Information Sciences 179 (2009) 2263–2272

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

An index selection method without repeated optimizer estimations

Kuo-Fong Kao [a,*], I-En Liao [b]

[a] Dept. of Information Networking Technology, Hsiuping Institute of Technology, No. 11, Gongye Road, Dali City, Taichung County, Taiwan
[b] Dept. of Computer Science and Engineering, National Chung Hsing University, No. 250, Kuo Kuang Road, Taichung, Taiwan

ARTICLE INFO

Article history:
Received 17 January 2007
Received in revised form 26 December 2008
Accepted 15 February 2009

Keywords:
Database management system (DBMS)
Index selection problem (ISP)
Performance tuning
Configuration recommendation
Optimizer

ABSTRACT

The index selection problem (ISP) concerns the selection of an appropriate index set to minimize the total cost for a given workload containing read and update queries. Since the ISP has been proven to be an NP-hard problem, most studies focus on heuristic algorithms to obtain approximate solutions. However, even approximate algorithms still consume a large amount of computing time and disk space because these systems must record all query statements and frequently request from the database optimizers the cost estimation of each query in each considered index. This study proposes a novel algorithm without repeated optimizer estimations. When a query is delivered to a database system, the optimizer evaluates the costs of various query plans and chooses an access path for the query. The information from the evaluation stage is aggregated and recorded with limited space. The proposed algorithm can recommend indexes according to the readily available information without querying the optimizer again. The proposed algorithm was tested in a PostgreSQL database system using TPC-H data. Experimental results show the effectiveness of the proposed approach.

1. Introduction

Indexing is an important method for improving the read performance of a database. However creating an index is not necessarily good for the performance of a query. The consistency maintenance between the original data file and its index file increases the response time of a write query. The index selection problem (ISP) involves choosing a proper index set to minimize the total cost for a given workload containing read and write queries. The ISP has been proven to be an NP-hard problem [13].

As a result of the difficulty of solving the ISP, most studies [2,3,7,14,21,27] propose heuristic algorithms to obtain an approximate solution whose performance resembles the performance of the exact optimal solution. An approximate solution is usually sufficient for general users. However even approximate algorithms still consume a large amount of computing time and disk space, because such systems must record complete query texts and frequently request the cost of each query from the database optimizer for each index considered. To reduce the burden of a database optimizer, some systems create a virtual index module to estimate the cost of each query for each index configuration considered. Fig. 1 illustrates the workflow of a traditional ISP solution.

The solution to the traditional ISP problem is difficult to be applied to a DBMS system without virtual index. The constraint motivated us to initiate this study. The key idea is to effectively exploit information when queries are executed. When a query is delivered to a database system, the optimizer evaluates the cost of various query plans and chooses an access path for the query. The table scan is the basic access path for considered query plans. If indexes exist, and the optimizer finds that

* Corresponding author. Tel.: +886 4 24961100x3399; fax: +886 4 24961187.
E-mail addresses: kfkao@mail.hit.edu.tw (K.-F. Kao), ieliao@nchu.edu.tw (I-En Liao).

0020-0255/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2009.02.009

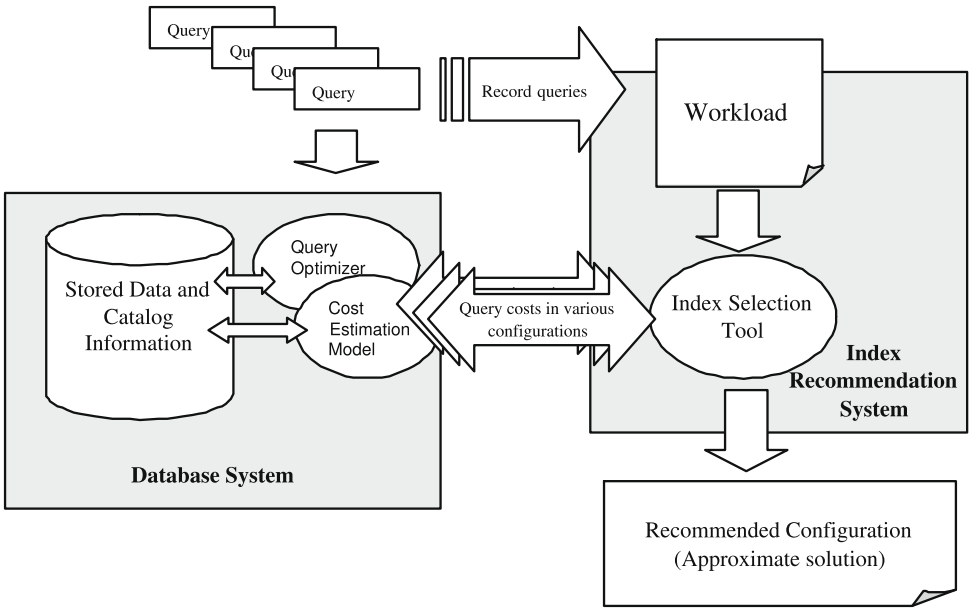

Fig. 1. Workflow of traditional ISP solution.

the cost of using those indexes is lower than that of the table scan, then the optimizer chooses these indexes as the access path for this query. The above evaluation action must be performed whenever a query arrives. Therefore, if an index selection tool can recommend indexes by only taking advantage of the extant information obtained from previous evaluations, then a virtual index module is not needed in the index tuning process. Fig. 2 illustrates the workflow of the proposed new ISP method.

