Adaptive and Automated Index Selection in RDBMS

Martin R. Frank, Edward R. Omiecinski and Shamkant B. Navathe

College of Computing, Georgia Institute of Technology, Atlanta GA 30332-0280, USA
{martin,edwardo,sham}@cc.gatech.edu

We present a novel approach for a tool that assists the database administrator in designing an index configuration for a relational database system. A new methodology for collecting usage statistics at run time is developed which lets the optimizer estimate query execution costs for alternative index configurations. Defining the workload specification required by existing index design tools may be very complex for a large integrated database system. Our tool automatically derives the workload statistics. These statistics are then used to efficiently compute an index configuration. Execution of a prototype of the tool against a sample database demonstrates that the proposed index configuration is reasonably close to the optimum for test query sets.

1 Introduction

Relational database management systems (RDBMS) are by far the most popular database systems today. Despite their known shortcomings in non-traditional applications like engineering and image processing, no clear alternatives have evolved despite substantial research. Thus RDBMS are likely to dominate the commercial arena for years to come, especially for business applications.

Relational databases use indices to provide fast access to data. The presence of an index reduces the search time for indexed data items but also complicates update operations since the tuples as well as the indices must be updated. Hence there is a tradeoff involved in selecting indices and indexing every column is rarely a good idea\(^1\). This tradeoff decision will be referred to as the index selection problem (ISP).

In the context of this paper the ISP refers to tailoring the index configuration to the database usage profile, not to selecting an index set to process a single query. The single-relation ISP denotes the ISP reduced to selecting an index configuration for a single relation, which is much easier than the general ISP since join queries pose the hardest problems. The single-index ISP refers to choosing an index configuration consisting of single indices, combined (concatenated) indices are excluded from consideration.

The decision as to which attributes to index is influenced by numerous factors, such as database usage, characteristics of the database system and the underlying operating system. Due to this complexity, it is difficult for the unassisted database administrator (DBA) to choose a good index configuration for a large integrated database. Tools were

\(^1\) In the context of B-tree indices, not Grid files.
proposed which relieve some of this burden from the DBA, however even the most sophisticated ones like DBDSGN [FST88] still require the DBA to manually specify the workload. The designer has to specify the workload as a small set of weighted representative queries. It is unclear how the DBA can condense the transactions on a huge DBMS (consider 100 tables and 1000 transactions per hour) to just - say - 20 representing the original workload. Hence mechanisms have to be found which derive the workload information for those tools from the database system itself.

Throughout this paper the workload specification (or usage input) denotes the part of the input for index selection tools which contains information about the workload for which the index configuration is optimized. The workload specification may consist of statistics (e.g. how often a certain column was referenced or updated during the last time period) or representative queries.

Research in the area of automated index selection has treated statistics gathering and statistics evaluation separately. Consequently: (1) the existing statistics gathering mechanisms - originally intended for debugging - consume too much overhead to continuously collect usage data , and (2) the existing index selection tools are rarely used in practice because they require hard-to-derive statistics as input. It is our belief that a successful tool must integrate both aspects.

Our tool requires no usage input specified by the designer, it derives all its input automatically during regular database usage. The output is an optimum\(^2\) index configuration for the queries during the recording period in the sense that it minimizes the average query execution time.

\section{Previous Research}

We discuss previous work on the index selection problem (ISP) with respect to the workload specification. The list is by no means complete. Research in the area not discussed here has been done by Palermo [Pal70], King [Kin74] and many others.

Stonebraker [Sto74] constructs a probabilistic model for database activity and solves the single-index single-relation ISP for certain special cases in polynomial time. The formalization of the index selection problem provides insight into its difficulty, but the results are valid for special cases only and there is no methodology presented for finding an index configuration for the general case. The usage input parameters are (1) the probability that a query is a non-retrieval query (Insert, Delete, Update) as opposed to a retrieval query (Select) and (2) for all columns \(i\) the probability that column \(c_i\) appears restrictively in a query. These statistics have to be specified as input to the tool.

