Pragmatic Approach to Query Optimization

Subhi H. Hamdoon, PhD.
College of Applied Sciences,
Ministry of Higher Education,
Oman

Virendra Gawande, PhD.
College of Applied Sciences,
Ministry of Higher Education,
Oman

Ahmed Al-Barashdi
College of Applied Sciences,
Ministry of Higher Education,
Oman

ABSTRACT
One of the most critical functional requirements of a DBMS is its ability to process queries in a timely manner. This is particularly true for very large, applications such as weather forecasting, banking systems and aeronautical applications, which can contain millions and even trillions of records. The need for faster and faster, and immediate results never ceases. Thus, a great deal of research and resources is required on creating smarter, highly efficient query optimization techniques. Some of the basic techniques of query processing and optimization have been presented in this paper. This paper highlights the basic concepts of query processing and query optimization in the relational database domain. The results of the experiment presented have been verified using Query Analyzer.

General Terms
DBMS, Query, Optimization, High-level Language, SQL, Indexes, Stored Procedures, DB Schema, Joins.

Keywords
Optimization, Query Processing, Query Analyzer, Query Metrics, Tuples, Cardinality.

1. INTRODUCTION
Query processing and optimization is a fundamental part of any DBMS. To be utilized effectively, the results of queries must be available in the timeframe needed by the user – be it a person, robotic, assembly machine or even another distinct and separate DBMS.

A database query is the vehicle for instructing a DBMS to update or retrieve specific data to/from the physically stored medium. The actual updating and retrieval of data is performed through various “low-level” operations. Examples of such operations for a relational DBMS can be relational algebra operations like project, join, select, Cartesian product etc. While the DBMS is designed to process these low-level operations efficiently, it can be quite the burden to a user to submit requests to the DBMS in these formats. Consider the following request [7]:

“Display all the vehicle ids built in the year 2000.”

While this is easily understandable by a human, a DBMS must be presented with a format it can understand, such as this SQL statement:

Select vehicle_id From vehicles Where year = 2000

Note that this SQL statement will still need to be translated further by the DBMS so that the functions/methods within the DBMS program cannot only process the request, but do it in a timely manner.

2. QUERY PROCESSING
There are three phases that a query passes through, during the DBMS’ processing of that query [2] [7]:

1. Parsing and Translation
2. Optimization
3. Evaluation

2.1 Parsing and Translating the Query
Most queries submitted to a DBMS are in a high-level language such as SQL. During this stage the human readable form of the query is translated into forms usable by the DBMS.

High-level query languages such as SQL represent a query as a string, or sequence of characters. Some of these represent various types of tokens such as keywords, operators, operands, literal strings, etc. Like other languages, there are rules (syntax and grammar) that govern how the tokens can be combined into valid statements.

The primary job of the parser is to extract the tokens from the raw strings and translate them into the corresponding internal data elements and structures, and then finally to verify the validity and syntax of the original query string [7].

2.2 Optimizing the Query
In this stage, the query processor applies rules to the internal data structures of the query to transform these structures into equivalent, but more efficient representations. The rules can be based upon mathematical models of the relational algebra expression and tree (heuristics), upon cost estimates of different algorithms applied to operations or upon the semantics within the query and the relations it involves. Selecting the proper rules to apply, when to apply them and how they are applied is the function of the query optimization engine [7].

2.3 Evaluating the Query
The final step in processing a query is the evaluation phase. The best evaluation plan candidate generated by the optimization engine is selected and then executed. Note that there may exist multiple methods of executing a query. Besides processing a query in a simple sequential manner, some of a query’s individual operations can be processed in parallel – either as independent processes or as interdependent pipelines of processes or threads. Regardless of the method chosen, the actual results should be same [7].

3. QUERY METRICS
The execution time of a query depends on the resources needed to perform the needed operations: disk accesses, CPU cycles, RAM and, in the case of parallel and distributed systems, thread and process communication. Since the data transfer to/from disks is substantially slower than the memory

based transfers, the disk accesses usually represent an overwhelming majority of the total cost, particularly for very large databases that cannot be pre-loaded into memory. The cost to access a disk is usually measured in terms of the number of blocks transferred from and to disk, which will be the unit of measure used in this paper [7].