Because the index selection tool can only use the extant information, we called the new ISP mechanism the information-limited ISP. The goal of this research is to propose an efficient solution to the information-limited ISP problem for DBMS systems without virtual index. In our design, these limited information are aggregated and stored in a small space. The index selection tool can quickly read these information and respond with the recommended index configuration without querying the optimizer again. The formal definition of the information limitation will be detailed in Section 3.3.

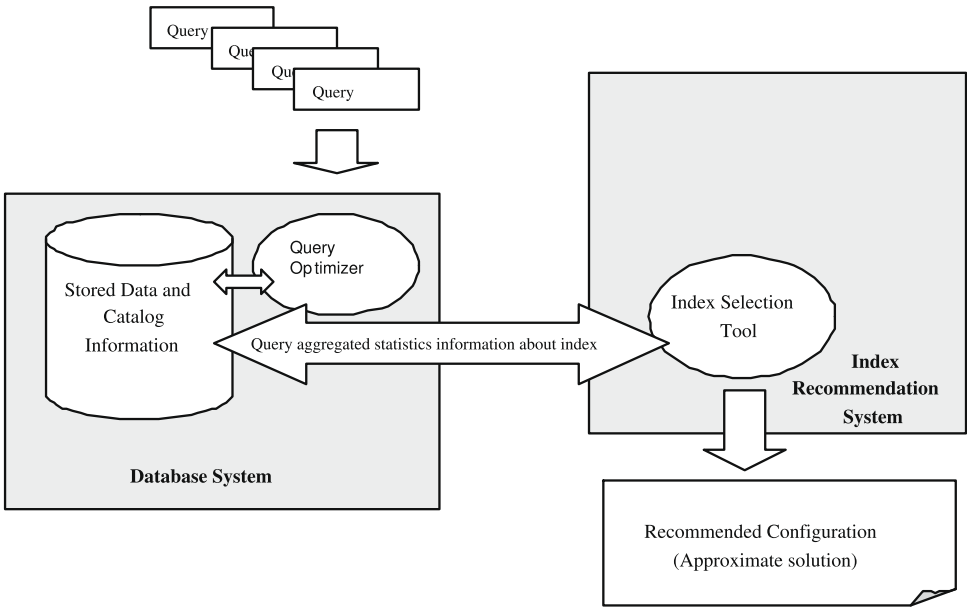

Fig. 2. Workflow of proposed ISP solution without repeated optimizer estimations.

The rest of this paper is organized as follows. Section 2 summarizes related work. Section 3 then explains the definition of ISP and the assumption of the proposed architecture. Sections 4 and 5 present the primary innovation of this mechanism. Section 4 describes how to evaluate index performance with limited information. Section 5 explains the design of tables used to store statistics and the practical algorithm that uses this information to evaluate the index. Section 6 presents the experiment testbed in PostgreSQL and the improved performance. Conclusions are finally drawn in Section 7, along with future research directions.

2. Related work

The index selection problem is an application of the classic 0-1 integer programming problem, which is a special case of the well-known integer knapsack problem [1,19,29]. The ISP has been studied since the early 1970s, and has been proven to be an NP-hard problem [13]. The proposed methods in this area can be roughly divided into knowledge-based approaches [14,27] and optimizer-based approaches [2,3,7,21]. The knowledge-based approach collects the knowledge of experts, and encodes them as rules which are used to generate a recommended index set. This approach suffers from being disconnected from the database optimizer. It is not certain whether the recommended index will be used by the optimizer. By contrast, the optimizer-based approach selects indexes based on cost estimations from the database optimizer. Hence, the optimizer-based approach does not have the drawback of knowledge-based approach; but the optimizer-based approach slowly performs for the repeated optimizer estimations of different index configurations.

The optimizer-based approach can compare the cost of complete enumeration of all possible indexes to find the optimal solution. The exhaustive search method is impractical in most cases, since the computational efforts grow exponentially with the number of candidate indexes. Finding the exact solution in a more efficient way led the researchers to propose a branch-and-bound algorithm [7]. However this method still works slowly for large numbers of indexes. In fact, most studies in the literature have focused on finding approximate solutions. Approximate algorithms mainly include the ADD and DROP methods [2,3,21]. In an ADD algorithm, the system starts from an empty set, and at each step adds the index most capable of reducing the cost function, stopping when no more cost-reducing indexes remain. The DROP algorithm starts with the set of all the possible indexes, and at each step it drops out the index that would cause the highest drop in cost function. Some other algorithms [4,12] use the hybrid of ADD and DROP methods to recommend an index configuration.