It is obvious that these statistics are not trivial to derive. There are also some general problems with analytical approaches to the ISP. First, substantial simplifications have to be made to derive an analytical solution. Second, the model becomes obsolete if there are changes to the query processing strategy or to other modeled aspects of the DBMS.

Schkolnick [Sch75] presents a more general probabilistic model and an algorithm that solves the single-index single-relation ISP significantly faster than the naïve approach. As in [Sto74] a cost function is derived that gives the expected average query execution cost depending on the index configuration. An algorithm is presented that finds the optimal

\(^2\) Truly optimal only for a restricted version of the index selection problem.
index configuration provided that the target function is \textit{regular}. The usage input has a similar flavor as in \cite{Sto74}, for example the probability $\alpha_j(a)$ that column $j$ is restricted to value $a$ in a query. The approach does not qualify for a practical index selection tool since (1) maintaining the detailed statistics used in the cost function requires too much overhead (2) the amount of storage for these statistics is substantial (3) and the cost of the evaluation run ($O(2^{\sqrt{\log c}})$) is still too costly even for a moderate number of columns $c$. There are also the general problems of analytical approaches for an index selection tool as discussed above.

Hammer and Chan \cite{HC76} envision fully-automated index selection (single-index, single relation) but their approach is unrealistic for real-life databases in various ways. Fully automated index selection means that the system adopts the current index configuration based on automatically gathered statistics so that users do not even have to know about the concept of indices. An example for the database usage statistics used are the restrictive clauses for every query. Gathering, maintaining and evaluating such detailed statistics is clearly infeasible for an unrestricted query language.

Whang et al. \cite{WWS81} present a single-index multiple-relation index selection method based on a set of join methods that is \textit{separable}. This property reduces the index selection problem to finding a \textit{locally} optimal index configuration for each relation. The set of join methods is reduced to two because these are the only ones adhering to separability. It is unclear if the advantage of a better index configuration outweighs the disadvantage of not using efficient join methods which would otherwise be available. The usage input consists of a weighted set of queries. This is an improvement over specifying statistics of the kind discussed above. The general problem with this form of usage input is that the “representative” query set might not be representative of the real workload because it has to be of moderate size for complexity reasons.

Finkelstein, Schkolnick and Tiberio \cite{FST88} discuss the single-index multiple-relation index selection methodology used by the commercially available physical design tool RDT (Relational Design Tool) and its experimental prototype DBDSGN (DataBase DeSiGN Tool). The usage input consists of a weighted representative query set as in \cite{WWS81}. An obstacle for its success is that the designer has to specify a set of non-intuitive parameters to reduce the run time of the tool which require insight into the internal algorithm of the tool. The designer could choose not to specify any of these parameters but then the execution time would become impractical. However, the most significant problem is – again – finding a “representative” query set for the real database usage.

3 Index Benefit Graphs

Which indices\(^3\) are useful for a given query and how useful are they? This is the central question for any index selection tool operating on a query-by-query basis.

Our tool “converses” with the optimizer to answer this question. An example for such a dialog is shown in figure 1, it is shown in natural language for illustration only. The last response shown terminates the dialog. A special case for the optimizer’s first response is that it would use a sequential scan (or that the database transaction does not require access to tables at all). Then no savings are recorded for any index.

\(^3\) This includes indices which do not currently exist.
Tool: Which index would you use for query \( q \) if you could choose from a virtual index set \( p_1 \) which contains all possible indices for the database? And what is your cost estimate for executing the query using this index?

Optimizer: I would use index \( i_1 \), my cost estimate for the query using this index is \( c_1 \).

Tool: What is your index choice and cost estimate for query \( q \) when you can choose from the previous index set minus the index you just used \( (p_2 = p_1 - i_1) \)?

Optimizer: Then I would use index \( i_2 \), the cost estimate is \( c_2 \).

Tool: What is your index choice and cost estimate for the index set \( p_3 = p_2 - i_2 \)?

Optimizer: Then I would use index \( i_n \), the cost estimate is \( c_n \).