In order to estimate the various costs of query operations, the query optimizer utilizes a fairly extensive amount of metadata associated with the relations and their corresponding file structures. These data are collected during and after various database operations (such as queries) and stored in the DBMS catalog. These data include [9] [2]:

nr	Number of records (tuples) in a relation *r*. Knowing the number of records in a relation is a critical piece of data utilized in nearly all cost estimations of operations.
fr	Blocking factor (number of records per block) for relation *r*. This data is used in calculating the blocking factor, and is also useful in determining the proper size and number of memory buffers.
br	Number of blocks in relation *r*'s data-file. Also a critical and commonly used datum, *br* is calculated value equal to *nr/br*.
lr	Length of a record, in bytes, in relation *r*. The record size is another important data item used in many operations, particularly when the values differ significantly for two relations involved in an operation.
dAr	Number of distinct values of attribute *A* in relation *r*. This value is important in calculating the number of resulting records for a projection operation and for aggregate functions like sum, count and average.
x	Number of levels in a multi-level index (B+-tree, cluster index, etc.). This data item is used in estimating the number of block accesses needed in various search algorithms. Note that for a B+-tree, x will be equal to the height of the tree.
sA	Selection cardinality of an attribute. This is a calculated value equal to *nr* / *dAr*. When *A* is a key attribute, *sA* = 1. The selection cardinality allows the query optimizer to determine the *"average number of records that will satisfy an equality selection condition on that attribute"*.

The query optimizer also depends on other important data such as the ordering of the data file, the type of index structures available and the attributes involved in these file organization structures. Knowing whether certain access structures exist allows the query optimizer to select the appropriate algorithm(s) to use for particular operations.

4. OPTIMIZATION PRINCIPLES

4.1 Optimizing Queries

Queries are very fast. Generally, many records can be retrieved in less than a second, even with joins, sorting and calculations. As a rule of thumb, if a query is taking longer than a second, it needs to be optimized. Start with the Queries that are most often used and also that take the more time to execute [8].

4.2 Better DB Schema

Most often, databases have poor designs and are not normalized. This can greatly affect the speed of database. In general all the database designs need to be normalized using the three normal forms. The normal forms above 3rd Normal Form are often called de-normalization forms, but what this really means is that they break some rules to make the Database faster. Normalization after the 3NF is often done at a later time, not during design, by DBA [8].

4.3 Filtering Query

Filter a query as much as possible. 'Where' clause is the most important part for optimization. Select only the fields that are required, and avoiding the use of "Select *". It will make the queries faster and will use less bandwidth.

Join clause needs to be used very carefully; it is expensive in terms of time. It is better to use it on indexed fields [8].

4.4 Add, remove or modify Indexes

All primary keys need indexes because they make joins faster. This also means that all tables need a primary key. Indexes may also be added on fields that are often used for filtering in the Where clauses.

Adding indexes is a careful process, because they need to be maintained by the database. If many updates are done, on that field, maintaining indexes might take more time than it saves [8].

4.5 Moving Queries to Stored Procedures

Stored Procedures (SP), are usually better and faster than queries for the following reasons:

Stored Procedures are compiled (SQL Code is not), making them faster than SQL code.

SPs don't use as much bandwidth, as a single SP may include or perform many queries. SPs also remains on the server until the final results are returned.

Stored Procedures are run on the server, which is typically faster.

Calculations in code (VB, Java, C++, ...) are not as fast as SP in most cases.

It keeps DB access code separate from presentation layer, which makes it easier to maintain (3 tiers model) [8].

4.6 Removing redundant Views

Views are a special type of Query, they are not tables. They are logical and not physical. When a view is called, it runs a query that generates a view and then performs query on the view.

If the same information is needed again and again views could be good. But if a view requires a filter, it's like running a query on a query, which will make it slower [8].

