An Automated Index Selection Tool for Oracle7: MAESTRO 7

Asuman Dogac Aysenur Yerdekalmazer Erisik Alper Ikinci

TUBITAK Software Research and Development Center
Middle East Technical University
06531 Ankara Turkey
e-mail:asuman@vm.cc.metu.edu.tr

ABSTRACT

The index selection problem is NP-Complete and therefore requires heuristic to be used. Heuristic must be based on the query optimizer principles because the indices decided should be usable by the query optimizer. It should be noted that since the query optimization principles of commercial DBMSs vary to a great extent, it is infeasible to implement a general purpose index selection tool.

We present a tool MAESTRO 7 (METU Automated indEx Selection Tool foR Oracle7) that assists the database administrator in designing an index configuration for a given database application for Oracle7. It decides on the complete set of indices both primary and secondary by considering the index maintenance costs. MAESTRO 7 automatically derives valid SQL statements and their usage statistics during regular database sessions through SQL Trace Facility. It classifies the SQL statements that will produce the same execution plan and accumulates their weights. MAESTRO 7 also decides on plausible columns for indexing at the beginning, before the more time consuming processing on the queries start, to improve performance of the tool. The selection of primary indices cannot be done independently for each table for multi table queries. Therefore in choosing primary indices, all combinations of plausible primary indices are considered. The benefit of using the indices are calculated by obtaining the cost of executing the queries from the query optimizer. Although this tool is specific to Oracle7, a similar tool for any DBMS that has a cost based optimizer can be developed very easily following the ideas presented in this paper.

I. Introduction

Indices provide fast access to the database through the column on which the index is defined. However the index maintenance cost is considerable. When there are many insertions and deletions to a relation, the index may degenerate as in the case of hash tables and needs reorganization which introduces a reorganization cost. B+ trees or linear hash tables do not degenerate, but if there are many updates, index maintenance cost becomes considerable. Therefore, the decision whether to create an index on a certain column in a given relation requires the comparison of the costs of using and maintaining an index versus the cost of reading and updating the unindexed relation sequentially. It should also be noted that if there is more than one index on the relation, insertions and deletions require all the index structures to be updated. This cost needs to be taken into consideration too.

Another important decision related with the choice of indices is to decide on the primary (clustering) index.

An index is called primary if the relation for which the index is defined is physically clustered according to the index column values. The clustering property can greatly reduce the access cost, unfortunately only one column of a relation can have clustering property, since clustering requires a specific physical ordering of records in the file. With a primary index, no block is fetched more than once when tuples with consecutive values of the indexed column are retrieved.

A straightforward approach to the index selection problem would be to evaluate the total cost for each possible choice of columns to be indexed, and then select that set of columns which gives the smallest cost. With n columns in a relation, there are 2^n possible choices of index set. It is clear that for moderate n, the cost evaluation procedure becomes very expensive, and for large n prohibitive. It has been proved in [C78] that index selection problem is inherently difficult. Therefore, it is appropriate to use heuristic methods that significantly prune down the search space and that work towards obtaining a near-optimal solution [HC 76].

Another important point is the following: Indices defined and the clustering decided for a database are then used by the query optimizer of a DBMS to optimize the execution of the queries. Therefore the design must provide for the indices usable by the optimizer. The behavior of query optimizers of the existing commercial DBMSs vary to a large extent:

- Systems like Informix and SQL/DS have optimizers that are cost based and ORACLE 6's optimizer is rule based. On the other hand, Oracle7 have both the rule based and cost based optimizers.

- The number of indices used per table is different. The optimizers of SQL/DS and Informix use only one index per table per query for conjunction predicates. Yet ORACLE 6 and Oracle 7 use up to five indices per table per query.

- The type of indices available are different, certain DBMS's provide for only B+tree indices like Informix, and SQL/DS ,yet Oracle7 and Ingres support both B+tree indices and hash indices. So for systems like Oracle7 and Ingres system it is not only necessary to decide on whether to index a column, but also the type of the index.

- The available join methods are different. Oracle7 and SQL/DS support both index join and sort/merge join, yet Informix supports only index join.

As can be seen from the above discussion, physical design is dependent on the optimizer principles, and hence it is not possible to give a set of general rules for all DBMSs, rather index selection tool should be specific to a product.

