DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes

Gary Valentin, Michael Zuliani, Daniel C. Zilio
IBM Toronto Lab
valentin,zuliani,zilio@ca.ibm.com

Guy Lohman
IBM Almaden Research Center
lohman@us.ibm.com

Alan Skelley
University of Toronto
alan@cs.utoronto.ca

Abstract

This paper introduces the concept of letting an RDBMS Optimizer optimize its own environment. In our project, we have used the DB2 Optimizer to tackle the index selection problem, a variation of the knapack problem. This paper will discuss our implementation of index recommendation, the user interface, and provide measurements on the quality of the recommended indexes.

1. Introduction

The performance of queries in a relational database management system (RDBMS) has always been very sensitive to the indexes that exist on the tables in a database. Traditional B+-tree indexes can speed the execution of a query in one or more of the following ways:

- Applying predicates, i.e. by limiting the data that must be accessed to only those rows that satisfy those predicates;
- Ordering rows, i.e. to apply ORDER BY, GROUP BY, or DISTINCT clauses, or to merge-join a table with another table;
- Providing index-only access, i.e. to save having to access data pages by providing all the columns needed by a query;
- Enforcing uniqueness, i.e. by restricting the index to one row identifier per key value.

Specialized indexes may provide other advantageous aspects to query execution, such as statistics on the number of keys.

Since the advent of relational DBMSs, researchers have attempted to automate the design of databases, including the selection of indexes that would best serve a particular workload of queries. An index may have multiple columns as key columns, and the ordering of those columns is significant. Given that real applications such as SAP can have tens of thousands of tables, each table can have hundreds of columns, and a typical workload can have thousands of queries, the number of possible indexes to consider is staggering. Finding the set of indexes that optimize a workload of complex, multi-table queries having varying importance and subject to resource constraints, is a daunting combinatorics challenge.

Initially these design tools were completely separate from the DBMS engine itself. They independently proposed candidate indexes and attempted to evaluate the cost and benefit of each set of candidate indexes. A major advance in the design tools was the use of the engine's optimizer to evaluate the cost of queries, given a set of candidate indexes [FST 88]. This advance prevented duplication of the optimizer's cost model in the design tool, and ensured consistency with the optimizer's choice of index when the recommended indexes were subsequently created.

This paper presents what was done as the next logical step: Have the engine's optimizer recommend candidate indexes, as well as evaluate their benefit and cost. The DB2 Advisor, new in IBM's DB2 Universal Database (UDB) V6.1, utilizes a component in the optimizer that recommends candidate indexes based upon an analysis of each query, and then evaluates those indexes, all in one call to the engine! This approach significantly improves the quality of the indexes that are considered, and speeds the evaluation of alternatives by reducing the number of calls to the engine. By modeling the index selection problem as a variant of the well-known Knapsack Problem [GN 72], the DB2 Advisor is also able to optimize large workloads of queries in a reasonable amount of time.

The remainder of this paper is structured as follows. Section 2 describes the overall architecture of the DB2 Advisor

and its user interface. Section 3 details how the optimizer through the recommendation algorithm is able to recommend the best indexes for a given query. Section 4 presents the algorithm to extend this concept beyond just a single statement at a time to a workload of SQL statements, and subject to resource constraints (such as disk space). Section 5 contrasts DB2 Advisor with previous work on index recommendation. In Section 6, we give some preliminary performance measurements, both of the time for the algorithm to run and of the resulting execution time for workloads benefitting from the Advisor's advice. Section 7 discusses future work.

2. Architecture

At the highest level, the DB2 Advisor works as a black-box index-recommendation engine. The black-box has two inputs: a set of SQL statements known as the workload, and statistics describing the target database. There is only one output: the recommended indexes.