4.7 Tuning DB settings

DB can be tuned in many ways. Update statistics used by the optimizer, run optimization options, make the DB read-only, etc. It is mostly under the control of DBA [8].

4.8 Using Query Analyzers

In many Databases, there is a tool for running and optimizing queries. SQL Server has a tool called the Query Analyzer, which is very useful for optimizing. User can write queries, execute and, more importantly, see the execution plan and analyze what SQL Server does with the query [8].

5. OPTIMIZATION IN PRACTICE

5.1 Using relational operators

Relational operators such as >, <, =, != etc. are very helpful in query. Some of the queries may be optimized by using relational operators, provided the column is indexed. For example [4],

SELECT * FROM TABLE WHERE COLUMN>16

Now, the above query is not optimized due to the fact that the DBMS will have to look for the value 16 and then scan forward to value 16 and below. However, an optimized value will be,

SELECT * FROM TABLE WHERE COLUMN >= 15

This way the DBMS might jump straight to value 15 instead.

5.2 Avoiding NOT operator

It is much faster to search for an exact match (positive operator) such as using the LIKE, IN, EXIST or '=' symbol operator instead of a negative operator such as NOT LIKE, NOT IN, NOT EXIST or '!=' symbol. Using a negative operator will cause the search to find every single row to identify that they are ALL not belong or exist within the table. On the other hand, using a positive operator just stop immediately, once the result has been found [4].

5.3 Wildcard Vs. Substr

Most developer practiced Indexing. Hence, if a particular COLUMN has been indexed, it is best to use wildcard instead of using Substr [4].

#Poor Query

SELECT * FROM TABLE

WHERE substr (COLUMN,1,1) = 'value'.

The above will substr every single row in order to seek for the single character 'value'. On the other hand,

#Better Query

SELECT * FROM TABLE WHERE COLUMN = 'value%'.

Wildcard query will run faster if the above query is searching for all rows that contain 'value' as the first character.

5.4 Appropriate data types

Use the most efficient (concise) data types possible. It is unnecessary and sometimes dangerous to provide a huge data type when a smaller one will be more than sufficient to optimize a structure. Example, using the smaller integer types if possible to get smaller tables. MEDIUMINT is often a better choice than INT because a MEDIUMINT column uses 25% less space. On the other hand, VARCHAR will be better than longtext to store an email or small details.

5.5 Primary indexing

The primary column that is used for indexing should be made as concise as possible. This makes identification of each row easy and efficient by the DBMS [4].

5.6 String indexing

It is unnecessary to index the whole string when a prefix or postfix of the string can be indexed instead. Especially if the prefix or postfix of the string provides a unique identifier for the string, it is advisable to perform such indexing. Shorter indexes are faster, not only because they require less disk space, but because they also give you more hits in the index cache, and thus fewer disk seeks [4].

5.7 Limiting the result

Another common way of optimizing query is to minimize the number of rows it returns. If a table has a few billion records and a search query without limitation will just break the database with a simple SQL query such as this [4].

5.8 Using default value

In MySQL, take advantage of the fact that columns have default values. Insert values explicitly only when the value to be inserted differs from the default. This reduces the parsing that MySQL must do and improves the insert speed [4].

5.9 Avoiding IN subquery

Using a subquery within the IN operator like;

SELECT * FROM TABLE WHERE COLUMN IN

(SELECT COLUMN FROM TABLE)

Doing this is very expensive because SQL query will evaluate the outer query first before proceed with the inner query. Instead a better form could be;

SELECT * FROM TABLE, (SELECT COLUMN FROM TABLE) as dummytable WHERE dummytable.COLUMN = TABLE.COLUMN;

Using dummy table is better than using an IN operator to do a subquery [4].

5.10 Utilizing Union instead of OR

Indexes lose their speed advantage when using them in OR situations in MySQL at least. Hence, this will not be useful although indexes are being applied [4].

SELECT * FROM TABLE WHERE COLUMN_A = 'value' OR COLUMN_B ='value'

On the other hand, using Union such as this will utilize Indexes.