Tool: What is your index choice and cost estimate for the index set \( p_n - i_n \)?

Optimizer: Then I would not use any index at all but would rather do a sequential scan. The cost estimate for it is \( c_{seq} \).

**Fig. 1.** The Dialog between Tool and Optimizer

The optimizer must provide the following functionality: presented with an SQL statement [CB74] and an index set to choose from for processing a given statement it must be able to export the index set it would choose and its cost estimate for processing the query using this set:

\[
\text{optimizer(st: statement, presented-with: index-set) \rightarrow chosen: index-set, cost: real}
\]

The difference of this requirement as compared to the existing EXPLAIN statements [FST88] is that the combined functionality of the EXPLAIN COST and EXPLAIN PLAN statements only provides for the following:

\[
\text{optimizer(st: statement) \rightarrow chosen: index-set, cost: real}
\]

So EXPLAIN statements can only provide optimizer information based on the currently existing index configuration. Our tool constructs an Index Benefit Graph (IBG) using the information from the dialog, a simple example of such a graph is shown in figure 2. The label on an arrow is the index set the optimizer is presented with, the index set the optimizer chooses is pointed to by the arrow. These sets will be referred to as the presented-with and chosen sets. The encircled number denotes the optimizer’s cost estimate for the query if the index set to its left is used. The numbers in squares provide a numbering of the nodes. This IBG corresponds to the dialog in figure 1 where \( i_1 = \{b\}, i_2 = \{d\}, i_3 = \{c\}, i_4 = \{a\}, p_1 = \{a, b, c, d\}, p_2 = \{a, c, d\}, p_3 = \{a\}, p_4 = \{a\}, p_5 = \{\}, c_1 = 23, c_2 = 27, c_3 = 51, c_4 = 51 \) and \( c_{seq} = 80 \).

So far we have assumed that the optimizer chooses at most one index per query. In this case constructing the IBG is straightforward. Let us now consider the general case.

\footnote{We will discuss alternative cost measures shortly, assume the cost is the execution time for the moment.}
Index sets are chosen to process a query. This leads to the problem of determining which subsets of the chosen set would also be beneficial, the beneficial subsets problem. If an index set \{a, b, c, d\} is used, would the set \{a, c, d\}, say, also be beneficial? And what would be the execution cost when using \{a, c, d\}?

There is a simple way to solve the problem. Let the elements of a set be numbered in an arbitrary but consistent way (e.g., alphabetically) and let a singleton set consisting of the ith element of set C be denoted by \(C^i\). For each node of the graph where the optimizer was presented with index set \(P\), spawn all possible subsets of size \(|P| - 1\), namely \(P - P^i\) for \(1 \leq i \leq |P|\). We will refer to a graph built in this way as a Brute-Force Graph (BFG). An example for a BFG is shown in figure 3. For example, in the root node \(P = \{a, b, c, d\}\) so that \(|P| = 4\) subgraphs starting with \(P - P^1 = \{a, b, c, d\} - \{a\} = \{b, c, d\}\), \(P - P^2 = \{a, c, d\}\), \(P - P^3 = \{a, b, d\}\) and \(P - P^4 = \{a, b, c\}\) are spawned. This results in a graph with \(2^{|P|-1}\) nodes where \(P_R\) denotes the set the optimizer is initially presented with in the root node. This graph presents the optimizer with all possible subsets of \(P\), and will therefore contain all sets that the optimizer may choose to process the query.

Building a graph exponential in the number of potential indices for every query is infeasible. Our IBGs consist of far less nodes than corresponding BFGs on average which makes their use practical. Index Benefit Graphs are constructed in the following way. If the optimizer chose the index set \(C\) from the index set \(P\) it was presented with, then \(|C|\) subgraphs starting with \(P - C^i\) are spawned for \(1 \leq i \leq |C|\). Figure 4 gives an example for an IBG constructed in this way. For example, in the root node \(P = \{a, b, c, d\}\) and \(C = \{a, b\}\), so that \(|C| = 2\) subgraphs starting with \(P - C^1 = \{a, b, c, d\} - \{a\} = \{b, c, d\}\) and \(P - C^2 = \{a, b, c, d\} - \{b\} = \{a, c, d\}\) are spawned.