Architecturally, the DB2 Advisor consists of:

Index SmartGuide A graphical user interface

db2advis A command-line driven utility for recommending indexes

Optimizer Extensions have been written into the DB2 Optimizer for the recommendation of indexes as well as their evaluation

Advise tables These new tables are created for the purpose of advising, and they are used as a communication vehicle between db2advis and the Optimizer

The preferred method of invoking the Index Advisor is through the Graphical User Interface called the Index SmartGuide. We have included here two screen snapshots of the Index SmartGuide. The screen snapshot in Figure 2 shows how the user can specify a workload of statements. The SmartGuide automatically searches for SQL statements and their frequency of execution in the SQL cache and imports them. Effectively, the DB2 dynamic SQL cache stores recently-executed SQL statements. The Index SmartGuide also imports SQL statements from statically-compiled SQL statements, which are known as packages in DB2 terminology. Other sources of SQL statements include the Query Patroller load scheduling product, and recently explained SQL statements. Lastly, statements can be entered manually or using cut-and-paste. The workload is stored in a user-owned table, called ADVISE_WORKLOAD.

In other windows of the SmartGuide (see the tabs at the top), the user may optionally specify constraints on the estimated disk to be consumed by all indexes recommended,

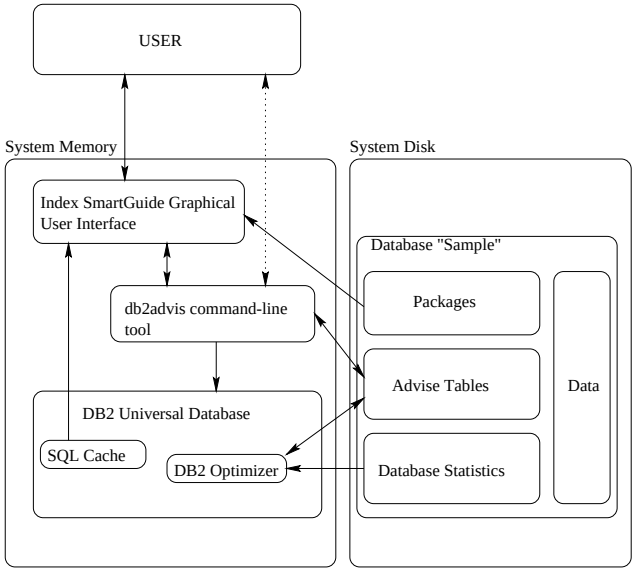

Figure 1. Architecture of DB2 Advisor

or on the maximum time for DB2 Advisor to spend improving its recommendations. For example, the user can require DB2 Advisor to work for no more than 5 minutes, and to recommend that all indexes consume no more than 5 Gigabytes.

The SmartGuide then calls the db2advis utility, an application program that contains the major optimization logic of DB2 Advisor. For each statement in the ADVISE_WORKLOAD table, it invokes the DB2 UDB Optimizer in one of two new EXPLAIN modes that either RECOMMEND_INDEXES or EVALUATE_INDEXES. The Optimizer stores the indexes it recommends in another user-owned table, called ADVISE_INDEX. The screen snapshot in Figure 3 shows the index recommended by db2advis for the workload of Figure 2. By clicking on the "Show workload details" button, the user can see how much the recommended indexes will benefit each statement in the workload.

Alternatively, the user may invoke the db2advis utility directly from the command line, providing options for specifying the database, the workload of SQL statements, the constraints, and various other options. Example 1 shows the invocation of db2advis for a single SQL statement in the "sample" database. In less than two seconds, DB2 Advisor determines the best indexes to create and the estimated improvement in the execution time if they were created, as well as the DDL to create them.