The most often used criterion in evaluating the physical design is the number of disk accesses which directly effects the response time. However space utilization, that is the amount of storage space used by the database files and their access path structures (indices), and the transaction throughput, that is the average number of transactions that can be processed per minute by the database system can also be used as the evaluation criteria. It should be noted that index storage cost is not negligible, for example an SQL/DS index consumes from 5 to 20 percent of the space used by the table it indexes.

There are two problems to be solved related with the index selection problem: one is to find out a descriptive workload for the system and second is to decide on the indices considering this workload and the system statistics. The current commercial systems provide such facilities like the trace facility which when enabled provides execution statistics for all SQL statements executed in a user session or in an instance. These systems also provide commands which when called give the execution plan and the associated cost.

By using these facilities provided by Oracle7, we have implemented a tool that assists the database administrator in deciding both primary and secondary index configuration for Oracle7. We have excluded combined (concatenated) indices from consideration. Investigating all possible combined indices introduces combinational explosion and there is no heuristic developed yet to handle this problem.

Although this tool is specific to Oracle7, a similar tool for any DBMS that has a cost based optimizer can be developed very easily following the ideas presented in this paper.

Some of the terminology used by ORACLE is different from the ones used in the database literature. In this paper, we will use database literature terminology. Equivalent ORACLE terminology is provided in the following:

Database Literature	ORACLE
Sequential Scan of a Table	Full Table Scan
Hash Index	Hash Cluster
Primary B+tree Index	Index Cluster
Secondary B+tree Index	Index

II. Previous Work

A manual physical database design method is suggested in [DD 85a, DD85b].

The method makes the following assumptions:
-the indices are structured as B+trees
-joins are performed according to the nested loops index method
-the choice of primary indices (or keys) has already been made in the logical design phase.

The indices are then compared on the basis of costs for all operation executions; those which are less efficient than others are eliminated and the secondary indices are thus decided. This method apart from being manual omits several essential details of index selection.

In [BPS 90] a heuristic algorithm based on some properties of the cost function for the selection of secondary indices in relational databases is described. The indices have the B+tree structure. An access cost function is defined that includes the index maintenance cost. It is proved that when an index is added to a given set and the cost function value increases, then it also increases when the index is added to any other set. In this paper it is assumed that any number of indices can be used per table. This assumption violates the optimizer principles of DBMSs like SQL/DS and Informix, since both of them use only one index per table for conjunction predicates.

In [WWS 81] separability property of the join techniques are discussed. This property reduces the index selection problem to finding a locally optimal index configuration for each relation. It is proved that a particular join index method described in the paper have the separability property. For this join index technique, it is possible to include the coupling effects directly into the cost formulas under the constraint that at least one relation participating to the join must have indices for all restriction columns. Although the sort merge join technique have the separability property coupling factors do not appear in the related cost formulas due to the nature of sort-merge join technique.

Coupling effect can be explained as follows: When restrictions are applied to a relation, say R1, before it participates to a join operation with a relation R2 than some tuples of R2 will not be accessed when the binary join index method described in the same paper is used. In other words, if a tuple of relation R1 does not satisfy the restriction predicate for R1, the corresponding tuples of R2 that have the same join column values are not accessed. It is clear that when a different index join technique is used or the restrictions are delayed after the join operation, there is no coupling effect. Therefore the separability property are applicable only in limited cases. Also the method described in this paper requires more than one index to be used per query per table.

In [FST 88], an experimental physical design tool, DBDSGN, that finds efficient solutions to the index-selection problem for System R is described. RDT is the commercially available version of DBDSGN for SQL/DS. In System R, indices are structured only as B+trees, and two methods namely the nested loops index and the sort merge methods are available for join processing. In System R exactly one index is used for each table in a statement. DBDSGN uses information supplied by the System R optimizer both to determine which columns might be worth indexing and to obtain estimates of the cost of executing statements in different index configurations. EXPLAIN REFERENCE option of System R is used in identifying the plausible columns for each query and only the columns that are plausible for indexing enter into the design process. Statistics on tables and columns to be used in cost evaluation are either provided by the designer or extracted from the database catalogs. The atomic costs are evaluated as the next step. Configurations with at most one index per relation are called atomic configurations, and their costs are called atomic costs. Since System R uses only one index per table, atomic costs are valid. The costs for all other configurations are computed from them. This step may be followed by elimination of some of the indices by using the heuristics given in [FST 88]. In the final step a controlled search of the survivor indices is performed to find good solutions to the index-selection problem. Index storage limit is taken into account at this step. Note that the technique described in [FST 88] is based on the atomic costs. Therefore it cannot be used for systems with query optimizers using more than one index per table per query like Oracle7 because in such a case atomic costs are no longer valid.