An IBG contains far less nodes than a BFG on average. Does this mean that we miss chosen sets that are contained in a BFG? In the rest of this section we prove that this
 cannot happen assuming that the optimizer shows reasonable (“sane”) behavior. The main result is that the IBG contains the same information as the corresponding BFG if the “Sanity Property” holds.

Let $P_1$ and $P_2$ denote index sets the optimizer is presented with, $C_1$ and $C_2$ are index sets the optimizer chooses for executing a given query. $P \rightarrow C$ means “presented with $P$ the optimizer chooses $C$ ($\subseteq P$)”. Let $A \subseteq B$ denote that $A$ is a proper subset of $B$. The empty set is denoted by $\{\}$. 

**Property 1 (The Sanity Property)** “For a given query, new indices to choose from can only affect the chosen index set when they are part of it.”

$$\forall P_1, C_1, P_2 \supseteq P_1, C_2 : (P_1 \to C_1 \land P_2 \to C_2) \Rightarrow (C_2 = C_1) \lor (C_2 \cap (P_2 - P_1) \neq \{\})$$

**Property 2 (Derived from Property 1)** “For a given query, if some indices were chosen from a presented-with set then they will be chosen again from a subset of this set which still contains all the chosen indices.”

$$\forall P_2, C_2 : (P_2 \to C_2) \Rightarrow (\forall P_1, C_2 \subseteq P_1 \subseteq P_2 : P_1 \to C_2)$$

*Proof of (1) ⇒ (2) (by Contradiction)* Assume a violation of property 2: $\exists P_1, C_1, P_2, C_2 : C_2 \subseteq P_1 \subseteq P_2 \land C_1 \neq C_2 \land P_1 \to C_1 \land P_2 \to C_2$. This implies: $\exists P_1, C_1, P_2 \supseteq P_1, C_2 : P_1 \to C_1 \land P_2 \to C_2 \Rightarrow (C_2 \neq C_1) \land (C_2 \subseteq P_1 \subseteq P_2)$. This violates property 1 since: $C_2 \subseteq P_1 \subseteq P_2 : C_2 \cap (P_2 - P_1) = (C_2 \cap P_2) - (C_2 \cap P_1) = C_2 - C_2 = \{\}$. \qed
**Fig. 4.** The Index Benefit Graph for the Brute-Force Graph in Figure 3

*Example for Property 2 (and Property 1)* Consider $P_1 = \{a, b, c, d\} \rightarrow C_1 = \{b, c\}$. Then according to property 2 this implies:

1. $(P_2 = \{b, c\}) \rightarrow \{b, c\}$
2. $(P_2' = \{a, b, c\}) \rightarrow \{b, c\}$
3. $(P_2'' = \{b, c, d\}) \rightarrow \{b, c\}$

**Property 3 (Also derived from Property 1)** “For a given query, if the optimizer chooses from a superset of some presented-with set then it cannot choose a subset of the set it chose for this presented-with set before.”

$$
\forall P_1, C_1, P_2 \supset P_1, C_2 : (P_1 \rightarrow C_1 \land P_2 \rightarrow C_2) \Rightarrow C_2 \not\subseteq C_1
$$

*Proof of (1) ⇒ (3)* Properties 1 and 3 have the same left hand side. The right hand side of 1 gives: $(C_2 \subseteq C_1) \lor (C_2 \cap (P_2 - P_1) \neq \emptyset)$. If $C_2 = C_1$ then $C_2 \not\subseteq C_1$ follows trivially, else: $C_2 \cap (P_2 - P_1) \neq \emptyset \Leftrightarrow \exists x : x \in C_2 \land x \in P_2 \land x \notin P_1 \Rightarrow \exists x : x \in C_2 \land x \notin P_1 \Rightarrow C_2 \not\subseteq P_1 \Rightarrow C_2 \not\subseteq C_1(\subseteq P_1)$. \qed

*Example for Property 3 (and Property 1)* Consider $P_1 = \{a\}, C_1 = \{a\}$ and $P_2 = \{a, b\}$. What are optimizer choices for $C_2$ adhering to this property?

