Volume 4, Issue 12, December 2014 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Comparative Analysis of Query Optimization in the Object-Oriented Database & Relational Databases Using Clauses

Abhijit Banubakode
Rajarshi Shahu College of Engineering,
Pune, India

Virandra Dakhode
Smt. Kashibai Navale College of Engineering
Pune, India

Abstract— In this paper, we present an approach using database clauses that permits to enrich technique of query optimization existing in the object-oriented databases and the comparative analysis of query optimization for relational databases and object oriented database based on cost, cardinality and no of bytes. Focus is on queries using where, group-by and having clauses. Our experimental study shows that the improvement in the quality of plans is significant only with decrease in cost. Looking at the success of query optimization in the relational model, our approach inspires itself of these optimization techniques and enriched it so that they can support the new concepts introduced by the object oriented databases.

Keywords— Include at least 5 keywords or phrases Query Optimization, Relational Databases, Object-Oriented Databases, Where clause, Group-by clause, having clause, Cost, Cardinality and Bytes

I. INTRODUCTION

Query optimization plays an important role in database system, without which performance of database system will not be yield very significant improvement. Conventional optimization techniques used in relational database systems were not design to cope with heterogeneous structures and of particular not suitable to handle collection objects[14].Group-by clause is used to group the rows in a table based on certain criteria. The grouping criteria are specified in the form of an expression. A group-by query groups the data from its source tables and produces a single summary row for each row group. The columns named in the group by clause are called grouping columns of the query [1]. Decision-support system use the SQL operation of group-by and aggregate functions extensively in formulating queries . For example, queries that create summary data are of great importance in data ware house applications. These queries partition data in several groups (e.g.in business sectors) and aggregate on some attributes (e.g. sum of total sales) . A recent study of customer. queries in DB2 [2] have found that the group-by construct occurs in a large fraction of SQL queries used in decision-support applications. Therefore, efficient processing and optimization of queries with group-by and aggregation are of significant importance [3]. For a single-block SQL query, the group-by operator is traditionally executed after all the join have been processed. Conventional relational optimizers do not exploit the knowledge about the group-by clause in a query. In this paper, we present significant techniques for processing and optimization of queries with group-by. Using select statement, we selected specific rows from a table that satisfied singular or multiple search conditions for this we used were clause. For example, if we know the details of employees in department number 20 in the table emp , then we include where clause in the select statement. With where clause, the select statement retrieves those rows that meet the search conditions the where clause follows form clause. The having clause works very much like where clause, except that its logic is only related to the result of group function. This paper is organized as follows. Next two sections describe Application and Preliminaries Notation. Section IV reviews the optimization technique. Section V establishes the Experimental setup. Section VI includes the Query Evaluation aspects. Section VII discusses the Statistical estimates. Section VIII list optimizer hints for estimating cardinality of different plans Finally, Section IX concludes the paper. We referered the recent paper [4], Yan and Larson identified a transformation that enables pushing the group-by past joins. Their approach is based on deriving two queries, one with and the other without a group-by clause, from the given SQL query. The result of the given query is obtained by joining the two queries so formed. Thus, in their approach, given a query, there is a unique alternate placement for the group-by operator. Observe that the transformation reduces the space of choices for join ordering since the ordering is considered only within each query. Prior work on group-by has addressed the problem of pipelining group-by and aggregation with join [5, 6] as well as use of group-by to flatten nested SQL queries [7, 5, 8, and 9]. But, these problems are orthogonal to the problem of optimizing queries containing group-by clause

II. APPLICATION

We are considering an example of retail banking system. The bank is organized into various branches and each branch located in a particular city and monitors the assets. Bank customers are identified by their cust-id values. Bank offers two type of accounts i.e. saving account & checking account with loan facility thus the object with and attributes in the schema are:

Customer (cust_name,cust_street,cust_city)
Branch (branch_city,branch_name,assets)
Account (acct_no,branch-name,balance)
Depositor (cust_name,acct_no)
Loan (loan_no,branch_name,amount)
Borrower (cust_name, loan_no)

Fig 1: Object Considered for banking system

Transformations: To begin with we observe that, since a group-by reduces the cardinality of a relation, an early evaluation of group-by could result in potential saving in the costs of the subsequent joins. We present an example that illustrates a transformation based on the above fact. An appropriate application of such a transformation could result in plans that are superior to the plans produced by conventional optimizers by an order of magnitude or more.