SELECT * FROM TABLE WHERE COLUMN_A = 'value'

UNION

SELECT * FROM TABLE WHERE COLUMN_B = 'value'

6. EXPERIMENTAL RESULTS

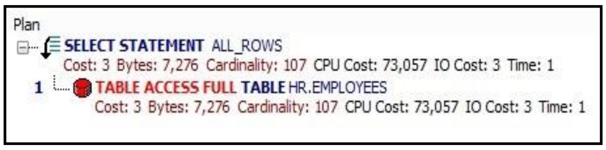
6.1 Result1

To retrieve name and salary of employees from R&D department.

Original query:

Select * From Employees

Analysis



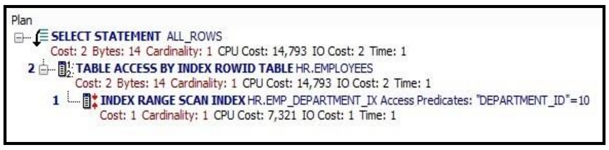
```
Plan
SELECT STATEMENT ALL_ROWS
  Cost: 3 Bytes: 7,276 Cardinality: 107 CPU Cost: 73,057 IO Cost: 3 Time: 1
1   TABLE ACCESS FULL TABLE HR.EMPLOYEES
      Cost: 3 Bytes: 7,276 Cardinality: 107 CPU Cost: 73,057 IO Cost: 3 Time: 1
```

Optimized query:

Select Name, Salary From Employees

Where Department_id = 10

Analysis



```
Plan
SELECT STATEMENT  ALL_ROWS
  Cost: 2  Bytes: 14  Cardinality: 1  CPU Cost: 14,793  IO Cost: 2  Time: 1
  2 TABLE ACCESS BY INDEX ROWID TABLE HR.EMPLOYEES
      Cost: 2  Bytes: 14  Cardinality: 1  CPU Cost: 14,793  IO Cost: 2  Time: 1
    1 INDEX RANGE SCAN INDEX HR.EMP_DEPARTMENT_IX Access Predicates: "DEPARTMENT_ID"=10
        Cost: 1  Cardinality: 1  CPU Cost: 7,321  IO Cost: 1  Time: 1
```

Comparative Analysis:

	Bytes	CPU Cost	I/O Cost
Original	7276	73057	3
Optimized	14	14793	2

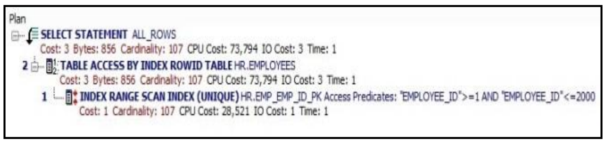
In the optimized version, the DB filters data because it filters faster than the program. The data that travels on the network will be much smaller, and therefore the performances will improve.

6.2 Result2

Original query:

Select salary From Employees Where EmpID >= 1 and EmpID <= 2000

Analysis



```
Plan
SELECT STATEMENT  ALL_ROWS
  Cost: 3  Bytes: 856  Cardinality: 107  CPU Cost: 73,794  IO Cost: 3  Time: 1
  2 TABLE ACCESS BY INDEX ROWID TABLE HR.EMPLOYEES
      Cost: 3  Bytes: 856  Cardinality: 107  CPU Cost: 73,794  IO Cost: 3  Time: 1
    1 INDEX RANGE SCAN INDEX (UNIQUE) HR.EMP_EMP_ID_PK Access Predicates: "EMPLOYEE_ID">=1 AND "EMPLOYEE_ID"<=2000
        Cost: 1  Cardinality: 107  CPU Cost: 28,521  IO Cost: 1  Time: 1
```

The original Query involves a lot of network bandwidth and will make whole system slow. Sometimes, Stored Procedure will be better off creating a temporary table, inserting data in it and returning it than going back and forth 10,000 times.

