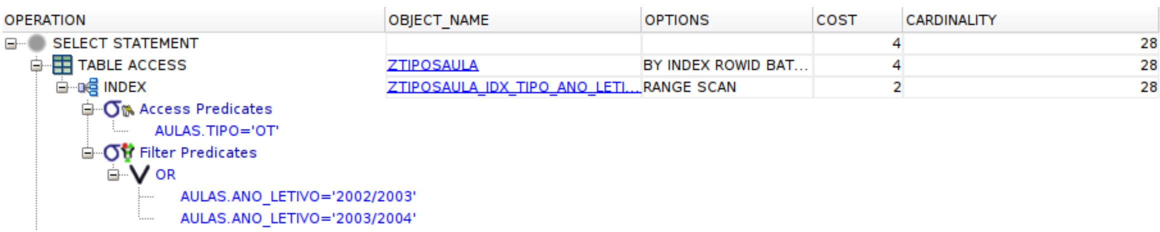
	CODIGO	ANO_LETIVO	PERIODO	HORAS
1	EIC5202	2002/2003	2S	27
2	EIC5202	2003/2004	2S	24

Figure 28: Query 5 result

5.3 Execution Plans

5.3.1 B-Tree Index

```
DROP INDEX ZTIPOSAULA_IDX_TIPO_ANO_LETIVO;
CREATE INDEX ZTIPOSAULA_IDX_TIPO_ANO_LETIVO ON ZTIPOSAULA(tipo, ano_letivo);
```

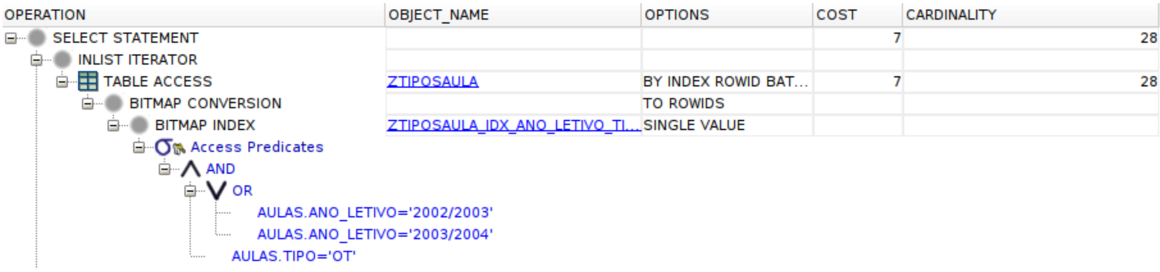


OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			4	28
TABLE ACCESS	ZTIPOSAULA	BY INDEX ROWID BAT...	4	28
INDEX	ZTIPOSAULA_IDX_TIPO_ANO_LETI...	RANGE SCAN	2	28
Access Predicates				
AULAS.TIPO='OT'				
Filter Predicates				
OR				
AULAS.ANO_LETIVO='2002/2003'				
AULAS.ANO_LETIVO='2003/2004'				

Figure 29: Query 5 execution plan using a B-Tree index

With a B-Tree index on `TIPO` and `ANO_LETIVO`, the query can be done by filtering the index on the where clauses directly, using an `INDEX RANGE SCAN`, and then access the table `ZTIPOSAULA` by the `ROWIDS` that matched, since the `SELECT` statement requires columns that are not contained in the index.

The columns' order on the index is important, since in this case the optimizer starts by retrieving the values on the index where the `TIPO` matches the `WHERE` clause and then just filters the resulting set by the ones that have an `ANO_LETIVO` in accordance with the `OR` operation. If the order on the index was inverted, it would need to first find all the occurrences where `ANO_LETIVO` matches while also checking that the `TIPO` was the correct one, by going "up" the index tree, which, although small, would raise the estimated cost to 5.



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			7	28
INLIST ITERATOR				
TABLE ACCESS	ZTIPOSAULA	BY INDEX ROWID BAT...	7	28
BITMAP CONVERSION		TO ROWIDS		
BITMAP INDEX	ZTIPOSAULA_IDX_ANO_LETIVO_TI...	SINGLE VALUE		
Access Predicates				
AND				
OR				
AULAS.ANO_LETIVO='2002/2003'				
AULAS.ANO_LETIVO='2003/2004'				
AULAS.TIPO='OT'				