Example: Consider the query that computes count of branches located in a particular city and total count of branches in each city. Many alternative plans are possible. We consider the following: First, group-by clause applied after condition and hence search time is more and CPU cost is high. In other words we first check the condition and then group on branch city. Second, group-by clause applied before condition hence search time is less and CPU cost is less. Here we group on branch city first and then check the condition. Third we split conditional operator into two separate symbols and performance were measured

III. PRELIMINARIES AND NOTATION

We will follow the operational semantics associated with SQL queries [10, 11]. We assume that the query is a single block SQL query, as below

SELECT All <columnlist> AGG1(bl)..**AGG**2(bn)
FROM <tablelist>
WHERE cond1 And cond2 . . . And condn
GROUP BY col1,..col2 HAVING condition

Fig 2: The typical query under consideration

The WHERE clause of the query is a conjunction of simple predicates. SQL semantics require that <columnlist> must be among col1,.. colj. In the above notation,AGGi…..AGGn represent built-in SQL aggregate functions.In this paper, we will not be discussing the cases where there is a an ORDER BY clause in the query. We will also assume that there are no nulls in the database. These extensions are addressed in [12]. We refer to columns in {b1, ..bn} as the aggregating columns of the query. The columns in (col1, ..colj} are called grouping columns of the query. The functions{AGG1, ..AGGn} are called the aggregating functions of the query. For the purposes of this paper, we inluded Count as well as cases where the aggregate functions apply on columns with the qualifier. Having imposes a condition on the group by clause, which further filters the groups created by the group by

IV. OPTIMIZATION

To illustrate the regular RDBMS and the object oriented query optimizations, we consider a typical Retail Banking System**, as** the database

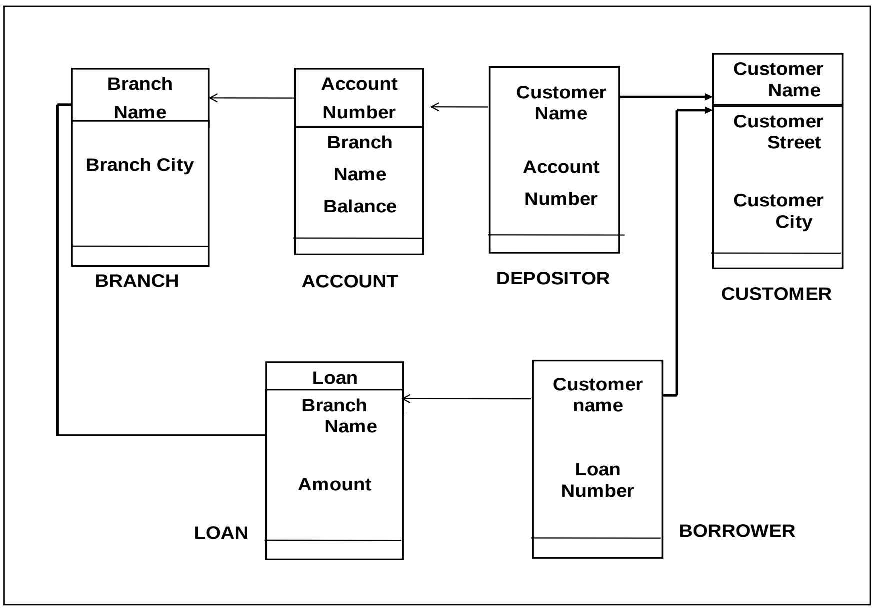

Fig 3: Object Oriented Schema for Retail Banking

A. Query Optimization in RDB

As an example, consider the query to **find the number of branch in each city except pune.**

We wrote different query evaluation plans and tested them on a Oracle10g environment. We propose to do the same with an OODB later.The various equivalent query evaluation plans for RDB could be: -

Plan1 :	**Plan2 :**
Select /*+index(rbranch)*/ branch_city , count(*) **From** rbranch **Where** branch_city != 'pune' group by branch_city	**Select** /*+ index(rbranch)*/ branch_city,count(*) **From** rbranch **Group by** branch_city **Having** Branch_city != 'pune'

Plan3
Select /*+ index_combine(rbranch) */ branch_city, count(*) **From** rbranch
Where branch_city < 'pune' or branch_city > 'pune'
Group by branch_city
The Algebraic expression trees for various plans are:

Algebraic expression tree for plan1

```
∏branch_city,count(*)
 |
σ branch-city!=pune
 |
rbranch
 |
group-by branch-city
```

Algebraic expression tree plan 2

```
∏branch_city,count(*)
|
group-by branch-city Having Branch_city != 'pune'
|
σ branch-city!=pune
 |
rbranch
```

Algebraic expression tree plan 3

```
∏branch_city,count(*)
 |
σ branch-city < 'pune' or  branch-city > 'pune'
 |
rbranch
 |
group-by branch-city
```