The approximate algorithms based on the optimizer-based approach have been developed on many real database systems. The DBDSGN system designed for IBM System R [15] is the most famous early implementation. It recommends an index configuration by considering the cost estimations from a "virtual index" module. Although it is an experimental database system, its architecture has become the basis for many subsequent systems. For example Microsoft proposed the index selection tool on the index tuning wizard of SQL Server 7.0 [9,10]. This tool adopts the System R architecture, and uses the "what-if" index creation module to provide the estimated cost of a query. The method can reduce the loading of a real database system. This system also considers multi-column index conditions. The algorithm was tested using workloads of the TPC-D schema, and was found to perform well. IBM also exploited the DB2 Advisor tool for the DB2 Universal Database system [24,30]. The DB2 Advisor uses the virtual index to estimate the cost of a query. To limit the number of virtual indexes, the system starts by analyzing the statement predicates and clauses to produce sets of columns that might be exploited. This approach is called the "Smart Column Enumeration for Index Scans" (SAEFIS) enumeration algorithm. Each virtual index from the SAEFIS algorithm has an associated benefit and size value. The DB2 Advisor recommends the best index according to the benefit-size ratio. Oracle also applied the Oracle Tuning Pack [25] to automatically recommend indexes, but did not disclose the index selection algorithm. In addition to the index selection tools on normal database systems, some studies focus on ISPs of special database systems, such as object-oriented DBMS [11], on-line analytical processing (OLAP) [18] and data warehouse systems [5,16]. The key idea of aforementioned algorithms is mainly on reducing the number of candidate configurations for the improvement of system performance. However, these algorithms all require repeated optimizer estimations. This motivated us to propose the new ISP solution under the constraint that repeated optimizer estimations are not allowed.

3. Problem definitions

This section will give a clear definition and assumptions of the ISP problems. The general definition of an ISP problem is first presented in Section 1. The ISP concerns the cost of a query. Hence, the cost model generally used in the ISP is described in Section 2. The final subsection will describe the focus of this study: the information-limited ISP which is caused by the no repeated optimizer estimation constraint.

3.1. Definition of index selection problem

A general ISP can be formulated as follows. A workload M contains m queries, and $q_i, 1 \leqslant i \leqslant m$, denotes the ith query in workload M. The candidate index set is defined as N containing n candidate indexes, and $a_j, 1 \leqslant j \leqslant n$, denotes the jth index in N. Each candidate index may be either built or not built. Once built, an index a_j requires a maintenance cost m_j for the workload M without considering whether the query adopts this index as an access path. The built status of all indexes is called the

index configuration. The set of all possible configurations is defined as P containing p ($p = 2^n$) configurations, and $c_k, 1 \leqslant k \leqslant p$, denotes the kth configuration in P. Each query q_i in configuration c_k accesses the database at a cost of $cost(q_i, c_k)$. The ISP attempts to find the configuration c^* with the minimum total cost from the finite number of possible configurations, which can be formulated as follows:

$$c^* = \min_k \left[\sum_{i \in M} cost(q_i, c_k) \right] \quad \text{for all } c_k \in P. \tag{1}$$

The problems with this formulation have been proven to be NP-hard problems [13]. To evaluate $cost(q_i, c_k)$ for making a configuration recommendation, the system would log the text of q_i in a query-text log file. This query-text log file is different from the log file for transaction processing in a database system.

3.2. Cost models of query

The cost function in Eq. (1) is critical for ISP, and there are a lot of researches which focus on formulating the cost model of a query [7,12,23]. In this research, we adopted the most general cost model for better usability. The cost model of a query can be divided into two cases depending on whether the query uses an index or not. If a query does not use an index, then the cost of the query includes the cost of scanning the table, computing the queried value, updating data and index files, and outputting the result. The cost model of a query that uses an index is more complex than that of a query without an index. A read query using an index generally contains the following operations: (1) The system accesses indexes corresponding to the attributes specified in the query to find the lists of the tuple IDs (TIDs); (2) The system then finds the intersection of the TID lists in memory to determine the TIDs of the tuples that satisfy the search conditions of the query; (3) Finally, the system retrieves the tuples from the table file in order to check the values of the specified attributes not having an index, and then produce the output. A write query using an index involves the above operations and the update operations of the index and data files.

For simplicity, the cost of a TID operation in memory is always ignored, and the number of disk I/O operations is used to represent the cost. Hence, the remained cost can be classified as I/O cost of read *ReadCst*(), I/O cost of index maintenance *MaintCst*(), and other constant cost *ConstCst*(). The *ReadCst*() depends on whether the query uses an index. If the query does not use an index, then the cost consists of the table scan cost only. If the query uses an index, then the cost consists of the index scan cost plus the table access cost, which is the cost when the system retrieves specified tuples from table. The *MaintCst*() depends on the index configuration. If the index exists, then the system must pay the index maintenance cost regardless of how the index is used. Other costs *ConstCst*(), such as the update cost of a data file and the output data cost, are paid constantly, and are not related to query plan and configuration. Hence, the cost of q_i in configuration c_k can be formulated as follows:

$$cost(q_i, c_k) = ReadCst(q_i, c_k) + MaintCst(q_i, c_k) + ConstCst(q_i). \tag{2}$$

The read cost contains the cost of the table and index. Hence *ReadCst*() can be formulated as follows:

$$ReadCst(q_i, c_k) = ReadIdxCst(q_i, c_k) + ReadTblCst(q_i, c_k). \tag{3}$$

3.3. Limited information caused by no repeated optimizer estimations

In practice, the general ISP problem is usually solved by adding additional constraints. For example, the space-limited ISP limits the upper bound of the total disk space used by indexes. In this study, we also add the following two constraints to the general ISP:

- An ISP solver can only use the original information calculated by a database optimizer for choosing a query plan.
- The original information related to index selection is stored in a limited space, which does not grow with the growth of database query operations.