Figure 30: Query 5 execution plan using a Bitmap index

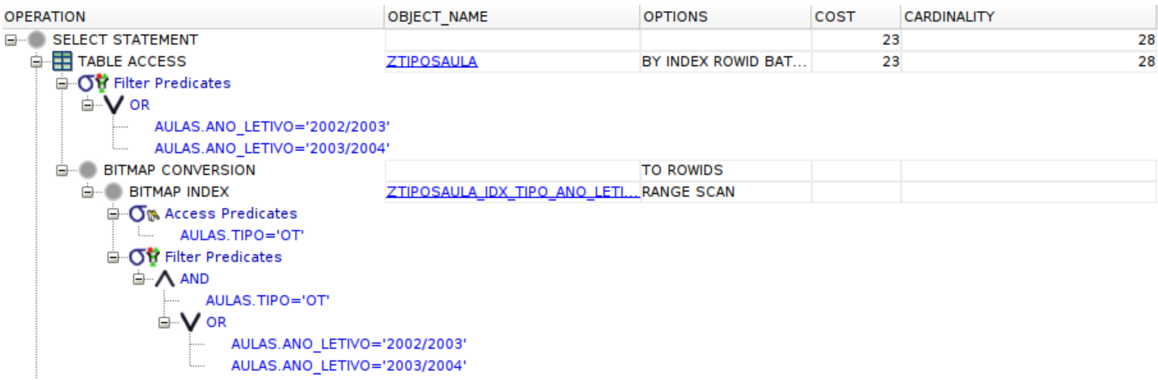
5.3.2 Bitmap Index

```sql
DROP INDEX ZTIPOSAULA_IDX_ANO_LETIVO_TIPO;
CREATE BITMAP INDEX ZTIPOSAULA_IDX_ANO_LETIVO_TIPO ON ZTIPOSAULA(ano_letivo, tipo);
```

From the execution plan of figure 30, we can see that with a Bitmap Index it performs a lookup where the bits match the values needed for the `WHERE` clause and then accesses the table using the `ROWIDs` returned after the necessary conversion.

Compared to the B-Tree index, the estimated cost is almost double. Usually, Bitmap indexes are more efficient than B-Tree indexes, especially as the number of rows or the number of `AND`/`OR` operations increases. We believe that, since the table `ZTIPOSAULA` isn't too large, and is densely distributed, occupying only 126 blocks, together with the fact that the `WHERE` constraints aren't many, the base query, without any indexes, ends up having an estimated cost of only 36, and as such, using indexes isn't going to have a major cost improvement, both ending up having very low costs, especially when they are optimized for the query at hand. With this, the difference one would expect when using a Bitmap index instead of a B-Tree index isn't significant in our case, with the B-Tree ending up being better.

Interestingly, order seems to be important for the Bitmap index as well, since swapping `ANO_LETIVO` and `TIPO` makes it so the plan has an estimated cost of 23, as it can be seen in figure 31.



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			23	28
TABLE ACCESS	ZTIPOSAULA	BY INDEX ROWID BAT...	23	28
Filter Predicates				
OR				
AULAS.ANO_LETIVO='2002/2003'				
AULAS.ANO_LETIVO='2003/2004'				
BITMAP CONVERSION		TO ROWIDS		
BITMAP INDEX	ZTIPOSAULA_IDX_TIPO_ANO_LETI...	RANGE SCAN		
Access Predicates				
AULAS.TIPO='OT'				
Filter Predicates				
AND				
AULAS.TIPO='OT'				
OR				
AULAS.ANO_LETIVO='2002/2003'				
AULAS.ANO_LETIVO='2003/2004'				

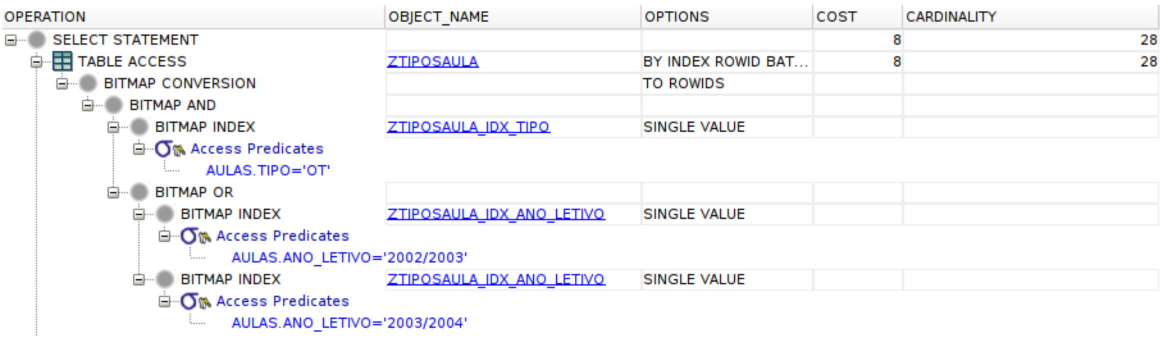
Figure 31: Query 5 execution plan using a Bitmap index with columns inverted

We couldn't find much information on why this occurs, but it seems that the query optimizer can't take full advantage of the Bitmap index, only being able to use it to access the rows that match `TIPO`, having to then filter the ones that have the desired `ANO_LETIVO` when accessing the table, which obviously makes the plan more costly than directly having the `ROWIDs` that obey the conditions.

We believe this might have to do with the way it generates the code to match the bitmaps in the index. It may be that it's only able to detect that it can use a code to match both the `TIPO` and `ANO_LETIVO` when the order on the query matches the one on the index, otherwise it only uses the index for the first column.

It's important to note that usually, bitmap indexes are created separately, precisely so they can be combined to work efficiently in various queries using bitwise operations. The reason why in this case we combined both columns into a composite bitmap index was because, in the correct order, it has an estimated cost better than when separated, but only by one. We can see the execution plan when using two separate bitmap indexes, one on `ANO_LETIVO` and another on `TIPO`, in figure 32.