EXAMPLE 1:

```
$ db2advis -d sample -s "select * from
t1,t2 where t1.c1 = t2.c2"
  execution started at timestamp 1999-07-
06-19.02.32.617867
Calculating initial cost (without recomm-
```

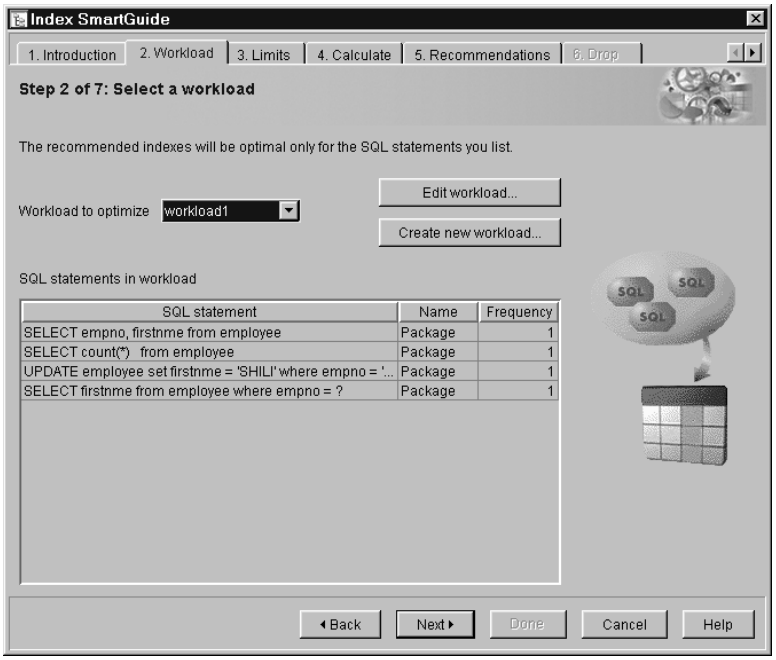

Figure 2. Specifying a workload of statements

```
mended indexes) [82.237053] timerons
Initial set of proposed indexes is ready.
Found maximum set of [2] recommended in-
dexes
Cost of workload with all indexes included
[25.879040] timerons
total disk space needed for initial set [2]
MB
total disk space constrained to [-1] MB

2 indexes in current solution
  [ 82.2371] timerons (without indexes)
  [ 25.8790] timerons (with current solu-
tion)
  [%68.53] improvement
  Trying variations of the solution set.
--
-- execution finished at timestamp 1999-07-
06-19.02.34.154307
--
--
-- LIST OF RECOMMENDED INDEXES
-- ===========================
-- index[1], 1MB
CREATE INDEX WIZ0 ON "VALENTIN"."T2" ("C2"
ASC) ;
-- index[2], 1MB
CREATE INDEX WIZ2 ON "VALENTIN"."T1" ("C1"
ASC) ;
-- ===========================
--
Index Advisor tool is finished.
```

The algorithm for the index-recommendation engine in db2advis is covered in the next two sections. The first section will discuss the simple case of recommending indexes for a single SQL statement. The subsequent section extends the algorithm to accommodate for a workload of queries.

3. Single query optimization

The algorithm for recommending indexes is an extension of the existing process for optimizing an SQL query in the DB2 Compiler. The old process is augmented with the injection of a multitude of ”virtual indexes” - hundreds of indexes whose metadata has been temporarily introduced into the schema only for the duration of the optimization process.

To illustrate the approach, suppose that all possible indexes were temporarily injected into the schema model. The DB2 Compiler would then be faced with its usual optimization process, except that there would be a lot more indexes in the schema to consider. When the optimization process has completed, the DB2 Compiler produces the optimal Query Access Plan. If this plan contains one or more virtual indexes, then these indexes are the recommended indexes. Effectively, we let the optimizer choose which indexes it likes.