The ISP problem with the above two constraints is called the information-limited ISP problem. For the first constraint, because optimizers are implemented differently and use different information for evaluating query plans [17,22], it is necessary to list the information that can be used. Commonly used information includes:

- The optimizer must know both the cost of the chosen query plan and the cost of the query plan using table scanning.
- The cost of a query plan is derived from the cost of each table or index file. Therefore, the system is assumed to know the costs of used tables or indexes, and to be able to distinguish between the read and write costs.
- A optimizer should adopt a cost model and know detailed information for using the cost model to estimate the costs of used tables or indexes. This information includes the number of records on a table, the number of tuples that have to be retrieved, the relevant attributes which are the indexing attributes in the WHERE clause of a query, and the selectivity of an index in the relevant attributes.

- Some system information which is independent of query should be known too. This information contains blocking factor of a system and the threshold of selectivity for using an index.

Besides, the optimizer is assumed to be sane. Sanity means that the optimizer chooses the query plan with the lowest cost among the known plans. In other words, the cost of the adopted plan must be lower than that of a table scan once the optimizer chooses the index scan query plan.

As the second constraint, when the number of database operations increases, the size of query-text log file also increases too. This is another heavy burden, because an ISP solver must read the whole query-text log file before analyzing these logs. In this study, we add the second constraint to limit the storage space of information used in index selection. Section 5 describes the data structure and algorithm used for storing this information compactly.

4. Index value evaluation

Whether an index is worth building depends on the comparative value of the benefit it provides and its maintenance cost. If the benefit is larger than the maintenance cost, then the index is worth building. However, to find the exact benefit consumes a large amount of resources because the benefits vary with different configurations. This section explains a two-step heuristic for evaluating the value of an index using only the information described in the previous section. The first subsection describes the heuristic which compares the cost between index scan and table scan. The second subsection explains the method of index substitution.

4.1. Index cost worse than table scan

As described in the Section 2, two methods: ADD and DROP can be used to solve the ISP. However, assuming limited information, the ADD method is not a proper approach, because the cost of non-existent index is hard to surmise. Hence, the DROP heuristic is adopted to solve the problem. Assume that c_k is a configuration and a is an index in c_k such that the cost of c_k is larger than that of c_k minus a. Index a can then be considered to be removed from the original index configuration c_k. The dropping criterion of index a can be formulated as follows:

$$\sum_{i\in M} cost(q_i, c_k) > \sum_{i\in M} cost(q_i, c_k - \{a\}). \tag{4}$$

Substituting the $cost()$ function by Eq. (2) and disregarding the constant cost gives:

$$\sum_{i\in M} ReadCst(q_i, c_k) + \sum_{i\in M} MaintCst(q_i, c_k) > \sum_{i\in M} ReadCst(q_i, c_k - \{a\}) + \sum_{i\in M} MaintCst(q_i, c_k - \{a\}). \tag{5}$$

The index maintenance cost does not depend on the index configuration or query plan. The system should pay the cost to maintain the index once the index is created. Hence the maintenance cost of index a, $a \in c_k$, can be written as follows:

$$MaintCst(\{a\}) = \sum_{i\in M} MaintCst(q_i, c_k) - \sum_{i\in M} MaintCst(q_i, c_k - \{a\}) \quad \text{for all } 1 < k < p \text{ and } a \in c_k. \tag{6}$$

Inserting $MaintCst(\{a\})$ in Eq. (6) into Eq. (5) gives:

$$\sum_{i\in M} ReadCst(q_i, c_k) + MaintCst(\{a\}) - \sum_{i\in M} ReadCst(q_i, c_k - \{a\}) > 0. \tag{7}$$

This DROP heuristic is commonly used in traditional index selection tools, but suffers from one difficulty in the proposed model. The proposed system does not know the read cost of configuration $c_k - \{a\}$, because the system is limited unless it queries the optimizer again. The optimizer may use another query plan in the configuration $c_k - \{a\}$, but the system does not know the read cost of this configuration. However, no matter which query plan is adopted by the optimizer, the cost must be lower than the cost of a table scan. Hence, the table scan cost can be used as the upper bound of the read cost of configuration $c_k - \{a\}$. The formula of determination can be written as follows:

$$\sum_{i\in M} ReadCst(q_i, c_k) + MaintCst(\{a\}) - TblSCst(a) \times IdxSNum(a) > 0. \tag{8}$$

In the above equation, $TblSCst(a)$ denotes the table scan cost of the table including indexing attribute a, and $IdxSNum(a)$ represents the number of index scan methods adopted. The left hand side of this formula is called the lower bound of the dropping criterion, and all indexes that satisfy Eq. (8) are deleted from the configuration.

4.2. Index substitution

The aforementioned DROP heuristic can detect some seemingly bad indexes. However if only this heuristic is adopted, the system may delete too many indexes. The reason is that once an index α is dropped, the optimizer may select another index β for the queries that originally used the index α, resulting in an increase in the scan number of index β. This phenomenon is

called index substitution. The dropping criterion for the index β decreases, once index substitution occurs on the index β. If the situation does not get detected, the lower bound of the dropping criterion of index β may be overestimated and deleted by mistake.

The exact method to find the substituting index of a deleted index is to query the optimizer. However, we wish to prevent this outcome. The method of identifying the substituting index from the limited information as described in section is just what we want. Since the optimizer is assumed to be aware of the relevant attribute list of a query and the selectivity of a relevant attribute, the original index α is substituted by the index β if they conform to the following conditions:

- Index α and index β are totally in the relevant attribute list.
- The selectivity of the index β is lower than the threshold currently applied by the optimizer to decide whether the database system uses an index.