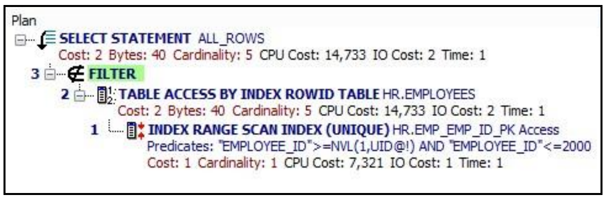
Optimized query:

SELECT salary FROM employees

WHERE employee_id >= NVL (1, UID)

AND employee_id <= 2000

Analysis



```
Plan
SELECT STATEMENT  ALL_ROWS
  Cost: 2  Bytes: 40  Cardinality: 5  CPU Cost: 14,733  IO Cost: 2  Time: 1
  3 FILTER
    2 TABLE ACCESS BY INDEX ROWID TABLE HR.EMPLOYEES
        Cost: 2  Bytes: 40  Cardinality: 5  CPU Cost: 14,733  IO Cost: 2  Time: 1
      1 INDEX RANGE SCAN INDEX (UNIQUE) HR.EMP_EMP_ID_PK Access
          Predicates: "EMPLOYEE_ID">=NVL(1,UID@!) AND "EMPLOYEE_ID"<=2000
          Cost: 1  Cardinality: 1  CPU Cost: 7,321  IO Cost: 1  Time: 1
```

Comparative Analysis:

	Bytes	CPU Cost	I/O Cost
Corrected	854	73794	3
Optimized	40	14733	2

6.3 Result3 (Weak Joins)

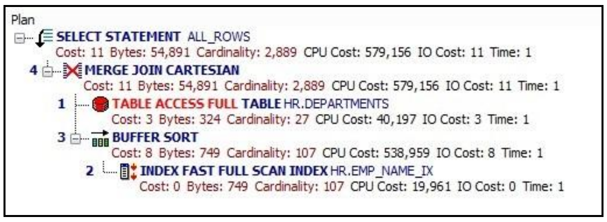
From the two tables, Orders and Customers. Customers can have many orders.

Original query:

Select e.first_name, d.department_name

From employees e, departments d;

Analysis

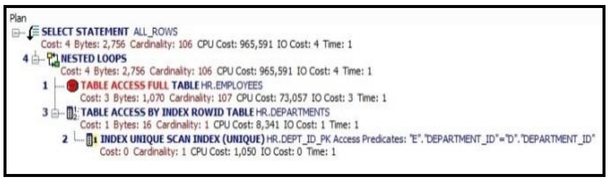


```
Plan
SELECT STATEMENT  ALL_ROWS
  Cost: 11  Bytes: 54,891  Cardinality: 2,889  CPU Cost: 579,156  IO Cost: 11  Time: 1
  4 MERGE JOIN CARTESIAN
      Cost: 11  Bytes: 54,891  Cardinality: 2,889  CPU Cost: 579,156  IO Cost: 11  Time: 1
    1 TABLE ACCESS FULL TABLE HR.DEPARTMENTS
        Cost: 3  Bytes: 324  Cardinality: 27  CPU Cost: 40,197  IO Cost: 3  Time: 1
    3 BUFFER SORT
        Cost: 8  Bytes: 749  Cardinality: 107  CPU Cost: 538,959  IO Cost: 8  Time: 1
      2 INDEX FAST FULL SCAN INDEX HR.EMP_NAME_IX
          Cost: 0  Bytes: 749  Cardinality: 107  CPU Cost: 19,961  IO Cost: 0  Time: 1
```

Optimized query1:

Select e.first_name, d.department_name from employees e, departments d where e.department_id = d.department_id;