The execution plans obtained using oracles 10g for above cases are:
Execution Plan for Plan 1

```
  0      SELECT STATEMENT Optimizer=CHOOSE
                  (Cost=830 Card=2 Bytes=34)
  1   0   SORT (GROUP BY) (Cost=830 Card=2 Bytes=34)
  2   1     TABLE ACCESS (BY INDEX ROWID) OF   'RBRANCH' (Cost=826 Card=2 Bytes=34)
  3   2       INDEX (FULL SCAN) OF 'SYS_C002720'
                    (UNIQUE) (Cost=26 Card=2)
```

Execution Plan for Plan2

```
  0     SELECT STATEMENT Optimizer=CHOOSE
  1         (Cost=830 Card=41 Bytes=697)
  1   0   FILTER SORT (GROUP BY)
         (Cost=830 Card=41 Bytes=697)
  3   2   TABLE ACCESS (BY INDEX ROWID) OF
          'RBRANCH' (Cost=826 Card=41 Bytes=697)
  4   3 INDEX (FULL SCAN) OF 'SYS_C002720'
         (UNIQUE) (Cost=26 Card=41)
```

Execution Plan for Plan 3

```
  0      SELECT STATEMENT Optimizer=CHOOSE
                (Cost=46 Card=4 Bytes=68)
  1   0   SORT (GROUP BY) (Cost=46 Card=4 Bytes=68)
  2   1    TABLE ACCESS (BY INDEX ROWID) OF
           'RBRANCH' (Cost=42 Card=4 Bytes=68)
  3   2    BITMAP CONVERSION (TO ROWIDS)
  4   3    BITMAP CONVERSION (FROM ROWIDS)
  5   4    SORT (ORDER BY)
  6   5    INDEX (FULL SCAN) OF 'SYS_C002720'
          (UNIQUE) (Cost=33)
```

Plan 3 is the case where group by is delayed most.

B. Query Optimization in OODB

An OODB requires that OOTypes be created and OOTables be created. This is what we have done first.

Creation of Object Oriented Type	**Creation of Object Oriented Table**
Create Type **Branchdet_Ty** as object (Branch_City Varchar2 (30), Assets Number (26, 2)); Create Type **Accountdet_Ty** as object (Branch_Name Varchar(30), Balance Number (12, 2));	Create Table **Branch1** (Branch_Name Varchar2 (30) Primary Key, Branchdetail Branchdet_Ty); Create Table **Account1** (Account_Number Varchar(15), Accountdetail Accountdet_Ty);

Now we are in a position to design query evaluation plans as required: The query evaluation plans for OODB are: -

Plan1	**Plan2**	**Plan3**
Select /*+ index(branch1)*/ client.branchdetail.branch_city ,count(*) **From** branch1 client **Where** client.branchdetail.branch_city != 'pune' **Group by** client.branchdetail.branch_city	**Select** /*+ index(branch1)*/ client.branchdetail.branch_city ,count(*) **From** branch1 client G**roup by** client.branchdetail.branch_city **Having** client.branchdetail.branch_city != 'pune'	**Select** /*+ index_combine(client) */ client.branchdetail.branch_city, **count(*)** **From** branch1 client **where** client.branchdetail.branch_city != 'pune' **Group by** Client.branchdetail.branch_city

Algebraic expression tree for plan1

$\prod$branchdetail.branch_city,count(*)
|
σ branchdetail.branch-city!=pune
|
branch1
|
group-by branchdetail. branch-city

Algebraic expression tree for plan1

$\prod$ branchdetail.branch_city,count(*)
|
group by branchdetail.branch-city
|
σ branchdetail.branch-city!= pune **having**

```
| client.branchdetail.branch_city != 'pune'

branch1
```

Algebraic expression tree for plan1

```
Π branchdetail.branch_city,count(*)
|
σ branchdetail.branch-city <'pune' or
|   branchdetail.branch-city > 'pune'

branch1
|
```

group-by branchdetail.branch-city

The execution plans for above alternatives generated using oracle using oracle 10g are:

```
Execution Plan for Plan1
            SELECT STATEMENT Optimizer=HINT:
            ALL_ROWS (Cost=5 Card=2 Bytes=34)
   1   0   SORT (GROUP BY) (Cost=5 Card=2 Bytes=34)
   2   1     TABLE ACCESS (FULL) OF 'BRANCH1'
              (Cost=1 Card=2 Bytes=34)
```

```
Execution Plan for plan2
            SELECT STATEMENT Optimizer=CHOOSE
            (Cost=5 Card=41 Bytes=697)
   1   0   FILTER
   2   1     SORT (GROUP BY) (Cost=5 Card=41
                                      Bytes=697)
   3   2       TABLE ACCESS (FULL) OF 'BRANCH1'
                (Cost=1 Card=41 Bytes =697)
```

```
Execution Plan for Plan3
  0        SELECT STATEMENT Optimizer=CHOOSE
           (Cost=46 Card=2 Bytes=34)
   1   0   SORT (GROUP BY) (Cost=46 Card=2 Bytes=34)
   2   1     TABLE ACCESS (BY INDEX ROWID) OF
             'BRANCH1' (Cost=42 Card =2 Bytes=34)
   3   2       BITMAP CONVERSION (TO ROWIDS)
   4   3         BITMAP CONVERSION (FROM ROWIDS)
   5   4           SORT (ORDER BY)
   6   5      INDEX (FULL SCAN) OF 'SYS_C002715'
              (UNIQUE) (Cost = 33)
```