If several indexes are in accordance with the two conditions, then the index with the lower selectivity is chosen as the substituting index. When a specific index is deleted, its cost is transferred to the substituting index. The transferring cost is divided into the read costs of the table and of the index, corresponding to *ReadTblCst* and *ReadIdxCst*() in Eq. (3), respectively. The cost of original index α is not entirely transferred to the substituting index β, because the costs of the two indexes are different. Due to the limitation of Section 3.3, it is not possible to ask the optimizer for the cost of index β. However, the system knows the detailed information which can be used to estimate the costs of the index by the cost model of index.

For the read cost of table *ReadTblCst*(), this study adopts the well-known Bernstein model [6], which is used to estimate the number of page accesses (NPA) of a table file. Assume that a relation contains T tuples, and requires P pages to store the relation. The number of tuples that would be retrieved when a query arrives at a database can be estimated as $\hat{t}$. The selectivity of this query is s, and the blocking factor of the database system is b. The formula of Bernstein model is as follows:

$$TableNPA(\hat{t},P)=\begin{cases}\hat{t} & \text{if } \hat{t}\leqslant\frac{P}{2},\\ \left\lceil\frac{\hat{t}+P}{3}\right\rceil & \text{if } \frac{P}{2}<\hat{t}\leqslant 2P,\\ P & \text{if } 2P<\hat{t}.\end{cases} \tag{9}$$

For the read cost of index *ReadIdxCst*(), assuming that the index file is organized as B+ tree, the number of page accesses in an index file can be formulated as follows:

$$IndexNPA(b,s,T)=\lceil\log_b T\rceil+\lceil T\times s\times\frac{1}{b}\rceil-1. \tag{10}$$

The values of those parameters in the above formulations are known by an optimizer in the assumptions of Section 3.3. The optimizer uses this information to choose a query plan, and the proposed system utilizes this information to recommend an index configuration.

Assume a query q_i is run at a configuration c_k. Using Eq. (10), let the estimated read cost of index α be $IndexNPA_\alpha$, and let that of index β be $IndexNPA_\beta$. Using Eq. (9), let the estimated read cost of the table with using index α be $TableNPA_\alpha$, and $TableNPA_\beta$ with using index β. Then the ratio between the original index estimation and substituting index estimation is used as the transfer multiplier. The table cost transfer function of a query q_i at a configuration c_k can then be formulated as follows:

$$TableCostTrans(\alpha,\beta)=ReadTblCst(q_i,c_k)\times\frac{TableNPA_\beta}{TableNPA_\alpha}. \tag{11}$$

The index cost transfer function can similarly be formulated as follows:

$$IndexCostTrans(\alpha,\beta)=ReadIdxCst(q_i,c_k)\times\frac{IndexNPA_\beta}{IndexNPA_\alpha}. \tag{12}$$

Noteworthy is the fact that the evaluation method of subsection 1 is independent of index type, but that Eq. (10) in Section 2 is related to the index type. In other words, if the database management system adopts the hashing index, then the evaluation method in Section 1 can be applied without any modification. Also, the evaluation method in Section 2 can be used but the cost model in Eq. (10) needs to be modified.

5. Table schemas and the index selection algorithm

The information used to evaluate the index in the above section is held by the database optimizer when queries are executed. The information should be stored in limited space for conforming to the constraints proposed in Section 3.3. In this section, we propose the table schemas, and then explain how an algorithm uses the data in these tables to execute the index evaluation presented above. The proposed system uses a total of three tables, namely, RelationCost, IdxStatistics and IdxSubstitute. Table 1 shows the table schemas.

Table 1
Schemas of tables used to store information for index selection.

Table Name	Attribute Name	Description
RelationCost	RelID	ID of relation
	TblSCost	cost of table scan
IdxStatistics	RelID	ID of relation
	IdxID	ID of index
	IdxSNum	accumulative number of index scans
	IdxRCost	accumulative cost of index read
	IdxWCost	accumulative cost of index write
	TblRCost	accumulative cost of table read when using index
IdxSubstitute	OIdxID	ID of original index
	SIdxID	ID of substituting index
	SIdxNum	accumulative number of index substitutions
	SIdxRCost	accumulative IndexCostTrans() in Eq. (12)
	STblRCost	accumulative TableCostTrans() in Eq. (11)

The RelationCost table is used to store the ID of the relation and the cost of table scan. The RelID attribute is the primary key of the RelationCost table. The IdxStatistics table is used to store the query statistics. The IdxID is set as the primary key, and the RelID is the foreign key referencing the RelID of the RelationCost table. When a database system executes a SQL statement using an index, the IdxSNum value of the index record is incremented by 1, and the estimated costs of *ReadIdxCst*(), *ReadTblCst*(), and *MaintCst*(), as described in Eqs. (2) and (3), are accumulated to the corresponding attributes IdxRCost, TblRCost, and IdxWCost. Besides, when a database system executes a write SQL statement, the IdxWCost value of the attribute in the SQL statement is updated regardless of which access path the system uses.

In addition to the cost of query, the substitution cost described in Section 4.2 is computed and accumulated to the IdxSubstitute Table. The two attributes OIdxID and SIdxID form the composite primary key, and are both foreign keys referencing IdxID in the IdxStatistics table. When the system detects the possibility of substitution, the SIdxNum attribute of the corresponding record is incremented by 1, and the estimated transformation values in Eqs. (11) and (12) are accumulated to the STblRCost and SIdxRCost attributes.