Nevertheless, although for the analysis we will use the composite bitmap index, in a real world application, a composite index shouldn't be chosen in this case, since the potential savings aren't significant to make it preferable to the flexibility of having two separate indexes and therefore being able to be reused in other queries efficiently.



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			8	28
TABLE ACCESS	ZTIPOSAULA	BY INDEX ROWID BAT...	8	28
BITMAP CONVERSION		TO ROWIDS		
BITMAP AND				
BITMAP INDEX	ZTIPOSAULA_IDX_TIPO	SINGLE VALUE		
Access Predicates				
AULAS.TIPO='OT'				
BITMAP OR				
BITMAP INDEX	ZTIPOSAULA_IDX_ANO_LETIVO	SINGLE VALUE		
Access Predicates				
AULAS.ANO_LETIVO='2002/2003'				
BITMAP INDEX	ZTIPOSAULA_IDX_ANO_LETIVO	SINGLE VALUE		
Access Predicates				
AULAS.ANO_LETIVO='2003/2004'				

Figure 32: Query 5 execution plan using two separate Bitmap indexes

5.4 Indexes Sizes

The indexes sizes were obtained by running the following query for each one.

```sql
SELECT SUM(bytes) / 1024
FROM   user_segments
WHERE  segment_name = <INDEX_NAME>;
```

B-Tree Index	Bitmap Index
640KB	64KB

Table 6: Size of the different indexes used for query 5

When analyzing the indexes size, we can spot a major difference, with the Bitmap index occupying 10 times less memory than its B-Tree counterpart.

Unlike B-Tree indexes, where one index entry points to a single row, in a Bitmap index an entry can point to multiple rows. They are also far easier to compress, due to being represented as a binary set of 1s and 0s.

These two facts explain why the Bitmap index is substantially smaller than the B-Tree index. If instead of a single composite Bitmap index, two separate Bitmap indexes were used, the total size would just double (128KB), still being considerably less than the B-Tree index.

Although it's much smaller and still performs in the same range as the B-Tree index, the Bitmap index could still not be adequate to use in this case, if the database was updated regularly, as operations on the table require a great deal of work in order to update a Bitmap index, in many cases being faster to redo the index than to update it. Even worse, is the fact that when updating, no other operation on the table can occur before the previous one finishes. Since Bitmap entries can point to multiple rows, two updates can't occur at the same time. This isn't the case with B-Tree indexes for the majority of situations, and as such, they are better suited for cases where there is a great deal of concurrency in updating the database.

5.5 Execution Time

B-Tree Index	Bitmap Index
0.042	0.048

Table 7: Query 5 average execution times for both types of indexes

6 Question 6.

"Select the programs (curso) that have classes with all the existing types."

In the spirit of experimentation, we developed two different queries, both yielding the same result. Since both have a very distinct formulation and complexity, we chose to analyze both separately. Lets start with option A.

6.1 Option A

Our first approach is rather simple: count the number of distinct types of classes and then select the programs whose number of types of classes matches the amount counted before. This query allowed us to get the answer very quickly and with a good degree of confidence.

6.1.1 The SQL Query

```sql
SELECT ct.curso
FROM   (SELECT DISTINCT ucs.curso,
                        tipo.tipo
        FROM   xucs ucs
               JOIN xtiposaula tipo
                 ON tipo.codigo = ucs.codigo) ct
HAVING Count(ct.tipo) = (SELECT Count(DISTINCT tipo)
                         FROM   xtiposaula)
GROUP  BY ct.curso;
```

6.1.2 The answer

SQL | All Rows Fetched: 4 in 1,516 seconds

	CURSO
1	9461
2	4495
3	9508
4	2021

Figure 33: Query 6 - Option A result

6.1.3 Execution Plans

6.1.3.1 X Environment

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			51	2
FILTER				
Filter Predicates				
COUNT(CT.TIPO)= (SELECT COUNT($vm_col_1) FROM (SELECT TIPO $vm_col_1 FROM XTIPOSAULA XTIPOSAULA GROUP BY TIPO) VM_NWVW_1)				
HASH		GROUP BY	51	2
VIEW			51	404
HASH		UNIQUE	51	404
HASH JOIN			49	21019
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
TABLE ACCESS	XUCS	FULL	13	5396
TABLE ACCESS	XTIPOSAULA	FULL	36	21019
SORT		AGGREGATE		1
VIEW	SYS.VM_NWVW_1		37	5
SORT		GROUP BY	37	5
TABLE ACCESS	XTIPOSAULA	FULL	36	21019

Figure 34: Query 6 - Option A execution plan in X environment

In this query the usual "filter then join" method does not work. The filter applied in the `HAVING` clause has to be applied to the whole table returned by the nested `SELECT`. That is where the optimizer starts: it performs a

Hash Join with two Full Table Scans to join XUCS with XTIPOSAULA. There is no other way around it, the tables are too large (XTIPOSAULA has 21019 rows and XUCS has 5396 rows), so sorting them for a Sort-Merge Join would cost too much, and completely loading one of them to memory for a Nested Loops does not seem feasible. After the Hash Join, the values are filtered by the DISTINCT keyword, which in practice means a hash function is used to place each unique value in a different bucket, discarding hash collisions, thus only saving each value once. Finally, the resulting values are grouped by CURSO so that the COUNT operation is performed only once per CURSO.