1. $C_2 = \{a, b\}$ is legal – using both indices may reduce execution time.
2. $C_2 = \{a\}$ is legal – index $b$ is of no use for this query.
3. $C_2 = \{b\}$ is legal – $b$ outperforms $a$ for this query.
4. $C_2 = \{\}$ is illegal – if $a$ is useful when presented with $\{a\}$ then $a$ should still be useful if the optimizer can choose from the superset $\{a, b\}$. If the index set the optimizer can chose from is augmented this should not make subsets useless that were chosen before.
Theorem 2 Cover. A graph starting with the presented-with superset $X \supset Y$ contains all chosen sets that a graph starting with the subset $Y$ contains.

Proof Let

$$A_i = \begin{cases} Y & \text{if } i = 0 \\ A_{i-1} \cup (X - Y) & \text{if } 1 \leq i \leq |X - Y|, \end{cases}$$

which implies $Y = A_0 \subset A_1 \subset \ldots \subset A_{|X - Y|} = X$ where $|A_i| + 1 = |A_{i+1}|$ for $0 \leq i \leq |X - Y| - 1$. Theorem 2 holds for each pair $(A_i, A_{i+1})$ due to Theorem 1, so that Theorem 2 holds for $(X, Y)$ by transitivity. \qed
**Lemma 4 Lossless.** An Index Benefit Graph contains all index sets the optimizer would choose to process the query if the Sanity Property is assumed.

*Proof* Since Theorem 3 holds for all nodes of the graph including the root node the pruning of the graph in this way is lossless and the Index Benefit Graph contains all chosen index sets that the corresponding Brute-Force Graph contains, where no subgraphs are pruned. □

**4 Our Tool**

An IBG contains all index sets the optimizer would choose for a given query. Retaining all these sets for each query would allow to find the optimal index configuration for a
set of queries at the expense of substantial storage cost for these statistics and of an evaluation run exponential in the number of indices. This is a viable methodology for a tool tailoring the index configuration to a small number of given queries like DBDSGN. It is not viable for a run-time tool operating on a continuous query stream due to the large number of queries processed.

Therefore our tool condenses the information contained in an IBG. Condensing the information is necessarily heuristic so that the proposed index configurations are no longer guaranteed to be truly optimal. However the performance of the methodology proved to be excellent in our simulations.

The information we extract is which indices bring which cost savings for a query as compared to processing the query without using any indices. In the simple case of an optimizer choosing at most one index to process a query the savings of an index $i$ in the graph is computed as $c_{eq} - c_i$. Consider the simple IBG of figure 2. For example the savings of index $a$ is computed as $80 - 51 = 29$ cost units. Table 1 contains all savings for the simple IBG in figure 2. No heuristics are involved so far.

<table>
<thead>
<tr>
<th>An index on column: $a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>would bring savings of</td>
<td>29</td>
<td>57</td>
<td>29</td>
</tr>
</tbody>
</table>

**Table 1.** The Savings for the Simple IBG

In the general case index sets will bring savings. This leads to the *savings attribution problem*. Namely, which individual indices of the chosen set should be credited with which savings? If index set $\{a, b, c\}$ provides a savings of 30 then which part of the savings are due to index $a$? We deal with the *savings attribution problem* in a heuristic way.

**Heuristic 1** If the savings of index set $\{i_1, \ldots, i_n\}$ is $s$ then we equally credit the savings for each individual index with $\frac{s}{n}$.

Therefore we loose the information which indices were used *together*, which is relevant information. Consider a hypothetic optimizer which only uses TID (tuple identifier) intersection. In this case using two indices together might be useful but using just one of them is not. However our simulations indicate that such situations are rare.