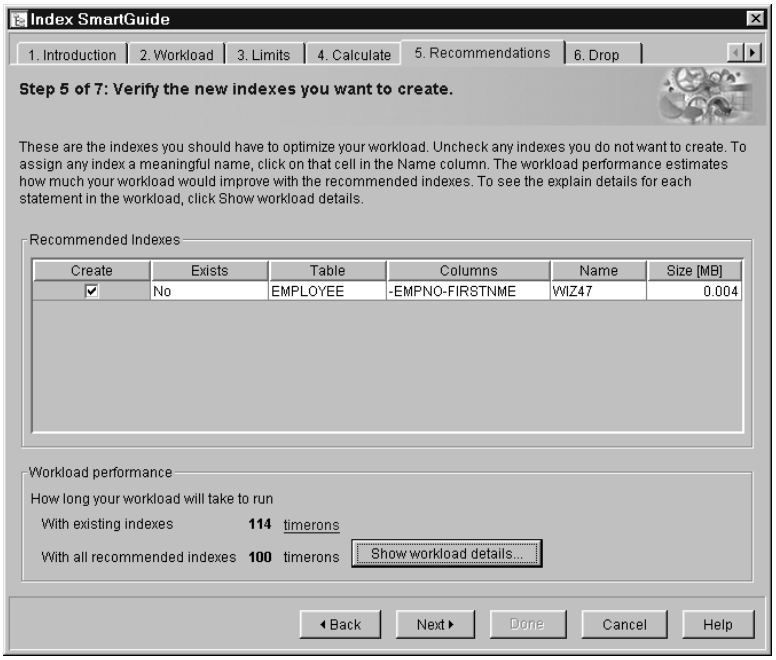

Figure 3. View the recommended indexes

In practice, there are problems with this approach. The most immediate issue is that the enumeration of all possible indexes produces a working set which is too big. In DB2, a table with n columns can support a very large number of indexes, as shown by Formula 1.

Formula 1 (Number of Possible Indexes) *Given a table with n columns, how many different indexes can exist containing k columns, where $k <= n$? There are n choices for the first column in the index. For the second column, there are $n-1$ remaining choices. As more columns are added, the total number becomes $(n)(n-1)(n-2)...(n-k+1)$ or $n!/(n-k)!$. Therefore the total number of indexes that can be created on a table with n columns is*

$$\sum_{k=1}^{n} \frac{n!}{(n-k)!}$$

However, in DB2 UDB, each column of an index may individually be defined as either ascending or descending. Therefore, for a given k, the space of possible indexes is multiplied by 2^k. As a result, we adjust our first formula to become:

$$\sum_{k=1}^{n} \frac{(2^k)n!}{(n-k)!}$$

Therefore, in practice, there has to be a limit on the number of virtual indexes enumerated. The DB2 Advisor limits the number of virtual indexes by using the DB2 Optimizer itself to suggest indexes intelligently, based upon its knowledge of how it wants to evaluate a given query. We call this approach the ”Smart column Enumeration for Index Scans” (SAEFIS) enumeration algorithm. This algorithm analyzes the statement predicates and clauses to produce sets of columns that might be exploited in a virtual index. There are 5 such sets:

EQ columns that appear in EQUAL predicates

O columns that appear in the INTERESTING ORDERS list. This includes columns from ORDER BY and GROUP BY clauses, or join predicates.

RANGE columns that appear in range predicates

SARG columns that appear in any predicates but nested subqueries or those involving a large object (LOB).

REF Remaining columns referenced in the SQL statement.

Then various combinations of (subsets of) these sets are formed, in order, eliminating any duplicate columns:

1. EQ + O
2. EQ + O + RANGE
3. EQ + O + RANGE + SARG
4. EQ + O + RANGE + REF

5. O + EQ
6. O + EQ + RANGE
7. O + EQ + RANGE + SARG
8. O + EQ + RANGE + REF

As a safety net to make sure the simplest indexes are not missed, and to evaluate how well the SAEFIS approach works, we have also implemented an algorithm to enumerate all possible indexes, stopping after a certain maximum number of indexes is reached. We call this the "Brute Force and Ignorance" (BFI) enumeration algorithm. There are several ways to implement this enumeration. We have taken a simple recursive algorithm and extended it to accommodate for ascending/descending columns.