Afterwards, the expression inside the HAVING clause has to be computed. The optimizer performs some manipulation on this expression that can be read on the "Filter Predicates" line of figure 34. The DISTINCT keyword is replaced by a GROUP BY and an alias VM_NWVW_1 is used to refer to the selection made on XTIPOSAULA table. This selection is planned on the bottom of the execution plan. As we can see, the optimizer performs a Full Table Scan on XTIPOSAULA followed by a sort operation to separate the rows into different groups, each with the same value for the TIPO column. The resulting table is then aggregated by the COUNT operation and finally used to filter the XUCS and XTIPOSAULA join. After applying the filter, the optimizer ends up selecting the column CURSO for the resulting rows.

6.1.3.2 Y Environment

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			51	2
FILTER				
Filter Predicates				
COUNT(CT.TIPO)= (SELECT COUNT($vm_col_1) FROM (SELECT TIPO $vm_col_1 FROM YTIPOSAULA YTIPOSAULA GROUP BY TIPO) VM_NWVW_1)				
HASH		GROUP BY	51	2
VIEW			51	404
HASH		UNIQUE	51	404
HASH JOIN			49	21019
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
TABLE ACCESS	YUCS	FULL	13	5396
TABLE ACCESS	YTIPOSAULA	FULL	36	21019
SORT		AGGREGATE		1
VIEW	SYS.VM_NWVW_1		37	5
SORT		GROUP BY	37	5
TABLE ACCESS	YTIPOSAULA	FULL	36	21019

Figure 35: Query 6 - Option A execution plan in Y environment

This plan is exactly the same as the one for environment X. Although we now have an index on YUCS.CODIGO, it is not enough to optimize this query. The reason is that we need both YUCS.CURSO and YUCS.CODIGO for this query, so even if the optimizer used the index to make the join faster, it would need to afterwards join with the full YUCS table to get the CURSO column. Another option would be to access YUCS by Rowid after querying the index. However, since we need all rows from both tables, accessing the index would be a waste of time when it can directly scan the table and get all rows with all information.

That said we could clearly benefit from an index on both CURSO and CODIGO (by this order), thus completely eliminating the need to access the YUCS table. Apart from YUCS.CODIGO there is no other index available for this query, so there is nothing that could be further optimized in this execution plan.

6.1.3.3 Z Environment

```
DROP INDEX ZUCS_IDX_CURSO_CODIGO;
CREATE UNIQUE INDEX ZUCS_IDX_CURSO_CODIGO ON ZUCS(curso, codigo);
```

As suggested in the analysis for environment Y, an index on both ZUCS.CURSO and ZUCS.CODIGO improves the query as we can see in figure 36. Since those are the only columns we need in this query from ZUCS, the need to get rows from the original table does no longer exist. Therefore, the Hash Join is now performed on the index through a Fast Full Scan instead of a Full Table Scan on ZUCS, droping the cost from 13 to 6. This is already a good optimization and allows us to reuse an index that already existed from previous questions. However, there are only two more columns being used in the query - ZTIPOSAULA.CODIGO and ZTIPOSAULA.TIPO. By creating an index on both of these columns, in the same order, we can further optimize the query:

```
DROP INDEX ZTIPOSAULA_IDX_CODIGO_TIPO;
CREATE INDEX ZTIPOSAULA_IDX_CODIGO_TIPO ON ZTIPOSAULA(codigo, tipo);
```

This index has a bigger impact than the previous one because the table ZTIPOSAULA was being used in two distinct parts of the execution plan (see figure 37). First, the Hash Join no longer needs to access the table ZTIPOSAULA because a Fast Full Scan on the index is enough to join through the CODIGO column and to fetch the

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			44	2
FILTER				
Filter Predicates				
COUNT(CT.TIPO)= (SELECT COUNT($vm_col_1) FROM (SELECT TIPO $vm_col_1 FROM ZTIPOSAULA ZTIPOSAULA GROUP BY TIPO) VM_NWVW_1)				
HASH		GROUP BY	44	2
VIEW			44	404
HASH		UNIQUE	44	404
HASH JOIN			42	21019
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
INDEX	ZUCS_IDX_CURSO_C...	FAST FULL SCAN	6	5396
TABLE ACCESS	ZTIPOSAULA	FULL	36	21019
SORT		AGGREGATE		1
VIEW	SYS.VM_NWVW_1		37	5
SORT		GROUP BY	37	5
TABLE ACCESS	ZTIPOSAULA	FULL	36	21019

Figure 36: Query 6 - Option A execution plan in a not fully optimized Z environment

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			26	2
FILTER				
Filter Predicates				
COUNT(CT.TIPO)= (SELECT COUNT($vm_col_1) FROM (SELECT TIPO $vm_col_1 FROM ZTIPOSAULA ZTIPOSAULA GROUP BY TIPO) VM_NWVW_1)				
HASH		GROUP BY	26	2
VIEW			26	404
HASH		UNIQUE	26	404
HASH JOIN			24	21019
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
INDEX	ZUCS_IDX_CURSO_C...	FAST FULL SCAN	6	5396
INDEX	ZTIPOSAULA_IDX_CO...	FAST FULL SCAN	18	21019
SORT		AGGREGATE		1
VIEW	SYS.VM_NWVW_1		19	5
SORT		GROUP BY	19	5
INDEX	ZTIPOSAULA_IDX_CO...	FAST FULL SCAN	18	21019

Figure 37: Query 6 - Option A execution plan in a fully optimized Z environment

`TIPO` column. This operation's estimated cost of 18 is half the cost of a Full Table Scan on `ZTIPOSAULA`, which is quite a big improvement. In addition, the selection of distinct `TIPO`'s is also improved since it can perform the whole query on the index.