V. ANALYSIS: COMPARISON OF RESULTS

We did an experimental study. We achieved statistically significant improvement in the quality of plans with a modest decrease in the optimization cost. The experiments were conducted using on oracle database Table: 1 shows the Query Comparisons of RDBMS & OODBMS Based on Cost, Cardinality & No of Bytes. From experimental setup we observed that there is significant improvement after query optimization in object oriented database.

Table 1

Relational Database (RDB)				**Object Oriented Database (OODB)**			
Group-By Clause				**Group-By Clause**			
Plans	**Query Performance**			**Plans**	**Query Performance**		
	Cost	**Card**	**Bytes**		**Cost**	**Card**	**Bytes**
Plan 1	**2512**	**8**	**104**	**Plan1**	**11**	**8**	**102**
Plan 2	**2512**	**164**	**2091**	**Plan2**	**11**	**123**	**2091**
Plan 3	**167**	**12**	**204**	**Plan3**	**167**	**6**	**102**

Query performance comparison of RDB and OODB for GROUP BY clause

VI. COST ESTIMATION

Given a query there are many equivalent alternative algebraic expression for each expression there are many ways to implement them as operators. The cost estimates are based upon I/O, CPU and Memory resources required by each query operation, and the statistical information about the database object such as Table, Indexes and Views. In a large number of systems, information on the data distribution on a column is provided by *histograms*. A histogram divides the values on a column into k buckets. In many cases, k is a constant and determines the degree of accuracy of the histogram. However, k also determines the memory usage, since while optimizing a query; relevant columns of the histogram are loaded in memory. There are several choices for "bucketization" of values. In many database systems, equi-depth (also called equi-height) histograms are used to represent the data distribution on a column. If the table has n records and the histogram has k buckets, then an equi-depth histogram divides the set of values on that column into k ranges such that each range has the same *number* of values, i.e., n/k. compressed histograms place frequently occurring values in singleton buckets.