Here is a pseudo-code version of the final recommendation algorithm:

ALGORITHM 1:
RECOMMEND_INDEXES(Statement S)

1. Enable "RECOMMEND INDEXES" mode
2. Enter the DB2 Optimizer
3. Inject the schema with virtual indexes using SAEFIS and generate their statistics
4. Inject the schema with virtual indexes using BFI and generate their statistics
5. Construct the best plan for S by calling the DB2 Optimizer
6. Scan the optimal plan, searching for virtual indexes
7. Submit these indexes back to the user as "recommended".

The essence of this algorithm is that the DB2 Optimizer both suggests candidate indexes and makes the decision on which indexes perform best. Importantly, both steps happen in a single call to the DB2 UDB engine. This approach has many advantages. The first advantage is that the efficiency of the recommendation process is maximized by entering the DB2 Optimizer (and hence the DB2 Engine) just once per single query. The second advantage is that no secondary or external optimizer is needed, either to suggest candidate indexes or to evaluate their cost. This reduces the maintenance of code that is redundant of the Optimizer's cost equations. Instead, by having the Optimizer itself simply inject likely-looking virtual indexes, and estimating their statistics, we have easily extended the DB2 Optimizer from an SQL Optimizer into an index selection optimizer. The last advantage is that the DB2 Optimizer does not need to be significantly modified. Once the virtual indexes are injected, the Optimizer continues working as it always has by enumerating plans, join orderings, and access methods. Only a small amount of code was written in order to enumerate virtual indexes and inject them into the schema.

Note that Algorithm 1 could be used as a subroutine within any existing Index Recommendation algorithm, not just our algorithm, which is detailed in Section 4 below. For example, it could be plugged in as a method for enumerating indexes in Daniel Zilio's Branch-and-Bound based method [Zilio 98] or in Whang's Drop-based method [Whang 85].

3.1. Index Statistics

Once the index columns are defined, the optimizer still requires statistical information about each virtual index. Without proper statistics, the optimizer will be unable to evaluate the cost of scanning an index, fetching selected rows from an index, or updating an index.

The statistics for virtual indexes are generated based on the corresponding table and column statistics, deducing information on index cardinalities, B+-Tree levels, and the number of leaf pages and non-leaf pages. Some properties cannot be deduced easily, such as clustering and uniqueness. For these properties, we assign pessimistic values. For example, we assume that there will be no clustering on the table per the index order. This behaviour allows the optimizer to be cautious as it uses virtual indexes, and avoid costing these indexes at performance levels which cannot be guaranteed.

The statistics for each virtual indexes are derived as follows:

Index Key Width, KW: the sum of the average width of each column in the index definition.

Index Clustering: none (worst-case value).

Index Density: none (worst-case value).

Percent Free: DB2 default, 15%.

Cardinality of an index with k columns, FKCARD:
$FKCARD = min\{CARD, \prod_{i=1}^{k} COLCARD_i\}$
where

CARD: cardinality of the table

COLCARD$_i$: cardinality (i.e. number of distinct values) of the ith column of the index

Number of Leaf Pages, NL: calculated from the index cardinality, page size, overheads for each key and page, assuming each page is fully packed with keys, using the following formula:

$KPP = \frac{PSIZE - POH}{KW + KOH}$

$NL = \frac{FULLKEYCARD}{KPP}$

where:

KPP: keys per page

PSIZE: page size (can be 4096 or 8192 bytes in DB2 UDB)

POH: page header overhead in a leaf page

KOH: key overhead.