Overall, by using both proposed indexes (`ZUCS_IDX_CURSO_CODIGO` and `ZTIPOSAULA_IDX_CODIGO_TIPO`) we were able to cut the cost of the query in half (from 51 to 26). This would be a very important optimization if this query was very frequent on a running system.

6.2 Option B

Our second approach is way more complex. It is heavily based on set theory. We start by creating a set with all possible pairs of program and type of class. Afterwards, we subtract this set by the set of pairs which are actually stored on the database. The resulting set is made of all pairs that should exist in order for all programs to have classes of all types. This means that the programs which already have all types of classes are removed by subtracting the two sets. Based on this, we subtract the set of all programs on the database by the set of programs that were part of the subtraction made before and we end up with the answer.

Translating this to SQL took a bit of effort, but the result is a much more demanding query. Therefore, this option takes approximately 6 minutes to run, being a lot slower than option A, which runs in less than a second.

6.2.1 The SQL Query

```sql
SELECT DISTINCT uc.curso -- Select 4
FROM   xucs uc
WHERE  uc.curso NOT IN (SELECT DISTINCT ctp.curso -- Select 3
                        FROM   (SELECT DISTINCT curso, tipo -- Select 2
                                FROM    xucs,
                                        xtiposaula) ctp
                        WHERE  ( ctp.curso, ctp.tipo )
                                NOT IN (SELECT DISTINCT ucs.curso, tipo.tipo -- Select 1
                                        FROM   xucs ucs
                                               JOIN xtiposaula tipo
```

```
                                  ON tipo.codigo = ucs.codigo));
```

6.2.2 The answer

SQL | All Rows Fetched: 4 in 331,938 seconds

	CURSO
1	2021
2	4495
3	9461
4	9508

Figure 38: Query 6 - Option B result

6.2.3 Execution Plans

6.2.3.1 X Environment

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			573829	114
HASH		UNIQUE	573829	114
HASH JOIN		ANTI NA	573827	5380
Access Predicates				
UC.CURSO=CURSO				
TABLE ACCESS	XUCS	FULL	13	5396
VIEW	SYS.VW_NSO_2		573811	1134185
MERGE JOIN		ANTI NA	573811	1134185
SORT		JOIN	573761	113418524
MERGE JOIN		CARTESIAN	184771	113418524
TABLE ACCESS	XUCS	FULL	13	5396
BUFFER		SORT	184758	21019
TABLE ACCESS	XTIPOSAULA	FULL	34	21019
SORT		UNIQUE	51	21019
Access Predicates				
AND				
INTERNAL_FUNCTION(CURSO)=INTERNAL_FUNCTION(CURSO)				
INTERNAL_FUNCTION(TIPO)=INTERNAL_FUNCTION(TIPO)				
Filter Predicates				
AND				
INTERNAL_FUNCTION(TIPO)=INTERNAL_FUNCTION(TIPO)				
INTERNAL_FUNCTION(CURSO)=INTERNAL_FUNCTION(CURSO)				
VIEW	SYS.VW_NSO_1		49	21019
HASH JOIN			49	21019
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
TABLE ACCESS	XUCS	FULL	13	5396
TABLE ACCESS	XTIPOSAULA	FULL	36	21019

Figure 39: Query 6 - Option B execution plan in X environment

Due to the complexity of this query, we have marked each `SELECT` to help on the explanation of the execution plan. The first one to be computed is Select 1, which is essentially the same as option's A nested `SELECT`. Both `XUCS` and `XTIPOSAULA` are completely joined through the column `CODIGO` using an Hash Join, mainly due to the size of both tables and to the existence of an enquality condition. Afterwards, they are grouped using a Unique Sort operation to eliminate duplicates and prepare the results for a Sort-Merge Join.

The second select to be computed is the number 2. This query will join all existing `CURSO`s and `TIPO`s by performing a Cartesian Product on tables `XUCS` and `XTIPOSAULA`. Since both tables have a big cardinality, a Nested Loops is impossible. The Hash Join could be a possibility, just like in Select 1, but this time there is no condition to match, which makes the Sort-Merge Join a better alternative for joining all columns from one table to the other. The optimizer uses a buffer to Sort all rows from `XTIPOSAULA`, which has a tremendous cost, but is a necessary evil. The overall cost of the Select 2 execution is quite big, being 184771 at the end of the Sort-Merge Join.