Fig 4 and Fig 5 shows a histogram for query performance in RDB & OODB

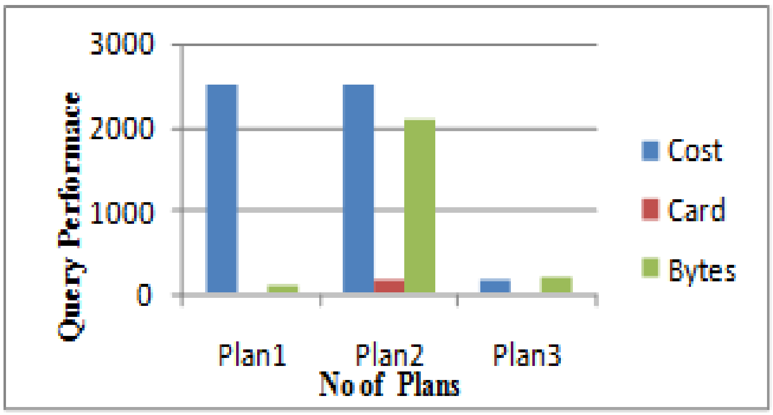

Fig 4: Query performance histogram for RDB

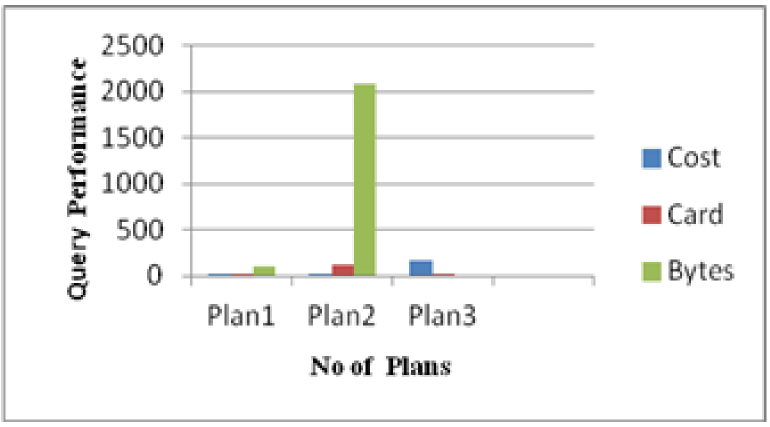

Fig 5: Query performance histogram for OODB

Statistical Estimates

We measured query performance using statistics measure such as Mean and Standard deviation. Mean, also known as arithmetic average, is the most common measure of central tendency and may be defined as the value which we get by dividing the total of the values of various given items in a series by the total number of items. Standard deviation is most widely used measure of dispersion of a series and is commonly denoted by 'σ' (pronounced as sigma). Standard deviations defined as the square-root of the average of squares of deviations, when such deviation for the values of individual items in a series is obtained from the arithmetic average. Mean and Standard deviation worked as under

$$\textbf{Mean}(\overline{X}) = \frac{\sum X_i}{n} = \frac{X_1 + X2 + \ldots\ldots + Xn}{n}$$

$$\textbf{Standard Deviation } (\sigma) = \sqrt{\frac{\sum (Xi - \overline{X})^2}{n}}$$

Table 2 defines the statistical measures of Cost, Card and Bytes for P1, P2 and P3. The Cost, Card and Bytes obtained after optimization in object oriented database is less as compared to Cost, Card and Bytes obtained in relational database using group by clause.

Table 2 Comparative statistical Measures of Cost, Card and Bytes For P1, P2 and P3

Relational Database (RDB)			Object Oriented Database (OODB)		
Group By Clause			Group By Clause		
Query Attrib-utes	Statistical Measures		Query Attrib-utes	Statistical Measures	
	$\overline{X}$	σ		$\overline{X}$	σ
Cost	1730.3	1353.8	Cost	63	52.51
Card	61.33	88.92	Card	45.66	66.9762
Bytes	172.33	59.22	Bytes	765	1148.34

VII. OPTIMIZER HINTS

While generating different query plans we use Optimizer hints. Hints make decisions usually made by the optimizer. Hints provide a mechanism to the optimizer to choose a certain query execution plan based on the specific criteria. [13]
Hints falls into the following general classifications: Single-table hints are specified on one table or view. INDEX and USE_NL are examples of single-table hints. Multi-table hints are like single-table hints, except that the hint can specify one or more tables or views. The USE_NL is not considered a multi-table hint because it is actually a shortcut for USE_NL and USE_NL Query block hints operate on single query blocks. STAR_TRANSFORMATION and UNNEST are examples of query block hints. Statement hints apply to the entire SQL statement. ALL_ROWS is an example of a statement hint .Hint Syntax can send hints for a SQL statement to the optimizer by enclosing them in a comment within the statement. A block in a statement can have only one comment containing hints following the SELECT, UPDATE, MERGE, or DELETE keyword.
Following types of hints we use in our experimentation.
The ALL_ROWS hint explicitly chooses the query optimization approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption). The PARALLEL hint lets you specify the desired number of concurrent servers that can be used for a parallel operation. The hint applies to the SELECT, INSERT, UPDATE, and DELETE portions of a statement, as well as to the table scan portion. If any parallel restrictions are violated
then the hint is ignored. The INDEX_FFS hint causes a fast full index scan to be performed rather than a full table scan.
The INDEX hint explicitly chooses an index scan for the specified table. we use the INDEX hint for domain, B-tree, bitmap, and bitmap join indexes. However, Oracle recommends using INDEX_COMBINE rather than INDEX for the combination of multiple indexes, because it is a more versatile hint. The INDEX_DESC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in descending order of their indexed values. In a partitioned index, the results are in descending order within each partition.
The INDEX_COMBINE hint explicitly chooses a bitmap access path for the table. If no indexes are given as arguments for the INDEX_COMBINE hint, then the optimizer uses whatever Boolean combination of indexes has the best cost estimate for the table. If certain indexes are given as arguments, then the optimizer tries to use some Boolean combination of those particular indexes.