**Heuristic 2** The maximum of all savings an index has been credited with in the graph is retained and this constitutes the savings of this index for the current query.

This means that we take an optimistic view and retain the maximum savings the index *could* bring for the query. For example the savings of index $a$ in the IBG of figure 4 is computed as follows. Due to heuristic 1 its savings in node 1 is $\frac{420-80}{2} = 170$. Its savings in node 2 is $\frac{420-122}{2} = 298$. Due to heuristic 2 the maximum of 170 and 298 is retained so that the savings of index $a$ for the query of this IBG is defined as 298. Table 2 lists all savings for the IBG in figure 4.
Table 2. The Savings for the Complex IBG

All statistics that our tool maintains are the accumulated savings for each possible index. These savings are reset to zero at some time, then for every query the savings of each index involved are added to the corresponding overall savings. At a later time (e.g. after one week) the overall savings are transformed into an index configuration proposal. In the simplest case all indices with positive overall savings are taken into the proposed configuration. Several ways to limit the amount of storage consumed by indices are discussed in the Extensions section.

5 Simulation of the Tool’s Performance

We implemented a prototype of our tool using ORACLE Version 6 on a Sequent Symmetry. The prototype consists of shell scripts and PRO*C programs. Since existing optimizers cannot choose from a virtual set, we actually constructed the indices the optimizer was presented with and dropped the others for every query to simulate an optimizer capable of choosing from a virtual index set. Since we were changing the actual (not virtual) index configuration for every query in our prototype these experiments could measure the quality of the proposed index sets but not the runtime overhead of the tool.

We executed each query while enabling ORACLE’s trace facility and extracted information about which indices were used from the execution plan and extracted our cost measure from the execution statistics for the query. Therefore the prototype measures the actual execution cost while our tool relies on optimizer cost estimations. We experimented with three cost measures:

1. The CPU time used for the query. This measure turned out to yield near optimal index configurations. Its disadvantage is that it is influenced by the machine workload.
2. The number of logical block accesses for the query. The term “logical” refers to a block access in the main memory buffer which may or may not require a physical block access on disk. The advantage of this measure is that it does not vary with the machine workload. However it is not proportional to the time required to process the query. Consider a table which fits into the main memory buffer in its entirety. Then a query simply selecting the whole table will use e.g. $r_1$ logical reads - one per block of the table. Now consider a query joining this table with itself, this will result in $r_2 \gg r_1$ logical reads, but since the table fits into the buffer no more I/O is required than for the query above. Thus the execution time for the second query is not significantly higher than for the first but the number of logical block reads is.

5 ORACLE’s version of SQL embedded in C.
In our experiments, measuring the logical block accesses worked fine only if no join queries were involved.

3. The number of physical block accesses. The reasoning behind this measure is that the processing cost for a query tends to be dominated by the amount of physical I/O needed. Its disadvantage is that it is dependent on the current contents of the main memory buffer. However it still outperformed measuring the logical block accesses in our experiments.

Our tool is capable of using any cost measure for query execution as long as this measure is a simple numeric value. For example it would be possible to combine the measures discussed above.

5.1 Performance Results

Tables 3 and 4 contain the condensed results of our simulation runs. The “Batch” column indicates for which SQL batch the results to the right were obtained. Each such batch consisted of 10 – 20 queries, where a wide range of query types were used. The batches contained Select, Update, Insert, Delete, Create Table and Drop Table statements. Batch 6 contained especially complex queries. We used two tables with four columns each for our benchmarks. All columns were numeric to facilitate automatic tuple generation. The selectivities were close to 0% (single tuple), 25%, 50% and 100% for the columns of both tables. The first table contained 500 tuples, the second contained 5000.