However, it is the cost of Select 3 that goes through the ceiling. This query has to filter the `CURSO`s that appear in Select 2 but not in Select 1. Due to the huge number of rows involved, the best option is an Anti Sort-Merge Join on the results of those two selects. This operation does exactly what we need: it merges the tables by excluding the rows that share values on both of them. The problem is that we have to sort an estimate of 113 million rows from Select 2. As expected, this operation's cost (573811) represents pretty much the cost of the whole query (573829) and is the clear bottleneck of performance.

To finish the operation and yield the expected results we still need to compute Select 4, i.e. to select the values of `XUCS.CURSO` that exist on the database but not on the result of Select 3. The `NOT IN` operation is actually interpreted as an Anti Join with an equality condition on `CURSO`, which leads the optimizer to choose an Anti Hash Join using the `CURSO` as the access predicate. Finally, the resulting values are hashed to remove duplicates and the query completes after *"just"* 6 minutes.

6.2.3.2 Y Environment

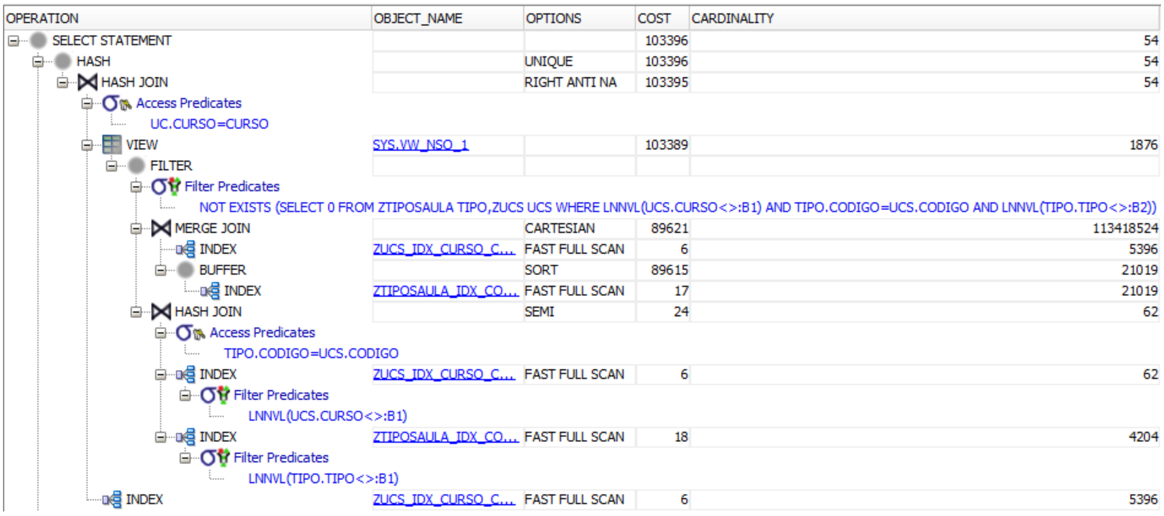
OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			573829	114
HASH		UNIQUE	573829	114
HASH JOIN		ANTI NA	573827	5380
Access Predicates				
UC.CURSO=CURSO				
TABLE ACCESS	YUCS	FULL	13	5396
VIEW	SYS.VW_NSO_2		573811	1134185
MERGE JOIN		ANTI NA	573811	1134185
SORT		JOIN	573761	113418524
MERGE JOIN		CARTESIAN	184771	113418524
TABLE ACCESS	YUCS	FULL	13	5396
BUFFER		SORT	184758	21019
TABLE ACCESS	YTIPOSAULA	FULL	34	21019
SORT		UNIQUE	51	21019
Access Predicates				
AND				
INTERNAL_FUNCTION(CURSO)=INTERNAL_FUNCTION(CURSO)				
INTERNAL_FUNCTION(TIPO)=INTERNAL_FUNCTION(TIPO)				
Filter Predicates				
AND				
INTERNAL_FUNCTION(TIPO)=INTERNAL_FUNCTION(TIPO)				
INTERNAL_FUNCTION(CURSO)=INTERNAL_FUNCTION(CURSO)				
VIEW	SYS.VW_NSO_1		49	21019
HASH JOIN			49	21019
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
TABLE ACCESS	YUCS	FULL	13	5396
TABLE ACCESS	YTIPOSAULA	FULL	36	21019

Figure 40: Query 6 - Option B execution plan in Y environment

This execution plan is exactly the same as the above and the reasons are the same as option A's. Therefore we will not dive into much detail here. There simply is no way to get the information needed for the query through the index available on `YUCS.CODIGO`, making accessing it a worse option. This time though, an index just on `YUCS.CURSO` would actually benefit Select 3 which would not need to perform a Full Table Access on `YUCS`. Since `CURSO` is not a primary key, the index does not exist and thus there is no optimization at all.