Table 3 Measures of Cardinality for P1, P2, P3, P4, P5and P6 using Indexing Hint

Plans	**Indexing Hint**	**Cardinality**		**Cost**
		Affected	**Not-affected**	
Plan 1	**/*+ ALL_ROWS */**		**123**	**3**
Plan 2	**/*+ PARALLEL(CLIENT, 2) */**		**123**	**3**
Plan 3	**/*+ INDEX_FFS(CLIENT) */**		**123**	**12**
Plan 4	**/*+ INDEX(CLIENT) */**		**123**	**78**
Plan 5	**/*+ INDEX_DESC(CLIENT) */**		**123**	**3**
Plan 6	**/*+ INDEX_COMBINE(CLIENT) */**	**82**		**105**

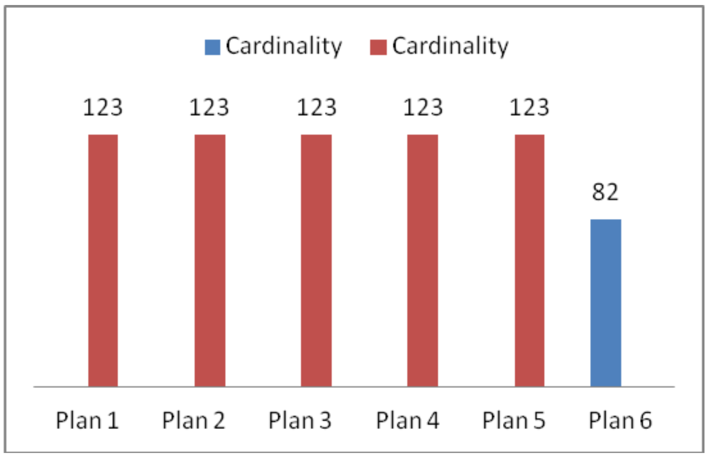

Fig 6: Cardinality performance histogram for OODB

Table 3 shows measures of cardinality for P1, P2, P3, P4, P5 and P6 using various indexing hint. We constructed six queries Plans and uses six different types of indexing hints and measured the cardinality performance. Fig 6 shows cardinality performance histogram for object oriented database, we observed that out of six plans cardinality is not change in five cases; hence cardinality in the object oriented database does not affected when the indexing method is change.

VIII. CONCLUSIONS

One of the biggest problems in Object Oriented Database is the optimization of queries. Due to these problems optimization of object-oriented queries is extremely hard to solve and is still in the research stage. This work is expected to be a significant contribution to the Database Management area which will not only reduce time or efforts but will also improve the quality and will reduce the cost. From above results we could conclude that first if the group-by clause applied before conditional statement then there is significant cost reduction and second cardinality in the object oriented database does not affected when the indexing methods are change.

REFERENCES

[1] Surajit Chaudhuri," An Overview of Query Optimization in Relational Systems", In Proc.of ACM SIGMOD,San Francisco,1987
[2] Tsang A., Olschanowsky M."A Study of Database 2 Customer Queries," IBM Santa Teresa Labortary, TR-o3.413.
[3] Chaudhuri,S.,Shim K."Including Group-By in Query Optimization." In Proc. Of VLDB, Santiago, 1994.
[4] Yan W. P., Larson P. A., "Performing Group-By before Join," International Conference on Data Engineering, Feb. 1993, Houston.
[5] DayaI U. "Of Nests and Trees: A Unified Approach to Processing Queries that contain sub queries, aggregates and quantifiers," in Proceedings of the 13th VLDB, Aug 1987.
[6] Klug A. "Access Paths in the ABE Statistical Query Facility," in Proceedings of 1982 ACMSIGMOD Conference on the Management of Data.
[7] Kim W. "On Optimizing an SQL-like Nested Query," in ACM Transactions on Database Systems, 7(3):443-469, Sep 1982.
[8] Ganske R. A., Wong H. "Optimization of Nested Queries Revisited," in Proceedings of 1987 ACMSIGMOD Conference on Management of Data, San Francisco, May 1987.
[9] Murahkrishna M. "Improved Unnesting Algorithmsfor Join Aggregate SQL Queries," in Proceedings of the 18th VLDB, 1992
[10] Date C. J., Darwen H. "A Guide to the SQL Standard: A User's Guide," Addison-Wesley, 1993.
[11] ISO. Database Language SQL ISO/IEC, Document ISO/IEC 9075:1992. Also available as ANSI Document ANSI X3.135-1992.
[12] Chaudhuri S., Shim K. "The promise of Early Aggregation," HPL Technical Report, 1994.
[13] "Understanding Optimizer hints", Oracle database Performance Tuning Guide
[14] Cluet,S. and Delobel,C., " A General Framework for the Optimization of Object-Oriented Queries".Proceedings of the ACM SIGMOD Conference,pp.383-392,199.

ABOUT AUTHOR

Abhijit Banubakode received ME degree in Computer Engineering from Pune Institute of Computer Technology (PICT), University of Pune ,India in 2005 and BE degree in Computer Science and Engineering from Amravati University, India, in 1997. Presently he is perusing his Ph.D. from Symbiosis Institute of Research and Innovation(SIRI), a constituent of Symbiosis International University (SIU), Pune, India. His current research area is Query Optimization in Compressed Object-Oriented Database Management Systems

(OODBMS). Currently he is working as Assistant Professor in Department of Information Technology, Rajarshi Shahu College of Engineering,Pune, India .He is having 13 years of teaching experience. He is a member of International Association of Computer Science and Information Technology (IACSIT), ISTE, CSI and presented six papers in International and National conference.