In [RS 91] the framework described for automating physical database design produces both physical data structures and physical query execution plans. However, in order to use this framework with a specific DBMS the query execution plans produced by the DBMS and the query execution plans produced by framework must match.

In [FON 92], a tool for designing an index configuration for relational databases is described and a prototype for ORACLE 6 is implemented. The tool derives its input automatically during regular database usage by using trace facility of ORACLE 6. The optimizer is presented with a query language statement and an index set to choose from for processing this statement. The optimizer exports the index set it would choose. Since ORACLE 6's optimizer is rule based, there is no way of obtaining the execution cost from the optimizer. Therefore a cost is estimated from the execution statistics for the query. By using this information, the tool constructs an index benefit graph (IBG) for each query which contains all index sets the optimizer would choose for this query. Retaining IBGs for each query would allow to find optimal index configuration.

Notice that in this work only the secondary indices are considered and more importantly the index maintenance costs are not taken into account. Also since a rule based optimizer is used, the execution costs are estimated rather than obtaining it from the optimizer.

III. Oracle7

A. Query Optimizer Characteristics

To choose an execution plan for a SQL statement, the Oracle7 optimizer uses either the rule based approach or the cost based approach. The cost based approach is used when the statistics of tables are provided.

The cost based approach uses statistics to estimate the cost of each execution plan. These statistics quantify the data distribution and storage characteristics of tables, columns, and indexes, and are generated using the ANALYZE command. The optimizer estimates how much I/O, CPU time, and memory is required to execute a SQL statement using a particular execution plan by using these statistics.

B. EXPLAIN PLAN statement

The EXPLAIN PLAN command inserts a row describing each step of the execution plan of SELECT, INSERT, UPDATE, DELETE statements into a specified table. A statement's execution plan is the sequence that ORACLE performs to execute the statement. If cost based optimization is used this command also determines the cost of the executing the statement.

C. SQL Trace Facility

The SQL trace facility provides performance information on individual SQL statements. The SQL trace facility can be enabled for a session or for an instance. When the SQL trace facility is enabled, execution statistics for all SQL statements executed in a user session or in an instance, are placed into a trace file. The trace files should be translated into readable form by running the TKPROF program.

IV. Implementation of MAESTRO 7

The basic steps involved in implementing the MAESTRO 7 are as follows:

1. Collecting the SQL statements and their frequency through the SQL trace facility.

2. Deciding on the columns plausible for indexing and their primary index type.

3. Deciding on the primary indices.

4. Deciding on the secondary indices.

In the following these steps are described in detail:

1. Collecting the SQL statements and their frequency through the SQL trace facility.

The SQL trace facility is turned on for a specified period of time on the database whose indices are to be decided. The purpose of trace files in ORACLE is to help with the debugging. Therefore it contains extra information like the ones created by the database kernel. Through a parser we process the trace files to extract valid SQL statements and their usage statistics that are collected during this period. Note that there may be more than one occurrence of an SQL statement in the trace file, with the same or different constants. Depending on the nature of the operator used (like IN, LIKE, BETWEEN, =, etc.), these queries are classified. As an example, consider the following queries:

```
SELECT E.EMPNO, D.DEPTNO
FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO
AND E.SAL = 10000
AND E.ENAME LIKE 'A%'

SELECT E.EMPNO, D.DEPTNO
FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO
AND E.SAL = 50000
AND E.ENAME LIKE 'M%'
```

The general structures of these two queries are the same. The only difference is the value of the constants. Therefore these two queries are considered as a single query, adding their workloads. This processing reduces the number of Index Benefit Graphs (IBG) to be formed later for each query.