To judge the performance of the index sets our tool proposed, we ran a brute-force script for each batch that simply measured the execution time of a batch for all possible relevant index sets. This brute-force method necessarily finds the best and worst index configuration. The “Best” and “Worst” columns in table 3 contain the best and worst execution times found for every batch. We then ran our tool for the three cost measures “CPU time”, “logical block accesses” and “physical block accesses” as described above. The columns CPU, LOGICAL and PHYSICAL in table 3 indicate the execution time when using the index set proposed by our tool for this batch. Table 4 contains the relative ranking of the index sets found by our tool. For example, measuring the logical block accesses with our tool for query batch 4 found the second best out of 16 possible index sets. The tool’s performance on query batches with a substantial gap between the best

<table>
<thead>
<tr>
<th>Batch</th>
<th>Best</th>
<th>Worst</th>
<th>CPU</th>
<th>LOGICAL</th>
<th>PHYSICAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>57.6</td>
<td>1:13.3</td>
<td>57.6</td>
<td>1:00.5</td>
<td>57.6</td>
</tr>
<tr>
<td>2</td>
<td>3:14.7</td>
<td>4:07.7</td>
<td>3:15.6</td>
<td>3:31.3</td>
<td>3:25.1</td>
</tr>
<tr>
<td>3</td>
<td>2:41.6</td>
<td>2:56.2</td>
<td>2:41.6</td>
<td>2:50.8</td>
<td>2:41.6</td>
</tr>
<tr>
<td>4</td>
<td>2:55.8</td>
<td>23:44.1</td>
<td>2:55.8</td>
<td>3:36.2</td>
<td>2:55.8</td>
</tr>
<tr>
<td>5</td>
<td>1:05.5</td>
<td>1:20.3</td>
<td>1:06.2</td>
<td>1:11.3</td>
<td>1:11.1</td>
</tr>
<tr>
<td>6</td>
<td>56.2</td>
<td>1:17.6</td>
<td>1:00.4</td>
<td>1:00.4</td>
<td>56.8</td>
</tr>
</tbody>
</table>

Table 3. Condensed Results - Execution Times

and worst execution times was excellent (see batch 4). The tool performed slightly worse
for batches where the best and worst execution time were close to each other (see batch 5). This is reasonable behavior since the index configuration is not of great importance in the latter case anyway. We hypothesize that the performance of the tool decreases with increasing complexity of the IGs because the effects of our heuristics are more drastic for large IGs than for moderate ones. The results for a batch of complex queries (batch 6) support this.

<table>
<thead>
<tr>
<th>Batch</th>
<th>Best</th>
<th>Worst</th>
<th>CPU</th>
<th>LOGICAL</th>
<th>PHYSICAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>64</td>
<td>1</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>256</td>
<td>3</td>
<td>84</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>256</td>
<td>11</td>
<td>81</td>
<td>71</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>16</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4. Condensed Results - Ranking

5.2 Estimation of the Run-Time Overhead

The run-time overhead of our tool is proportional to the number of nodes in the IG for the query. Therefore the average complexity of the IGs is critical.

It turned out that for nearly all cases the graphs had a modest number of nodes, even for highly complex queries. For example the optimizer did not use any indices for the quite complex query "select autogen1.col1 from autogen1 where not exists (select * from autogen2 where autogen1.col1 = 10000 and exists (select * from autogen2 where autogen1.col1 = autogen2.col1))". The largest IG for queries of our test batches contained eight nodes. The exception were queries of the type "select * from X where X.c1 = const1 and X.c2 = const2 and ...;" processed by TID intersection, where the particular optimizer we experimented with chose all subsets of the indices on the attributes connected by and. The number of nodes in the IG for this kind of query was exponential in the number of attributes.

6 Limitations

Our approach cannot handle combined indices. It is hypothesized that no efficient statistics gathering and evaluating tool can be built that investigates all possible combined indices because the performance degradation induced by the combinatorial explosion involved in doing this would outweigh the benefits of the tool. To our knowledge no tool was ever presented that is capable of detecting beneficial combined indices.

---

6 DBDSGN [FST88] has the option of specifying particular combined indices to be evaluated but it cannot detect such desirable combined indices itself.
Fig. 7. Using an Additional Machine for our Tool

Allow to combine the results of several recording periods. So far all savings are reset to zero at time $t_1$ and used to improve the index configuration at $t_2 > t_1$ when they are reset again for the new recording period. It may be desirable to combine the results of the last several recording periods. A standard solution — also used in [HC76] — is to use exponential smoothing to combine the results of the last recording periods.