The above three tables occupy limited space; and the maximum amount of space can be roughly estimated as follows. In a workload, assume the number of tables is t, the number of considered indexes is r, and the sizes of one record in table RelationCost, IdxStatistics, and IdxSubstitute are s_1, s_2 and s_3, respectively. Then the space used by RelationCost is $s_1 * t$, and that used by IdxStatistics is $s_2 * r$. Besides, the maximum space used by IdxSubstitute is $s_3 * r * (r-1)$ because an index may be maximally substituted by the other $(r-1)$ indexes. Hence the maximum space used by the three tables is:

$$s_1 * t + s_2 * r + s_3 * r * (r-1). \tag{13}$$

For example, assume the number of tables is 10, the number of considered indexes is 100, and sizes of a single record in the three tables are 8, 24, and 20, respectively. Then the maximum space used by the three tables is 196 KB. Hence, no matter how great the amount of database operation is, the system will only use this limited space to store information for later index estimation. Actually, the estimated boundary is not tight, because we assume that each index may be substituted with the other $(r-1)$ indexes. In the real case, an attribute is only used together with specific attributes in queries. It is almost impossible that an attribute is used in conjunction with the other 99 attributes. For example, if the average number of substituted indexes is 5, then the actual used space in the above example is only about 10 KB, a case that is more plausible.

In a nutshell, the proposed system bases its index recommendation on the above information. The whole algorithm is given in Fig. 3. Once the user introduces a request for index recommendation, the system first creates a table @*idxCandidate* which contains the index information, and then defines a calculated column *isDropped* which calculates the dropping criterion according to the Eq. (8). The system filters out the indexes with negative *isDropped* values, and selects the index with the maximum *isDropped* value for deletion. Once an index is selected as the deleted index, its cost is transferred to other indexes according to the SIdxID attribute. The system executes the above procedure until the *isDropped* value of each remaining index is smaller than zero. The system performance of the proposed algorithm mainly depends on the number of I/O of tables containing RelationCost, IdxStatistics, and IdxSubstitute. The algorithm reads the three tables only once, and their size is small, as described previously. The algorithm in Fig. 3 is written in ANSI SQL style statements and uses variables of type table. The style makes the description of the algorithm more clear and concise. However, the proposed algorithm can be made suitable for environments lacking SQL by statement transformation.

6. Performance evaluation

The performance of the recommended configuration was tested in PostgreSQL (Version 7.3), a well-known open source database system [26]. The tests were created according to the schema defined in TPC-H [28], and the size of the experimental database was 1 G-byte. The queries defined in TPC-H only contained reading operations. Hence, some clauses in TPC-H

```
Input: RelationCost, IdxStatistics, and IdxSubstitute tables
Output: recommended index set
Algorithm:
  declare variable of type table: @ExistIdx
  set schema of @ExistIdx as only one column: IdxID
  declare variable of type table: @IdxCandidate
  set schema of @IdxCandidate as : (all columns of IdxStatistics, TblSCost, isDropped)
  set the values of the column isDropped using the following equation:
        IdxRCost+IdxWCost+TblRCost-IdxSNum*TblSCost
  declare variable of type table : @IdxCanSub
  set schema of @IdxCanSub as the table schema of IdxSubstitute
  /******************** Begin of disk I/O Part1 ********************/
  // Execute the following SQL to get candidate index data
  Insert @IdxCandidate(I.*, TabSCost)
  Select I.*, TblSCost
  From IdxStatistics as I join RelationCost as R where I.RelID=R.RelID
  /******************** End of disk I/O Part1 ********************/
  // Execute the following SQL to add recommended index
  Insert @ExistIdx (select IdxID from @IdxCandidate where isDropped<0)
  // Execute the following SQL to delete index from @IdxCandidate
  Delete @IdxCandidate where isDropped<0
  /******************** Begin of disk I/O Part2 ********************/
  // Execute the following SQL to get index substitution data
  Insert @IdxCanSub
  Select * from IdxSubstitute
  Where OIdxID in ( select distinct IdxId from IdxCandidate where isDropped>0 )
  /******************** End of disk I/O Part2 ********************/
  declare variable of type integer: @DelID
  do
        // Execute the following SQL to select the dropped index
        Select top 1 @DelID=IdxID From IdxCandidate Order by isDropped
        // check the break condition
        if row count of above statement is zero then exit and return @ExistIdx
        // Execute the following SQL to delete index from @IdxCandidate
        Delete @IdxCandidate where IdxID=@DelID
        // Execute the following SQL to do cost transformation
        Update @IdxCandidate
        Set IdxRCost=IdxRCost+S.SIdxRCost, TblRCost=TblRCost+S.STblRCost
        , IdxSNum=IdxSNum+S.SIdxNum
        From @IdxCanSub as S
        Where OIdxId=@DelID and @IdxCandidate.IdxID=S.SIdxID
        // Execute the following SQL to add recommended index
        Insert @ExistIdx (Select IdxID From @IdxCandidate Where isDropped<0)
  while( true )
```

Fig. 3. Index selection algorithm without repeated optimizer estimations.

queries were selected to create 25 read queries and 15 write queries. These 40 queries were used as the basic samples of our experiments. All indexes were created in the initial state, and the information described in Table 1 of each query was logged by using a modified EXPLAIN statement [8,20].