Virendra Dakhode received ME degree in Computer Engineering from Rajarshi Shahu College of Engineering, University of Pune, India in 2013 and BE degree in Information Technology from Amravati University, India, in 2007.His current research area is database and data mining. Currently he is working as Assistant Professor in Department of Computer Engineering, Smt. Kashibai Nawale College of Engineering, Pune-41 India .He is having 7 years of teaching experience. He is a member of ISTE, CSI and presented three papers in International journal.

Appendix 1: Queries Processing Used for Testing for RDB

Query Plans	**Query Statement**	**Query Plans**	**Query Statement**
Plan1	SELECT /*+ INDEX(RBRANCH)*/ BRANCH_CITY,COUNT(*) FROM RBRANCH GROUP BY BRANCH_CITY HAVING BRANCH_CITY != 'PUNE'	**Plan10**	SELECT /*+ PARALLEL(RBRANCH1, 2) */ BRANCH_CITY, COUNT(*) FROM RBRANCH RBRANCH1 WHERE RBRANCH1.ROWID IN (SELECT /*+ PARALLEL(RBRANCH2, 2) */ RBRANCH2.ROWIDFROM RBRANCH RBRANCH2 WHERE BRANCH_CITY < 'PUNE' UNION SELECT /*+ PARALLEL(RBRANCH3, 2) */ RBRANCH3.ROWID FROM RBRANCH RBRANCH3 WHERE BRANCH_CITY > 'PUNE') GROUP BY BRANCH_CITY
Plan2	SELECT /*+ ALL_ROWS */ BRANCH_CITY, COUNT(*) FROM RBRANCH WHERE BRANCH_CITY != 'PUNE' GROUP BY BRANCH_CITY	**Plan11**	SELECT /*+ INDEX_COMBINE(RBRANCH) */ BRANCH_CITY,COUNT(*) FROM RBRANCH WHERE BRANCH_CITY != 'PUNE' GROUP BY BRANCH_CITY
Plan3	SELECT /*+ INDEX_COMBINE(RBRANCH) */ BRANCH_CITY, COUNT(*) FROM RBRANCH WHERE BRANCH_CITY < 'PUNE' OR BRANCH_CITY > 'PUNE' GROUP BY BRANCH_CITY	**Plan12**	SELECT /*+ INDEX(RBRANCH) */ BRANCH_CITY,COUNT(*) FROM RBRANCH WHERE BRANCH_CITY < 'PUNE' OR BRANCH_CITY > 'PUNE' GROUP BY BRANCH_CITY
Plan4	SELECT /*+ PARALLEL(RBRANCH, 2) */ BRANCH_CITY, COUNT(*) FROM RBRANCH WHERE BRANCH_CITY != 'PUNE' GROUP BY BRANCH_CITY	**Plan13**	SELECT BRANCH_CITY, COUNT(*) FROM RBRANCH RBRANCH1 WHERE RBRANCH1.ROWID IN (SELECT /*+ INDEX_COMBINE(RBRANCH2) */ RBRANCH2.ROWID FROM RBRANCH RBRANCH2 WHERE BRANCH_CITY < 'PUNE' UNION SELECT RBRANCH3.ROWID