Total Number of Non-Leaf Pages, TNL: calculated from number of leaf pages, key size, and page overhead as a recursive function. The recursion starts at the leaf level, and computes the number of pages at each level, continuing until the number of pages has reached one (representing the root node):

$$EPNL = \frac{PSIZE - NLPOH}{KW + NLEOH}$$

$$NL_{(0)} = FULLKEYCARD$$

$$NL_{(n)} = \frac{NL_{(n-1)}}{EPNL}$$

$$NLEVELS = n$$

$$TNL = \sum_{i=1}^{n} NL_{(i)}$$

where:

EPNL number of entries per non-leaf page

NLPOH page header overhead in a non-leaf page

NLEOH overhead of an entry in a non-leaf page

NL$_{(i)}$ number of non-leaf pages in level i

NLEVELS number of levels in the index

4. Workload Optimization

In this section, we will present the extensions to the algorithm that permit the DB2 Advisor to recommend indexes for a workload of statements.

Ideally, we would optimize the recommendation of indexes for a workload of statements in a single invocation of the DB2 Optimizer. There is a method of using an optimizer to work on several statements in one invocation, which is called Mass Query Optimization (MQO). Today, however, no commercial Relational Database product supports Mass Query Optimization, and therefore this was not an option for the DB2 Advisor.

As seen before in Figure 1, the DB2 Advisor has as a component a utility called db2advis. In this utility, we have added an index-selection algorithm which uses the results of the single-query recommendations as a starting point and searches for the optimal combination of indexes for a full workload.

The workload optimization algorithm contained in db2advis models the index selection problem as an application of the classic Knapsack Problem, a special type of 0-1 integer programming [GN 72]. Each index is an item that may or may not be put into the knapsack, as indicated by a variable for that index that can be 0 or 1 (a part of an index is useless). Each index also has an associated benefit and size. The benefit for an index is defined as the improvement in estimated execution time that an index contributes to all queries that exploit it, times the frequency that each query occurs in the workload. The size is just the estimated size of the entire index, in bytes. The knapsack has a fixed maximum size for all items in the solution. The objective is to maximize the benefit of all items in the knapsack. If the integrality constraint is relaxed, it is well known that the optimal solution accepts the entities into the knapsack in order of decreasing ratio of benefit to size, until the knapsack is full.

There are, however, a few complications in our straightforward application of the Knapsack Problem. First of all, we have relaxed integrality, but in reality it makes no sense to have a fraction of an index. Secondly, negative benefit accrues for updating each index in UPDATE, INSERT, and DELETE statements to that index's table. But at the time we compute the benefit for such statements, we don't yet know all the indexes that might be created by RECOMMEND_INDEX. Thirdly, we have attributed all the benefit resulting from a set of indexes to every index in a query. In reality, the benefit of each index is a function of what other indexes exist (i.e. the benefit of index A can differ when index B is present or absent), and attributing all the benefit to every index of the query is double-counting. This relates to the concept of "separability", discussed in the next section. To adjust for all of these complications, we refine the initial solution found by the Knapsack order in a routine called TRY_VARIATION, which creates a variant of the solution by randomly swapping a small set of indexes in the solution for a small set of indexes not in the solution. The workload is then re-EXPLAINed with this variant set of virtual indexes in the EVALUATE_INDEXES EXPLAIN mode. If the variant solution is cheaper overall, it becomes the current solution. TRY_VARIATION continues until the user's time budget has been exhausted.

Algorithm 2 describes the algorithm of db2advis for a workload W of SQL statements:

ALGORITHM 2:

1. GetWorkload W, including the frequency of execution of each statement.
2. R = $\emptyset$
3. For each Statement S in W,

(a) EXPLAIN S with existing indexes, returning S.cost_with_existing_indexes.

4. For each Statement S in W,

   (a) EXPLAIN S in RECOMMEND_INDEX mode, i.e. with virtual indexes

   (b) R = R ∪ RECOMMEND_INDEXES(S)

5. For each index I in R

   (a) I.benefit = S.cost_with_existing_indexes - S.cost_with_virtual_indexes

   (b) I.size = bytes in index

6. Sort indexes in R by decreasing benefit-to-cost ratio.

7. Combine any index subsumed by an index with a higher ratio with that index.

8. Accept indexes from set R until disk constraint is exhausted.

9. while (time did not expire) repeat

   (a) TRY_VARIATION

As stated before, the final step can be allowed to process for any length of time. This allows for flexibility in various cases: Where a feasible solution is needed quickly, the algorithm can be given less processing time; when obtaining an optimal solution is paramount, the algorithm can be given more processing time.