However note that this classification is valid only for statements that have "=", "LIKE" and "IN" SQL operators in their WHERE clauses.For "LIKE" operator, it is important to know the position of the wildcard characters in the string. Because the position of the wildcard characters determines whether the query is a range query, equality or sequential scan. The characters in the string are eliminated preserving the position of the wildcard characters. In evaluating the range scans, the value of the constant determines the use of an index. The constant are not eliminated in such cases.

The statements whose constants are eliminated, are compared with the ones previously processed. If the same statement has occurred before, then their workloads are added.

2. Deciding on the columns plausible for indexing and their primary index type.

We define the columns that are candidate to an index structure as plausible columns. The reasoning behind defining plausible columns is the following: There are times that Oracle7 does not use an existing index in accessing a table. If a column is modified in a WHERE clause, the index on this column is not used. Furthermore, an index can not be used on columns that are not modified in a WHERE clause for the following cases:

i.The columns with NOT IN, NOT LIKE, IS NOT NULL, IS NULL comparison operators.

ii.The columns in LIKE clause with a comparison string having a wildcard at the beginning.

iii.The columns of the same table appearing on both sides of a comparison operator are not plausible since sequential scan must be performed in such a case.

iv. In correlated subqueries, it is necessary to execute the subquery for every record of the main query. Therefore the index on the column in main query that is compared with the result of subquery is useless.

In order to reduce the execution time of the MAESTRO 7 considerably, the unplausible columns are eliminated before the start of the more time consuming processing on the queries.

All the plausible columns are candidates for primary indices. Before proceeding any further, the index types are determined.

The columns indicated below are plausible for a hash index.

```
<column_name> =      <constant> | <subquery>
<column_name> LIKE <constant>
<column_name> IN     <list_of_values> | <subquery>
```

The rest of the plausible columns and all equi-join columns, are plausible for B+tree index.

The reasoning for this heuristic is the following: A hash index is faster than a B+tree index on single record retrievals. However, Oracle7's hash index is static hash index, and therefore performance degradation is substantial if the table to be indexed is not static because of overflown records. For columns plausible for a hash index it is necessary to check whether the table is static. This is achieved through information obtained from the trace files: if the number of rows obtained by taking the difference of inserted rows and deleted rows exceed the total number of rows by a certain percentage, then the table is considered to be unstatic. For unstatic tables, columns are eliminated from the list of plausible columns for hashing.

3. Deciding on the primary indices.

It should be noted that a table can have at most one primary index since the records in a table can be physically organized only in one way.

After the columns plausible for primary indices and their types are determined, the primary indices are decided as follows:

For each query, for all the columns plausible for primary index in every table involved in the query, all the combinations of columns are created by taking a column from a table at a time and by considering the type of plausibility.

The reasoning behind considering all the combinations is the following: The selection of primary indices cannot be done independently for each table for multi table queries.

As an example to clarify the point, consider the following query:

```
SELECT E.ENAME
FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO
AND D.DEPTNO = 1010
AND E.SAL BETWEEN 2000 AND 3000
```

In this query E.DEPTNO, D.DEPTNO and E.SAL are candidates for primary B+Tree index and D.DEPTNO is candidate for both a primary hash index and a primary B+tree index.

The combinations presented to the optimizer are:

E.DEPTNO (B+TREE)	D.DEPTNO (B+TREE)
E.DEPTNO (B+TREE)	D.DEPTNO (HASH)
E.SAL (B+TREE)	D.DEPTNO (B+TREE)
E.SAL (B+TREE)	D.DEPTNO (HASH)

These are all useful indices: The primary B+ tree indices on DEPTNO in EMP and DEPTNO in DEPT tables may reduce the cost of join drastically since no block is fetched more than once when tuples with consecutive values of the indexed column are retrieved and also a primary B+tree index is useful on SAL. Furthermore the best index for executing D.DEPTNO= 1010 by itself is the primary hash index. However, if DEPTNO in EMP has a primary B+tree index, SAL can not have primary B+tree index. Also, the primary B+tree index on DEPTNO in EMP is more useful if there is a primary B+tree index on DEPTNO in DEPT. Note that the best combination is dependent on the system statistics and therefore all of these combinations must be given to the optimizer. The optimizer thus has the chance of finding the best combination of indices by considering the system statistics for the given query.

Note that for a different query the best set of indices on the same tables might be quite different. In order to preserve the benefit of an index set for the queries Index Benefit Graphs (IBGs) are formed.