Take index storage cost into account. So far our tool strictly optimizes the average query execution time without considering the tradeoff between execution time and index storage.
requirements. The simplest solution is to introduce heuristic restrictions like (1) the total space used by indices should not surpass 20% of the space for the real data, (2) use at most three indices per table, (3) use at most n indices overall. A less heuristic approach is to let the designer specify a factor λ which formalizes the tradeoff between response time r and storage requirements s. λ informs the tool as to which improvement in r will justify what increase in s. For example, an average response time improvement of 0.1 seconds might be worth spending 2 megabytes of disk space, λ = 5 × 10^{-8} sec/byte. So an index with the characteristics (r = 0.2 sec, s = 5 MB) will be disqualified since r/s < λ.

Allow the specification of exceptions. The tool assumes that the performance goal is to always minimize the average query execution time. This is not necessarily the case. Consider an unconventional database where critical data is rarely accessed but if it is it should be available in minimum time. Then it will be desirable to index this data despite its low reference frequency. Therefore the tool should allow the designer to specify a column set that will be indexed independent of the tool’s standard considerations. Implementing exceptions is straightforward.

Take index creation cost into account. If the proposed index configuration yields only slightly better performance than the current one, then the index reorganization cost might be higher than the total savings of the better index configuration. This issue has a low priority for us. First many conventional databases are taken offline at night or at weekends for maintenance – there is no performance penalty for queries if reorganization is done during this time. Second even if indices are constructed and dropped while the system is online it may still be done at times of low database usage where the performance penalty will hardly be noticeable; and the response time during the critical usage peaks will be improved.

8 Comparison against DBDSGN

Our research can also be used as a basis for an offline tool, which tailors the physical database design to a batch of queries provided to the tool as does DBDSGN [FST88]. However the run time of our tool is (in the worst case) only exponential in the number of relevant indices for each query and linear in the number of queries, whereas DBDSGN is exponential in the number of indices plausible for any of the queries in the batch\textsuperscript{7}. Consider the query batch shown in figure 8. There are 9 indices plausible for the batch.

\begin{verbatim}
select * from R_1, R_2, R_3 where R_1.a = R_2.a and R_2.a = R_3.a;
select * from R_1, R_2, R_3 where R_1.b = R_2.b and R_2.b = R_3.b;
select * from R_1, R_2, R_3 where R_1.c = R_2.c and R_2.c = R_3.c;
\end{verbatim}

Fig. 8. A Sample Batch

DBDSGN would execute the query batch \(2^9 = 512\) times whereas our tool would – in the

\textsuperscript{7}The effects of various optional heuristics in DBDSGN are not considered here.
worst case – execute each query for $2^3 = 8$ configurations. So DBDSGN would execute $3 \times 512 = 1536$ queries while our tool would execute $3 \times 8 = 24$. However, running DBDSGN in this way would guarantee to find the optimum configuration while our tool yields only near-optimal configurations. DBDSGN is inherently designed to work on query batches (of moderate size) – its methodology cannot be extended to work on continuous query streams.

9 Conclusions and Future Research

This research presents, to our knowledge, the first viable methodology for a run-time facility collecting statistics to optimize the physical design of a database. It has important characteristics which are prerequisites for such a tool: (1) the storage space consumed by the statistics is constant in the number of queries traced (2) the evaluation of the statistics is fast and straightforward.

We demonstrated that the proposed tool will tailor the physical design of the database to the past usage profile if the optimizer can estimate execution costs for virtual index sets, and that the resulting index configuration is reasonably close to the optimum index configuration.

The performance of the index sets proposed by our tool have been verified by simulation, but the actual run-time overhead of the statistics-collection part of our tool could not be measured but only estimated since it would involve modifying an optimizer. The natural extension of this research is to experiment with an actual optimizer.

References