Analysis



```
Plan
SELECT STATEMENT  ALL_ROWS
  Cost: 4  Bytes: 2,756  Cardinality: 106  CPU Cost: 965,591  IO Cost: 4  Time: 1
  4 NESTED LOOPS
      Cost: 4  Bytes: 2,756  Cardinality: 106  CPU Cost: 965,591  IO Cost: 4  Time: 1
    1 TABLE ACCESS FULL TABLE HR.EMPLOYEES
        Cost: 3  Bytes: 1,070  Cardinality: 107  CPU Cost: 73,057  IO Cost: 3  Time: 1
    3 TABLE ACCESS BY INDEX ROWID TABLE HR.DEPARTMENTS
        Cost: 1  Bytes: 16  Cardinality: 1  CPU Cost: 8,341  IO Cost: 1  Time: 1
      2 INDEX UNIQUE SCAN INDEX (UNIQUE) HR.DEPT_ID_PK Access Predicates: "E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID"
          Cost: 0  Cardinality: 1  CPU Cost: 1,050  IO Cost: 0  Time: 1
```

In that case, the join was not there at all or was not there on all keys. That would return so many records that your query might take hours. It's a common mistake for beginners.

Optimized query2:

Depending on the DB, the user will need to specify the Join type that is required in different ways.

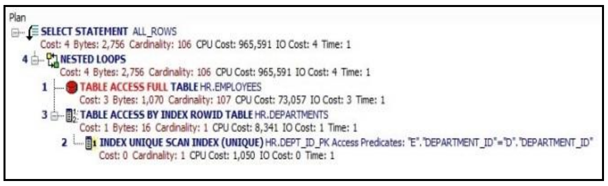
In SQL Server, the query would need to be corrected to:

Select e.first_name, d.department_name

from employees e inner join departments d

on e.department_id = d.department_id;

Analysis



```
Plan
SELECT STATEMENT  ALL_ROWS
  Cost: 4  Bytes: 2,756  Cardinality: 106  CPU Cost: 965,591  IO Cost: 4  Time: 1
  4 NESTED LOOPS
      Cost: 4  Bytes: 2,756  Cardinality: 106  CPU Cost: 965,591  IO Cost: 4  Time: 1
    1 TABLE ACCESS FULL TABLE HR.EMPLOYEES
        Cost: 3  Bytes: 1,070  Cardinality: 107  CPU Cost: 73,057  IO Cost: 3  Time: 1
    3 TABLE ACCESS BY INDEX ROWID TABLE HR.DEPARTMENTS
        Cost: 1  Bytes: 16  Cardinality: 1  CPU Cost: 8,341  IO Cost: 1  Time: 1
      2 INDEX UNIQUE SCAN INDEX (UNIQUE) HR.DEPT_ID_PK Access Predicates: "E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID"
          Cost: 0  Cardinality: 1  CPU Cost: 1,050  IO Cost: 0  Time: 1
```

Note that in SQL Server, Microsoft suggests to use the joins like in the second optimized form instead of the joins in the Where Clause because it will be more optimized.

Comparative Analysis:

	Bytes	CPU Cost	I/O Cost
Original	54891	579156	11
Optimized1	2756	965591	4
Optimized2	2756 (same)	965591 (same)	4 (same)

6.4 Result4 (Weak Filters)

This is a more complicated example, but it illustrates filtering at its best.

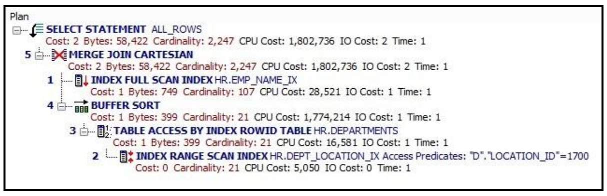
It is based on two tables – Products (ProductID, DescID, Price) and Description (DescID, LanguageID, Text). There are 100,000 Products and unfortunately all of them are required.

There are 100 languages (LangID = 1 = English). Product descriptions are required only in English language. Expected number of products is 100000 (ProductName, Price).