6.2.3.3 Z Environment

```sql
DROP INDEX ZUCS_IDX_CURSO_CODIGO;
CREATE UNIQUE INDEX ZUCS_IDX_CURSO_CODIGO ON ZUCS(curso, codigo);
DROP INDEX ZTIPOSAULA_IDX_CODIGO_TIPO;
CREATE INDEX ZTIPOSAULA_IDX_CODIGO_TIPO ON ZTIPOSAULA(codigo, tipo);
```



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			103396	54
HASH		UNIQUE	103396	54
HASH JOIN		RIGHT ANTI NA	103395	54
Access Predicates				
UC.CURSO=CURSO				
VIEW	SYS.VW_NSO_1		103389	1876
FILTER				
Filter Predicates				
NOT EXISTS (SELECT 0 FROM ZTIPOSAULA TIPO,ZUCS UCS WHERE LNNVL(UCS.CURSO<>:B1) AND TIPO.CODIGO=UCS.CODIGO AND LNNVL(TIPO.TIPO<>:B2))				
MERGE JOIN		CARTESIAN	89621	113418524
INDEX	ZUCS_IDX_CURSO_C...	FAST FULL SCAN	6	5396
BUFFER		SORT	89615	21019
INDEX	ZTIPOSAULA_IDX_CO...	FAST FULL SCAN	17	21019
HASH JOIN		SEMI	24	62
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
INDEX	ZUCS_IDX_CURSO_C...	FAST FULL SCAN	6	62
Filter Predicates				
LNNVL(UCS.CURSO<>:B1)				
INDEX	ZTIPOSAULA_IDX_CO...	FAST FULL SCAN	18	4204
Filter Predicates				
LNNVL(TIPO.TIPO<>:B1)				
INDEX	ZUCS_IDX_CURSO_C...	FAST FULL SCAN	6	5396

Figure 41: Query 6 - Option B execution plan in a not fully optimized Z environment

By using the same indexes from option A we achieve drastically different results. Not only the query runs in approximately 24 seconds, but also the cost of the query is 5 times lower! Note that even the estimated cardinality is much closer to the real result of 5 rows (this plan estimates 54 while the plans for the last two environments estimated 114).

The reason is clear: we are only accessing indexes, which are much smaller than corresponding tables. This makes a big difference especially for Select 2 that had to calculate a Cartesian Product between two tables. The Sort-Merge Join is carried out pretty much in the same way, but the indexes are much faster to sort due to their sizes.

Interestingly enough, the plan for Select 3 changes quite a bit. This happens due to a major refactor on the way it is computed. Instead of using a Sort-Merge Join to join the results of Selects 1 and 2, the optimizer converts the `NOT IN` operation on a `NOT EXISTS` for these two tables. The whole "Filter Predicate" of figure 41 is hard to understand, but it seems the optimizer is binding the `CURSO` and `TIPO` values in a variable and iterating over the tables to guarantee the deletion of duplicates for Select 3.

Something similar happens for Select 1. The optimizer prefers to access both indexes through Fast Full Scans and merge them through a Semi Hash Join. The indexes are filtered to remove duplicates even before the join operation. A Semi Join stops when the first match is encountered, which makes sense since both indexes do no longer contain duplicate values. This way, the optimizer can calculate the buckets for the index `ZTIPOSAULA_IDX_CODIGO_TIPO` (where there are many `TIPO`s for the same `CODIGO`) and then go through the index `ZUCS_IDX_CURSO_CODIGO` (where there is only one `CURSO` for each `CODIGO`) to find the first bucket with a match on `CODIGO` and stop there to save some time.

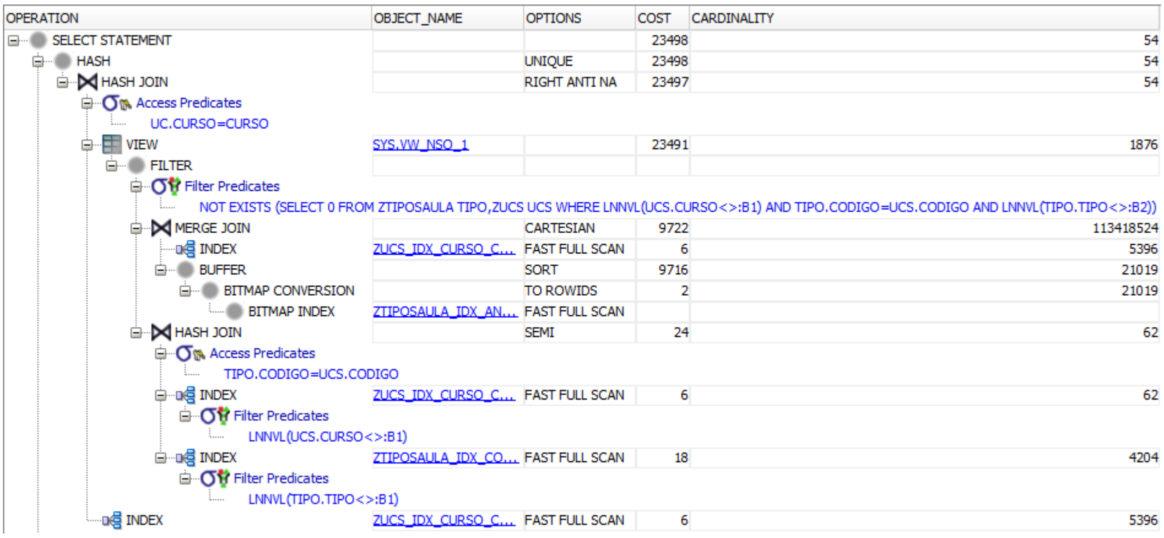
The Select 3 plan has already been explained above. After its results are calculated, the last step is to merge them with the index on `ZUCS` through an Anti Hash Join. The Anti variant will return the `CURSO`s that exist on the index but not on the result of Select 3. The Hash Join is obviously the best operation due to the equality condition used to match the two entities.