			FROM RBRANCH RBRANCH3 WHERE BRANCH_CITY > 'PUNE') GROUP BY BRANCH_CITY
Plan5	SELECT /*+ PARALLEL(RBRANCH, 2) */ BRANCH_CITY, COUNT(*) FROM RBRANCH WHERE BRANCH_CITY < 'PUNE' OR BRANCH_CITY > 'PUNE' GROUP BY BRANCH_CITY	**Plan14**	SELECT BRANCH_CITY, COUNT(*) FROM RBRANCH RBRANCH1 WHERE RBRANCH1.ROWID IN (SELECT RBRANCH2.ROWID FROM RBRANCH RBRANCH2 WHERE BRANCH_CITY < 'PUNE' UNION SELECT /*+ INDEX_COMBINE(RBRANCH3) */ RBRANCH3.ROWID FROM RBRANCH RBRANCH3 WHERE BRANCH_CITY > 'PUNE') GROUP BY BRANCH_CITY
Plan6	SELECT /*+ USE_CONCAT */ BRANCH_CITY, COUNT(*) FROM RBRANCH WHERE BRANCH_CITY < 'PUNE' OR BRANCH_CITY > 'PUNE' GROUP BY BRANCH_CITY	**Plan15**	SELECT /*+ INDEX(RBRANCH1) */ BRANCH_CITY, COUNT(*) FROM RBRANCH RBRANCH1 WHERE RBRANCH1.ROWID IN (SELECT RBRANCH2.ROWID FROM RBRANCH RBRANCH2 WHERE BRANCH_CITY < 'PUNE' UNION SELECT RBRANCH3.ROWID FROM RBRANCH RBRANCH3 WHERE BRANCH_CITY > 'PUNE') GROUP BY BRANCH_CITY
Plan7	SELECT /*+ ALL_ROWS */ BRANCH_CITY, COUNT(*) FROM RBRANCH RBRANCH1 WHERE RBRANCH1.ROWID IN (SELECT RBRANCH2.ROWID FROM RBRANCH RBRANCH2 WHERE BRANCH_CITY < 'PUNE' UNION SELECT RBRANCH3.ROWID FROM RBRANCH RBRANCH3 WHERE BRANCH_CITY > 'PUNE') GROUP BY BRANCH_CITY		
Plan8	SELECT /*+ FIRST_ROWS(30) */ BRANCH_CITY,COUNT(*) FROM RBRANCH RBRANCH1 WHERE RBRANCH1.ROWID IN (SELECT RBRANCH2.ROWID FROM RBRANCH RBRANCH2		

	WHERE BRANCH_CITY < 'PUNE' UNION SELECT RBRANCH3.ROWID FROM RBRANCH RBRANCH3 WHERE BRANCH_CITY > 'PUNE') GROUP BY BRANCH_CITY		
Plan9	SELECT /*+ FULL(RBRANCH1) */ BRANCH_CITY, COUNT(*) FROM RBRANCH RBRANCH1 WHERE RBRANCH1.ROWID IN (SELECT RBRANCH2.ROWID FROM RBRANCH RBRANCH2 WHERE BRANCH_CITY < 'PUNE' UNION SELECT RBRANCH3.ROWID FROM RBRANCH RBRANCH3 WHERE BRANCH_CITY > 'PUNE') GROUP BY BRANCH_CITY		

Appendix 2: Queries Processing Used for Testing for OODB

Query Plans	**Query Statement**	**Query Plans**	**Query Statement**
Plan1	SELECT /*+ INDEX(BRANCH1)*/ CLIENT.BRANCHDETAIL.BRANCH_CITY ,COUNT(*) FROM BRANCH1 CLIENT GROUP BY CLIENT.BRANCHDETAIL.BRANCH_CITY HAVING CLIENT.BRANCHDETAIL.BRANCH_CITY != 'PUNE'	**Plan5**	SELECT /*+ INDEX_DESC(CLIENT) */ CLIENT.BRANCHDETAIL.BRANCH_CITY, COUNT(*) FROM BRANCH1 CLIENT WHERE CLIENT.BRANCHDETAIL.BRANCH_CITY != 'PUNE' GROUP BY CLIENT.BRANCHDETAIL.BRANCH_CITY
Plan2	SELECT /*+ ALL_ROWS */ CLIENT.BRANCHDETAIL.BRANCH_CITY, COUNT(*) FROM BRANCH1 CLIENT WHERE CLIENT.BRANCHDETAIL.BRANCH_CITY != 'PUNE‘ GROUP BY CLIENT.BRANCHDETAIL.BRANCH_CITY	**Plan6**	SELECT /*+ INDEX_COMBINE(CLIENT) */ CLIENT.BRANCHDETAIL.BRANCH_CITY, COUNT(*) FROM BRANCH1 CLIENT WHERE CLIENT.BRANCHDETAIL.BRANCH_CITY != 'PUNE' GROUP BY CLIENT.BRANCHDETAIL.BRANCH_CITY
Plan3	SELECT /*+ INDEX_FFS(CLIENT) */ CLIENT.BRANCHDETAIL.BRANCH_CITY,COUNT(*)	**Plan7**	SELECT CLIENT.BRANCHDETAIL.BRANCH_CITY, COUNT(*) FROM BRANCH1 CLIENT

	FROM BRANCH1 CLIENT WHERE CLIENT.BRANCHDETAIL.BRANCH_CITY != 'PUNE' GROUP BY CLIENT.BRANCHDETAIL.BRANCH_CITY		
Plan4	SELECT /*+ INDEX(CLIENT) */ CLIENT.BRANCHDETAIL.BRANCH_CITY, COUNT(*) FROM BRANCH1 CLIENT WHERE CLIENT.BRANCHDETAIL.BRANCH_CITY != 'PUNE' GROUP BY CLIENT.BRANCHDETAIL.BRANCH_CITY		