For each query and for each combination an IBG is formed. IBG will be explained through an example query. The index set in the combination D.DEPTNO (Primary B+tree) and E.SAL (Primary B+ tree) are created and presented to the optimizer. The optimizer's choice of the index which is D.DEPTNO and the returned execution cost stored in node 1 as shown in Figure 1 and this index is dropped. The costs at each node of IBG is obtained by using EXPLAIN PLAN statement. Note that since EXPLAIN PLAN statement can only provide optimizer information based on the currently existing index configuration, the necessary indices are created on the tables. The cost based optimizer's EXPLAIN PLAN statement also needs the statistics on the tables and clusters. For this purpose the ANALYZE CLUSTER and ANALYZE INDEX statements must be executed for the related tables and columns respectively. Then the optimizer is presented with E.SAL and the second node of the IBG is formed. Finally the index on E.SAL is dropped and executing this query without any index is obtained from EXPLAIN PLAN statement and placed as the node 3 of the IBG.

```
          |      { D.DEPTNO, E.SAL}
[1]  { D.DEPTNO } (Cost: 22)
          |
          |    {E.SAL}
[2]  { E.SAL}  (Cost: 49)
          |
          |  {}
[3]     {}  (Cost: 110)
```

Figure 1. Index Benefit Graph for a combination for the example query

At each node of the IBG, the atomic saving of an index is calculated with the following formula:

Atomic Saving of an index = (Sequential Scan cost of statement without any cluster - cost returned by the EXPLAIN PLAN) / number of indices used.

The number of indices used is also obtained from the EXPLAIN PLAN statement.

The assumption over here is that the savings of an index set can be equally credited to each individual index [FON 92].

A table, called PIET(Primary Index Elimination Table) is formed as shown in Table 1 for each database table, consisting of three parts: In the first part each row and column position contains the maximum atomic saving, as, of an index for a given statement, s. Second part contains the sequential scan costs of a table with a primary index, ssc_wp. Third part contains index maintanence cost of UPDATE and DELETE statements.

		P A R T 1			P A R T 2		P A R T 3	
		s1	s2	..	s5	...	u1 ...	d1 ...
TABLE 1	col1	as11	-	...	ssc_wp15	...	imc11	imc11...
	col2	as21	-	...	ssc_wp25	...	-	imc21...
	.	.			.		.	
	.	.			.		.	
	.	.			.		.	
.								
.								
.								

Table 1. Primary Index Elimination Table, PIET

In part 1 as_{ij} denotes the atomic saving of using an index on col_i for statement s_j. All the statements with a WHERE clause and the INSERT statements are considered over here. For the INSERT statement, the atomic saving is obtained by calling EXPLAIN PLAN once with an index and once without an index and taking the cost difference. Clearly the insertions to a table with a primary index is faster.

In part 2, s_j's are SQL queries performing sequential scans like: SELECT * FROM DEPT. The reason that these type of statements are considered is the following: when there is a primary index on a table, the table may get larger because Oracle7 stores additional 19 bytes for each primary index column (cluster key in ORACLE's terminology). However it stores every primary index column value only once. Thus depending on the selectivity of the primary index column, the database table may get smaller or larger. Depending on this size the following cost may be positive or negative:

ssc_wp_{ij} = Sequential Scan cost of the statement s_j without any cluster - cost of the statement s_j when table clustered on col_i

In part 3, the u_j's and d_j's are the UPDATE and DELETE statements. When a table with an index on its col_i is updated on col_i, there is an index maintenance cost denoted by imc_{ij}. There is a similar index maintenance cost for DELETE statements.

Index maintenance cost applies to B+tree indices; hash indices do not introduce maintenance cost. They have reorganization cost, but since hash index is only for static tables there is no need to consider the maintenance cost.

However, for DELETE and UPDATE statements, the index maintenance cost can not be obtained from the EXPLAIN PLAN statement because if the index on the table is used in accessing the rows, the cost returned is index access cost. Once the records are brought into memory, they are updated there and since ORACLE's End of Transaction processing policy is NOT FORCE, the records are not forced to disk. Thus it is not possible to obtain the index maintenance cost.