The tested workload was named target workload, and was composed of some queries chosen from the above 25 read queries and 15 write queries. The experiment was performed in the following two steps: (1) creating the target workload, and (2) testing the performance of the target workload in initial and recommended configurations. In the first step, the test program randomly selected 10 read queries and 10 write queries from the basic samples to form workloads, and then picked the workloads which needed tuning index configuration as the target workloads. The workloads without needing tuning were ignored. A total of 500 workloads were created, and 28 workloads were chosen as the target workloads from the 500 workloads according to the dropping criterion of Eq. (10).

In the second step, the proposed algorithm was used to compute the recommended configuration of each target workload. The target workloads on the initial and recommended configurations were then executed to measure performance.

Table 2
Performance improvement between recommended and initial configurations.

ID	Initial	Recommended	IRatio (%)	ID	Initial	Recommended	IRatio (%)
435	3,668,837	2,000,279	45.48	328	2,106,179	1,446,008	31.34
154	4,308,673	2,469,117	42.69	293	2,716,573	1,925,860	29.11
264	1,739,998	1,042,065	40.11	53	2,754,271	2,014,732	26.85
88	3,012,668	1,839,305	38.95	260	1,536,498	1,146,068	25.41
247	2,244,862	1,378,796	38.58	312	2,873,897	2,196,138	23.58
274	3,169,001	1,993,144	37.10	384	3,573,072	2,736,112	23.42
281	1,604,861	1,029,140	35.87	242	2,813,891	2,272,141	19.25
4	1,984,013	1,289,817	34.99	243	2,601,292	2,149,489	17.37
302	3,039,222	1,983,747	34.73	33	1,257,269	1,090,107	13.30
90	3,031,366	1,988,234	34.41	121	3,419,598	2,990,653	12.54
204	3,614,481	2,370,697	34.41	262	3,914,131	3,460,650	11.59
472	2,837,094	1,864,751	34.27	348	1,832,293	1,652,538	9.81

The number of disk I/O and the number of buffer hits were logged in the database catalog for each query execution. The number of disk I/O operations can be regarded as the performance measure of actual execution time, because the buffer speed is far faster than the disk speed. However when evaluating the performance of index selection algorithms, it is always assumed that no buffers are used in the system for excluding other complex factors such as buffer management policy. Therefore, a buffer hit in the experiment should be treated as a disk I/O. This study adopts the above viewpoint of performance measurement. The summation of the number of disk I/O operations and the number of buffer hits was used as the performance measure of a configuration.

However, the above performance measurement depended on the content of queries in the workload. An ideal measurement should ignore other factors and only focus on the performance improvement from the proposed algorithm. Hence, we define the improvement value and the improvement ratio (*IRatio*). The improvement value is defined as the performance measure of the initial index configuration minus the performance measure of the proposed algorithm, and the *IRatio* is defined as the improvement value divided by the performance measure of initial configuration. The experimental results were sorted by the *IRatio*, and the first two and last two workloads were filtered out as outliers. The final results are shown in Table 2.

The *ID* column denotes the workload ID. The *Initial* and *Recommended* columns represent the performances of the initial configuration and recommended configuration, respectively. The *IRatio* column represents the improvement ratio. The recommended index configurations of all workloads preformed better than the initial index configurations. The maximum improvement ratio was 45%. The average of the improvement ratio was 28.97%. The experimental results indicate the effectiveness of the proposed algorithm.

7. Conclusions and future works

In this study, we define a modified ISP problem called the information-limited ISP, and propose an efficient solution to the information-limited ISP problem for DBMS systems without virtual index. The key idea of information-limited ISP is to effectively exploit information when queries are executed. While a query is delivered to a database system, the optimizer evaluates the cost of various query plans and chooses an access path for the query. The information in the evaluation stage is aggregated and recorded in a limited space. The proposed algorithm recommends indexes only based on the previously known information without requerying the optimizer. The proposed algorithm has been tested on a PostgreSQL database system using TPC-H data, and the experimental results demonstrate its effectiveness.

However, there are still some issues which should be studied further in the future. First, the considered indexes should be extended to multi-column indexes, which are useful for some complex queries. The proposed index evaluation method in Section 4.1 is still suitable for multi-column indexes, because this method is independent of index type. However, the substitution of multi-column index is more complex, and should be formulated carefully. Second, the index evaluation heuristics under the constraint of only using the information from the optimizer needs further developments. In this study, we have developed two heuristics in Section 4; more various heuristics would be helpful for recommending better index configuration. Third, the performance comparison between the recommended configuration and the optimal configuration should be studied. Although the proposed method does not focus on finding the optimal configuration, it is still interesting to study the performance gap between the two configurations. We did not attempt this because the PostgreSQL database system did not have a virtual index facility, making inquiry into optimal configuration very difficult. Fourth, how much minimum information is required to get the optimal configuration needs further study. The traditional ISP and the proposed information-limited ISP are merely the two extremities. The traditional ISP knows the information contained in all configurations and certainly can achieve the optimal configuration. The proposed information-limited ISP knows the information in the current configuration and tries to get a better configuration. An interesting question is whether there is an intermediate approach. In other words, if the system can offer more information in another configuration, then how much can this information contribute to an improved configuration? Or if the optimal configuration is the only target, then how much configuration infor-

mation is the lower boundary to achieve the optimal configuration. Although the ISP is a comprehensively-studied question, we believe that further studies concerning it will be helpful in shedding light on this classic problem.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments.