5. Comparison with Previous Work

Many papers have been written on this subject. The DB2 Advisor is unique because it can recommend indexes for an SQL statement within a single call to the RDBMS engine, using the DB2 Optimizer for the optimization.

Early designs for index recommendations started in the eighties [ISR 83], [BPS 90], [FON 92], [CFM 95], [GHRU 97], [CBC 93], [Whang 85]. These early papers had several shortcomings. First, they were restricted by existing technology. For example, none of these papers used an optimizer for cost estimates. One possible reason is that the existing optimizers would not externalize their cost estimates. These papers did, however, identify the nature of the problem as a variation on the classic Knapsack Problem.

Secondly, with the exception of [GHRU 97], all of these papers concerned themselves only with single-column indexes. [Whang 85] had an interesting addition, proposing a DROP optimization algorithm for the index selection problem, as opposed to a rule-based optimization.

Another weakness in these early algorithms was the assumption of separability. [Whang 85] made the case that index selection for each relation can be made independently of other relations. This assumption greatly simplifies the selection problem, but is this assumption correct? In fact, it is incorrect in many common cases. For example, in the case of a nested-loop join between two relations R and S, the presence of an index on relation R reduces the potential need for an index on relation S, and vice-versa, so long as one of the two relations has an index so that it can apply the join predicate on the inner relation. Obviously, this assumption is flawed.

Later solutions have used the RDBMS engine for evaluating solution sets, but never for recommending candidate indexes. The recommendation process always occurs in a module external to the RDBMS engine. These latter designs include [FST 88], [CN 98b], and [Zilio 98].

[Zilio 98] recognized the strong interdependence between indexes and partitioning keys. Zilio's implementation recommended partitioning keys as well as indexes. Zilio used a branch-and-bound optimization algorithm, which typically takes longer to find the optimal solution than the benefit-to-size ratio ordering of db2advis.

[CN 98b] was implemented in a commercial RDBMS, Microsoft SQL Server. Chauduri & Narasayya have made an essential contribution by combining the advantages of single-column recommendation with multi-column optimization algorithms. By considering index candidates with a small number of columns, they are more likely to optimize for several queries using the same candidate indexes, and still squeeze into small disk-constraints or small knapsacks. Taking this into account, their design starts by considering single-column indexes first, and working on wider indexes as time permits. Their goal was to reduce the number of optimizer invocations.

However, the same advantage of reducing the number of optimizer calls can be achieved by placing the enumeration algorithm inside the optimizer. That is the key to our implementation, and we believe it to be the better technique. The difference is most dramatic on a single-query basis, where our algorithm recommends indexes in a single optimizer invocation. Another advantage of this algorithm over [CN 98b] is the recommendation of wider indexes, intrinsic in the SAEFIS algorithm. The SAEFIS enumeration considers the three most likely uses of the index scan and combinations thereof. Yet another advantage is that the enumeration originates inside the DB2 engine, leveraging the existing optimizer, and thus reducing maintenance costs of two distinct optimizers.

6. Performance Measurements

There are several performance aspects that need to be addressed by DB2 Advisor. The first concern is the quality of the recommended indexes. How good are they? This is a difficult question to answer, but we have observed two cases where the DB2 Advisor has been used.