The following heuristics are used to estimate the index maintenance cost for UPDATE and DELETE statements: If the index is used in retrieving the rows then the related index maintenance cost is writing the modified block(s). However because of ORACLE's fast commit policy this cost is neglected. Fast commit writes all the updates into a log rather than into the database. If the index is not used in accessing the rows, the index maintenance cost involves both retrieving the index into memory and writing the index and the leaf block(s) back. Thus, it is necessary to obtain the cost of reading the index blocks into memory to calculate the index maintenance cost. Oracle7 cost optimizer estimates logical execution cost not the physical execution cost for each execution plan through EXPLAIN statement. Consequently, all the costs that will be used in index elimination must be logical execution costs rather than physical read costs. The following formula is used in evaluating the logical multi block read cost.

multi_block_read_cost = ceil(sequential scan cost / ceil(space used to store table in blocks / db_file_multi_block_read_count))
where 'db_file_multi_block_read_cost' is a parameter, in the configuration file of Oracle7, and is used to set the number of blocks that can be read at a time.

Note that for the index maintenance cost of primary indices, if the table is updated through the primary index the associated cost is writing the modified blocks which have been neglected due to fast commit. If the table is updated on a column different than the primary index column, there is no index maintenance cost for that column but it should be considered for other plausible columns.

The third part of PIET stores the index maintenance cost of UPDATE and DELETE statements. It should be noted that there is no index maintanence cost for INSERT statements because if there is a primary index, it is used by INSERT statements.

The index maintenance cost is calculated as described above. imc_{ij} is inserted into this table as a negative benefit. Note that for UPDATE statements this negative benefit is for particular columns, since it effects the indices only on columns in SET clauses. But for DELETE statements this negative benefit is valid for all plausible columns because it effects all the indices on the table.

When PIET is considered as an n*m matrix, the overall benefit of col_i;

$$OB_i(col_i) = \sum_{j=1}^{m} [PIET(i,j)*w_j] \quad \text{where } w_j \text{ is the workload of the statement } s_j.$$

The OB_i's are sorted in the descending order for each database table. Note that this ordering ranks the columns of a database table from most profitable to least profitable for indexing. The most profitable primary index is selected for each table.

After selecting most profitable primary indices they are created and dropped from the list of plausible indices, since they are no longer candidate for secondary indices.

4. Deciding on the secondary indices.

Once the primary indices are decided, they are eliminated from the plausible column set and the rest of the plausible columns become candidates for the secondary indices.

For each query indices for all plausible columns are created and ANALYZE INDEX statement is executed. Then the IBG's are formed to obtain the atomic savings of the indices.

A table, called SIET(Secondary Index Elimination Table) is formed as shown in Table 2 for each database table, consisting of three parts: In the first part each row and column position contains the maximum atomic saving, as, of an index for a given statement, s. In the second part, for UPDATE statements, the index maintenance costs, imc, are recorded. The third part contains the index maintenance cost of INSERT and DELETE statements.

		PART 1			PART 2		PART 3	
		s1	s2	...	u3	...	s4	...
TABLE1	col1	as11	-	...	imc13	...	imc14	...
	col2	as21	as22	...	-	...	imc24	...
	.	.	.		.		.	
	.	.	.		.		.	
	.	.	.		.		.	
.								
.								
.								

Table 2. Secondary Index Elimination Table, SIET

In part 1, as_{ij} denotes the atomic saving of using a secondary index on col_i for statement s_j. All the statements with a WHERE clause are considered over here. Note that there is no saving for an INSERT statement because a secondary index is not used in accessing the table when executing the INSERT command as opposed to the primary index case. However there is an index maintenance cost which is considered in Part 3 of SIET as shown in Table 2. It should be noted that since a secondary index does not change the size of a table, it does not effect the sequential scan cost.

In part 2 and part 3 the s_j's are the UPDATE, INSERT, DELETE statements. When a table with an index on its col_i is updated on col_i, there is an index maintenance cost denoted by imc_{ij}. There is a similar index maintenance cost for INSERT or DELETE statements.

Since for INSERT, DELETE and UPDATE statements, the index maintenance cost can not be obtained from the EXPLAIN PLAN statement it is calculated by using the heuristic described in step 3.

Note that, in secondary index elimination the index maintanence cost should be taken into account for the INSERT statements. Because secondary indices can not be used to execute INSERT statements.