References

[1] E. Balas, E. Zemel, An algorithm for large zero-one knapsack problems, Operations Research 28 (1980) 1130–1154.
[2] E. Barcucci, A. Chiuderi, R. Pinzani, M.C. Verri, Index selection in relational databases, Lecture Notes in Computer Science 364 (1989) 24–36.
[3] E. Barcucci, E. Grazzini, R. Pinzani, Index selection in a distributed data base, in: DDSS'84, Proceedings of the Third International Seminar on Distributed Data Sharing Systems, 1984, pp. 179–187.
[4] E. Barcucci, R. Pinzani, R. Sprugnoli, Optimal selection of secondary indexes, IEEE Transactions on Software Engineering 16 (1) (1990) 32–38.
[5] L. Bellatreche, R. Missaoui, H. Necir, H. Drias, Selection and pruning algorithms for bitmap index selection problem using data mining, Lecture Notes in Computer Science 4654 (2007) 221–230.
[6] P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve, J. James, B. Rothnie, Query processing in a system for distributed databases (SDD-1), ACM Transactions on Database Systems 6 (4) (1981) 602–625.
[7] A. Caprara, M. Fischetti, D. Maio, Exact and approximate algorithms for the index selection problem in physical database design, IEEE Transactions on Knowledge and Data Engineering 7 (6) (1995) 955–967.
[8] C.-C. Chang, An Index Tuning Scheme Based on the Evaluation of Index Replacement Costs, Master Thesis of National Chung Hsing University, Taiwan, 2005.
[9] S. Chaudhuri, V. Narasayya, Microsoft index turning wizard for SQL Server 7.0, in: SIGMOD'98, Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, ACM Press, New York, NY, USA, 1998, pp. 553–554.
[10] S. Chaudhuri, V.R. Narasayya, An efficient cost-driven index selection tool for Microsoft SQL Server, in: VLDB'97, Proceedings of the 23rd International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997, pp. 146–155.
[11] S.S. Chawathe, M.-S. Chen, P.S. Yu, On index selection schemes for nested object hierarchies, in: VLDB'94, Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994, pp. 331–341.
[12] S. Choenni, H.M. Blanken, T. Chang, On the automation of physical database design, in: SAC'93, Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing, ACM Press, New York, NY, USA, 1993, pp. 358–367.
[13] D. Comer, The difficulty of optimum index selection, ACM Transactions on Database Systems 3 (4) (1978) 440–445.
[14] Y.A. Feldman, J. Reouven, A knowledge-based approach for index selection in relational databases, Expert Systems with Applications 25 (1) (2003) 15–37.
[15] S. Finkelstein, M. Schkolnick, P. Tiberio, Physical database design for relational databases, ACM Transactions on Database Systems 13 (1) (1988) 91–128.
[16] M. Golfarelli, S. Rizzi, E. Saltarelli, Index selection for data warehousing, in: DMDW'02, Proceedings of the Fourth International Workshop on Design and Management of Data Warehouses, 2002, pp. 33–42.
[17] G. Graefe, Query evaluation techniques for large databases, ACM Comput. Surv. 25 (2) (1993) 73–169.
[18] H. Gupta, V. Harinarayan, A. Rajaraman, J.D. Ullman, Index selection for OLAP, in: ICDE'97, Proceedings of the 13th International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, 1997, pp. 208–219.
[19] J.A. Hoffer, An integer programming formulation of computer data base design problems, Information Sciences 11 (1) (1976) 29–48.
[20] C.J. Huang, A New Approach to Evaluating Index Benefit Using Database System Statistics, Master Thesis of National Chung Hsing University, Taiwan, 2004.
[21] M.Y.L. Ip, L.V. Saxton, V.V. Raghavan, On the selection of an optimal set of indexes, IEEE Transactions on Software Engineering 9 (2) (1983) 135–143.
[22] M. Jarke, J. Koch, Query optimization in database systems, ACM Comput. Surv. 16 (2) (1984) 111–152.
[23] S. Jiang, B.S. Lee, Z. He, Cost modeling of spatial operators using non-parametric regression, Information Sciences 177 (2) (2007) 607–631.
[24] S.S. Lightstone, G. Lohman, D. Zilio, Toward autonomic computing with DB2 universal database, SIGMOD Rec. 31 (3) (2002) 55–61.
[25] Oracle, Oracle Enterprise Manager 9i Tuning Pack. <http://www.oracle.com/technology/products/oem/files/tp.html>.
[26] PostgreSQL, The world's most advanced open source database system. <http://www.postgresql.org>.
[27] S. Rozen, D. Shasha, A framework for automating database design, in: Proceedings of the International Conference on Very Large Databases, 1991, pp. 401–411.
[28] TPC, Transaction Processing Performance Council. <http://www.tpc.org>.
[29] J. Wu, T. Srikanthan, An efficient algorithm for the collapsing knapsack problem, Information Sciences 176 (12) (2006) 1739–1751.
[30] D. Zilio, S. Lightstone, K. Lyons, G. Lohman, Self-managing technology in ibm db2 universal database, in: CIKM'01, Proceedings of the 10th International Conference on Information and Knowledge Management, ACM Press, New York, NY, USA, 2001, pp. 541–543.