Original query:

Select d.department_name, e.first_name from employees e inner join departments d on d.department_id = d.department_id where d.location_id = 1700;

Analysis



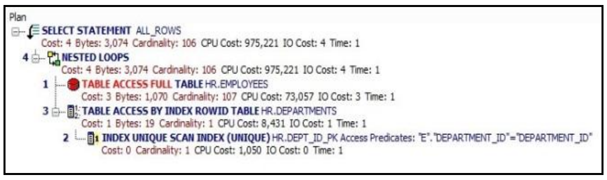
```
Plan
SELECT STATEMENT ALL_ROWS
  Cost: 2 Bytes: 58,422 Cardinality: 2,247 CPU Cost: 1,802,736 IO Cost: 2 Time: 1
 5 MERGE JOIN CARTESIAN
    Cost: 2 Bytes: 58,422 Cardinality: 2,247 CPU Cost: 1,802,736 IO Cost: 2 Time: 1
   1 INDEX FULL SCAN INDEX HR.EMP_NAME_IX
      Cost: 1 Bytes: 749 Cardinality: 107 CPU Cost: 28,521 IO Cost: 1 Time: 1
   4 BUFFER SORT
      Cost: 1 Bytes: 399 Cardinality: 21 CPU Cost: 1,774,214 IO Cost: 1 Time: 1
     3 TABLE ACCESS BY INDEX ROWID TABLE HR.DEPARTMENTS
        Cost: 1 Bytes: 399 Cardinality: 21 CPU Cost: 16,581 IO Cost: 1 Time: 1
       2 INDEX RANGE SCAN INDEX HR.DEPT_LOCATION_IX Access Predicates: "D"."LOCATION_ID"=1700
          Cost: 0 Cardinality: 21 CPU Cost: 5,050 IO Cost: 0 Time: 1
```

That works but it will be really slow because the DB needs to match 100,000 records with 10,000,000 records and then filter that Where LangID = 1. The solution is to filter On LangID=1 before joining the tables.

Optimized query:

Select d.department_name, e.first_name from (select department_id, department_name from departments where location_id=1700) d inner join employees e on d.department_id = e.department_id;

Analysis



```
Plan
SELECT STATEMENT ALL_ROWS
  Cost: 4 Bytes: 3,074 Cardinality: 106 CPU Cost: 975,221 IO Cost: 4 Time: 1
 4 NESTED LOOPS
    Cost: 4 Bytes: 3,074 Cardinality: 106 CPU Cost: 975,221 IO Cost: 4 Time: 1
   1 TABLE ACCESS FULL TABLE HR.EMPLOYEES
      Cost: 3 Bytes: 1,070 Cardinality: 107 CPU Cost: 73,057 IO Cost: 3 Time: 1
   3 TABLE ACCESS BY INDEX ROWID TABLE HR.DEPARTMENTS
      Cost: 1 Bytes: 19 Cardinality: 1 CPU Cost: 8,431 IO Cost: 1 Time: 1
     2 INDEX UNIQUE SCAN INDEX (UNIQUE) HR.DEPT_ID_PK Access Predicates: "E"."DEPARTMENT_ID"="DEPARTMENT_ID"
        Cost: 0 Cardinality: 1 CPU Cost: 1,050 IO Cost: 0 Time: 1
```

Now, that will be much faster. Also make that query a Stored Procedure to make it faster.

Comparative Analysis:

	Bytes	CPU Cost	I/O Cost
Original	58422	1802736	2
Optimized	3074	975221	4

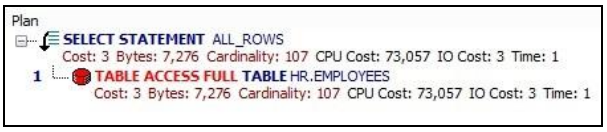
6.5 Result5 (Views)

Create View v_Employees AS

Select * From Employees

Select * From v_Employees

Analysis



```
Plan
SELECT STATEMENT ALL_ROWS
  Cost: 3 Bytes: 7,276 Cardinality: 107 CPU Cost: 73,057 IO Cost: 3 Time: 1
 1 TABLE ACCESS FULL TABLE HR.EMPLOYEES
    Cost: 3 Bytes: 7,276 Cardinality: 107 CPU Cost: 73,057 IO Cost: 3 Time: 1
```

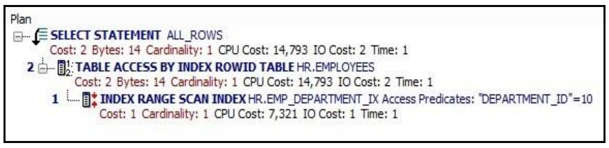
This is just like running Select * From Employees twice.