The second part of SIET stores the index maintenance cost of UPDATE statements. The index maintenance cost is calculated as described above. imc_{ij} is inserted into this table as a negative benefit. Note that this negative benefit is also for particular columns, since it effects the indices only on columns in the SET clauses.

The third part of SIET contains the index maintenance cost for INSERT and DELETE statements. This negative benefit is valid for all plausible columns, since these statements effect all the indices on the table.

For each table in the database; When SIET is considered as an n*m matrix, the overall benefit of col_i;

$$OB_i(col_i) = \sum_{j=1}^{m} [SIET(i,j)*w_j]$$ where w_j is the workload of the statement s_j.

The OB_i's are sorted in the descending order. The columns with negative benefits are eliminated. This ordering ranks the columns of a database table from the most profitable to the least profitable and gives the selected set of secondary indices. This is a list of indices from most profitable to the least for each table. There is also another list which sort secondary indices according to the ratio calculated by dividing overall benefits into storage consumed by the indices.

By considering the space limitations the DBA may eliminate some of these indices.

V. Conclusions

This paper presents an index selection tool for Oracle7, called MAESTRO 7. The research is influenced by the previous work in this area, especially from [FON92] and [FST88]. The contribution of the paper are as follows:

1. It decides on the complete set of indices both primary and secondary by considering the index maintenance costs. Note that it is impossible to neglect index maintenance cost for real life applications.

2. Our tool classifies the SQL statements that will produce the same execution plan and accumulates their weights. Also by noting that ORACLE does not use the existing index on certain columns, we have processed each SQL statement obtained from the trace file to eliminate such columns. In other words we have chosen the columns plausible for indexing from the beginning to improve the performance of the tool. These processes reduced both the number of IBGs and their complexity.

3. The selection of primary indices cannot be done independently for each table for multi table queries. Therefore in choosing primary indices, all combinations of plausible primary indices are considered.

4. Although this tool is specific to Oracle7, a similar tool for any DBMS that has a cost based optimizer can be developed very easily following the ideas presented in this paper.

5. It is a real product for a commercial DBMS.

6. The performance of the index sets proposed by MAESTRO 7 have been verified by testing it against to a sample database which consist of 25 MB database tables and approximately 2700 queries. The execution time to run the example is approximately 7 hours on the HP 9000/817. The set of indices proposed by MAESTRO 7 turned out to be the optimum set as verified through manual calculations.

References

[BPS 90] E. Barcucci, R. Pinzani, and R. Sprugnoli, "Optimal Selection of Secondary Indexes," IEEE Trans. on Software Eng., vol. SE-16, no. 1, January 1990.

[BCPR 91] E. Barcucci, R. Chiuderi, R. Pinzani, and E. Rodella, "Optimal Selection of Secondary Indexes in Relational Databases," in Proc. 6th Int. Sym. on Computer and Information Sciences, Antalya, 1991.

[C 78] D. Comer,"The difficulty of optimum index selection," ACM Trans. on Database Syst., vol. 3, no.4, Dec. 1978.

[DD 85a] V. DeAntonellis and A. DiLeva, "Dataid-1: A database design methodology," Inform. Syst., vol. 10, no. 2, 1985.

[DD 85b] V. DeAntonellis and A. DiLeva, "A case study of database design using the Dataid approach," Inform. Syst., vol. 10, no. 3, 1985.

[FON 92] M. Frank, E. Omiecinski, S. Navathe, "Adaptive and Automated Index Selection in RDBMS", Proc. of Extending Database Technology, March 1992.

[FST 88] S. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical database design for relational databases," ACM Trans. on Database Syst., vol. 13, no.1, March 1988.

[HC 85] M. Hammer and A. Chan, "Index selection in a self- adaptive database management system," in ACM-SIGMOD Proc. of Int. Conf. on Management of Data, June 1976.

[RS 91] S. Rozen and S. Dennis, "A Framework for Automating Physical Database Design", in Proc. 11th Conf. Very Large Databases, Barcelona, Sept. 1991.

[WWS 81] K-Y Whang, G.Wiederhold, and D.Sagalowicz "Separability- An approach to physical database design," in Proc. 7th Conf. Very Large Databases, Cannes, Sept.

1981.