Finally, an Hash operation is used to remove duplicates from the result, thus giving us the expected `CURSO`s.

Although the result is already pretty good, after trying some more indexes we arrived at an even better solution:

```
DROP INDEX ZTIPOSAULA_IDX_TIPO;
CREATE BITMAP INDEX ZTIPOSAULA_IDX_TIPO ON ZTIPOSAULA(tipo);
```

By creating a Bitmap index on `ZTIPOSAULA.TIPO`, the Sort-Merge Join performed to compute Select 2 becomes almost 10 times cheaper (see figure 42). After converting all the rows of the index into Rowids, they are sorted in a much more efficient way. By using all three indexes in this environment we are able to reduce the cost of the query 24 times, from 573827 to 23498. What an amazing optimization we achieved with this query.



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			23498	54
HASH		UNIQUE	23498	54
HASH JOIN		RIGHT ANTI NA	23497	54
Access Predicates				
UC.CURSO=CURSO				
VIEW	SYS.VW_NSO_1		23491	1876
FILTER				
Filter Predicates				
NOT EXISTS (SELECT 0 FROM ZTIPOSAULA TIPO,ZUCS UCS WHERE LNNVL(UCS.CURSO<>:B1) AND TIPO.CODIGO=UCS.CODIGO AND LNNVL(TIPO.TIPO<>:B2))				
MERGE JOIN		CARTESIAN	9722	113418524
INDEX	ZUCS_IDX_CURSO_C...	FAST FULL SCAN	6	5396
BUFFER		SORT	9716	21019
BITMAP CONVERSION		TO ROWIDS	2	21019
BITMAP INDEX	ZTIPOSAULA_IDX_AN...	FAST FULL SCAN		
HASH JOIN		SEMI	24	62
Access Predicates				
TIPO.CODIGO=UCS.CODIGO				
INDEX	ZUCS_IDX_CURSO_C...	FAST FULL SCAN	6	62
Filter Predicates				
LNNVL(UCS.CURSO<>:B1)				
INDEX	ZTIPOSAULA_IDX_CO...	FAST FULL SCAN	18	4204
Filter Predicates				
LNNVL(TIPO.TIPO<>:B1)				
INDEX	ZUCS_IDX_CURSO_C...	FAST FULL SCAN	6	5396

Figure 42: Query 6 - Option B execution plan in a fully optimized Z environment

6.3 Execution Times

	X Environment	Y Environment	Z Environment
Option A	0.036	0.040	0.036
Option B	344	321	23

Table 8: Query 6 average execution times in all environments for both options (in seconds)

7 Conclusion

Over the course of this project we deepened our knowledge on indexes, the different types and their internal structure by exploring each query, the different environments, and how each of our decisions, whether it be on how we structured the query, the columns that we chose to index, the type of index or even the order of the indexes columns, affected the execution plan.

As such, one of the biggest takeaways was learning to interpret a query's execution plan and with it infer the best indexes to create in order to optimize it, and visualize exactly how they affect the plan. In order to fully comprehend some execution plans, we had to analyze how the characteristics of a table's data influences the execution plan, and subsequently, how integrity constraints and indexes can help work around its shortcomings.

In summary, this work was extremely useful for applying the concepts learned along the classes in a more practical way and force us to really think about our decisions and their repercussions.

References

[1] *4 Understanding Indexes and Clusters.* URL: https://docs.oracle.com/cd/A97630_01/server.920/a96533/data_acc.htm#8131.

[2] *Database Data Warehousing Guide.* July 2005. URL: https://docs.oracle.com/cd/B28359_01/server.111/b28313/indexes.htm.

[3] *Hash join semi - Ask TOM.* Nov. 2017. URL: https://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:561666200346114038.

[4] Dan Hotka. *Index Clustering Factor and Oracle.* Mar. 2017. URL: https://blog.toadworld.com/2017/03/20/index-clustering-factor-and-oracle.

[5] Jibba Jabba. *Execution Plan: what does "2 - access("DEPARTMENT*$_I$*D" =: B1)"mean?*. Mar. 2013. URL: https://community.oracle.com/tech/developers/discussion/2512264/execution-plan-what-does-2-access-department-id-b1-mean.

[6] Oracle. *Database Administrator's Guide.* Mar. 2008. URL: https://docs.oracle.com/cd/B28359_01/server.111/b28310/indexes003.htm.

[7] Chris Saxon. *How to Create and Use Indexes in Oracle Database.* Aug. 2017. URL: https://blogs.oracle.com/sql/how-to-create-and-use-indexes-in-oracle-database#why.

[8] thatjeffsmith. *Are the grey rows in the execution plan in SQL Developer are executed?* Mar. 2019. URL: https://stackoverflow.com/questions/55223136/are-the-grey-rows-in-the-execution-plan-in-sql-developer-are-executed.

[9] *Why Isn't My Query Using an Index? Databases for Developers: Performance #5.* June 2020. URL: https://www.youtube.com/watch?v=7sS9bqdxM3g.