If the intended data is always for employees of R&D and would not like to give the rights to everyone on that table because of salaries being confidential, the following view may be used:

Create view v_adminEmployees as select first_name, salary from employees where department_id=10;

select * from v_adminEmployees; (Dept 1 is R&D).

Analysis



```
Plan
SELECT STATEMENT ALL_ROWS
  Cost: 2 Bytes: 14 Cardinality: 1 CPU Cost: 14,793 IO Cost: 2 Time: 1
 2 TABLE ACCESS BY INDEX ROWID TABLE HR.EMPLOYEES
    Cost: 2 Bytes: 14 Cardinality: 1 CPU Cost: 14,793 IO Cost: 2 Time: 1
   1 INDEX RANGE SCAN INDEX HR.EMP_DEPARTMENT_IX Access Predicates: "DEPARTMENT_ID"=10
      Cost: 1 Cardinality: 1 CPU Cost: 7,321 IO Cost: 1 Time: 1
```

Comparative Analysis:

	Bytes	CPU Cost	I/O Cost
Original	7276	73057	3
Corrected	14	14793	2

**Example 5 result as same as example 1.*

Then rights may be assigned to the View v_R&DEmployees to some people and would restrict the rights to Employees table to the DBA only.

7. CONCLUSION

In this study we analyzed the fundamental pragmatic aspects of query optimization that underlies all optimization algorithms known in the research literature. Results of the experiments conducted have been verified using Optimization Analyzer tool, and presented as analysis. Query optimization is essential for large information retrieval systems.

8. REFERENCES

[1] Andrew Eisenberg and Jim Melton, June 2004. An Early Look at XQuery API for Java™ (XQJ). *ACM SIGMOD Record*, Vol. 33.

[2] Avi Silbershatz, Hank Korth and S. Sudarshan, 2002. *Database System Concepts*, 4th Edition. McGraw-Hill.

[3] Chiang Lee, Chi-Sheng Shih and Yaw-Huei Chen, 2001. A Graph-theoritic model for optimizing queries involving methods. *The VLDB Journal — The International Journal on Very Large Data Bases*, Vol. 9, Issue 4, Pages 327-343.

[4] Clay Lua, 15 ways to Optimize your Queries, [hungred.com/useful-information/ways-optimize-sql-queries].

[5] Hsiao-Fei Liu, Ya-Hui Chang and Kun-Mao Chao, June 2004. An Optimal Algorithm for Querying Tree Structures and its Applications in Bioinformatics. *ACM SIGMOD Record* Vol. 33.

[6] Jingren Zhou and Kenneth A. Ross, June 2004. Buffering Database Operations for Enhanced Instruction Cache Performance. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, Pages 191-202.

[7] Michael L. Rupley, 2008. Jr. Introduction to Query Processing and Optimization. Indiana University at South Bend.

[8] Olivier Perron, 2001. How to Optimize Queries [Theory and Practice], [serverwatch.com/tutorials/article.php/2175621].

[9] Ramez Elmasri and Shamkant B. Navathe. 1994. Fundamentals of Database Systems, second edition. Addison-Wesley Publishing Company.

[10] Reza Sadri, Carlo Zaniolo, Amir Zarkesh and Jafar Adibi. June 2004. Expressing and Optimizing Sequence Queries in Database Systems. *ACM Transactions on Database Systems*, Vol. 29, Issue 2, Pages 282-318.

[11] Reza Sadri, Carlo Zaniolo, Amir Zarkesh and Jafar Adibi. May 2001. Optimization of Sequence Queries in Database Systems. In *Proceedings of the twentieth ACM SIGMOD-SIGACTSIGART symposium on Principles of database systems*, Pages 71-81.

[12] Thomas Schwentick. March 2004. XPath Query Containment. *ACM SIGMOD Record*, Vol. 33.