The first such case was with the TPCD workload, an industry benchmark for decision support. Running DB2 Advisor on the TPCD V1 workload showed that, in 14 out of the 17 queries, DB2 Advisor recommended indexes which performed optimally, or as near to optimal as is known to the DB2 TPCD team. In the remaining 3 queries, the DB2 Advisor missed some key indexes. The reason for this is that these indexes had to be defined as UNIQUE in order to take advantage of the improvement. But unfortunately we placed the restriction on the DB2 Advisor not to recommend UNIQUE indexes, because uniqueness is application-dependant and cannot necessarily be deduced from the existing data.

In another case, the DB2 Advisor was faced with a very complex machine-generated query that ran in over 48 hours. After the creation of three indexes recommended by the DB2 Advisor, the elapsed time reduced to 11 minutes. This shows the dramatic effect that automatic recommendation can have in those cases where a human eye is not available to analyze the incoming SQL, or the query is too complex.

Another aspect of operating performance is the execution time of DB2 Advisor. Because the DB2 Advisor can be interrupted at any time, then how much time should it be allowed to execute, before the recommendations are ”good enough”?

In order to answer this question, we exercised the DB2 Advisor against a 1GB TPCD database, with 6 levels of disk constraints. The results appear in Figure 4 and Figure 5. As the results in Figure 4 indicate, within 90 seconds, all levels of constraints had made a contribution to performance of 50% to 88%. This is reflected in the abrupt drop that occurs between 60 and 90 seconds. This improvement shows that much of the benefit of new indexes is achieved soon after the initial optimizer pass, as seen in steps 1 through 8 of Algorithm 2.

The benefits of allowing small permutations of that initial solution in step 9 of Algorithm 2 is seen in Figure 5, the detailed improvement chart. In this example, optimal indexes were found after 6 minutes, but the time to achieve optimality is very dependent upon the size and complexity of the workload.

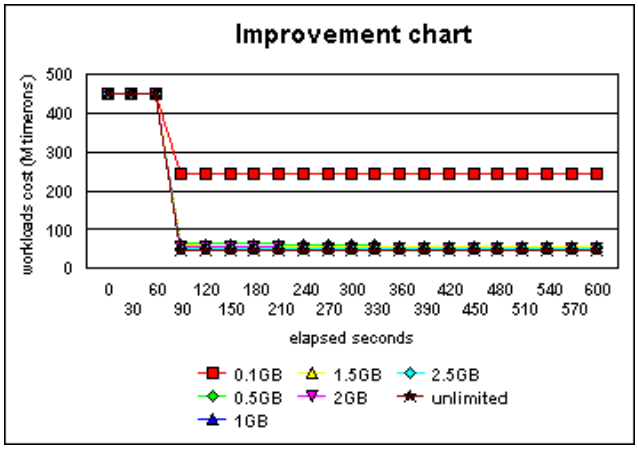

Figure 4. Quality of recommended indexes over time

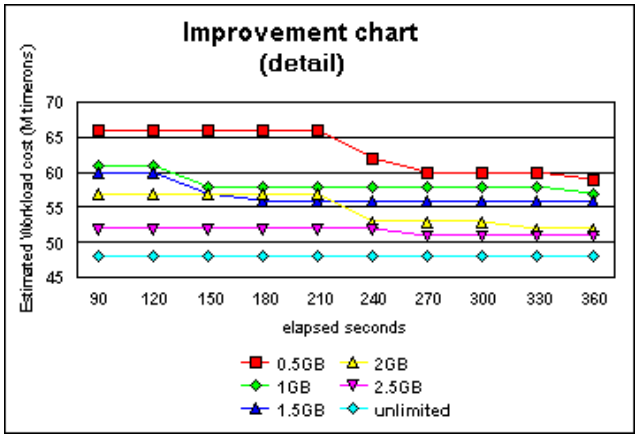

Figure 5. Quality of recommended indexes over time (detail)

7. Future Work

One of the strongest features of our algorithm relates to future work. One of the future directions for this project is

to extend this algorithm to the recommendation of materialized views and indexes on materialized views. Currently, the selection of materialized views in DB2 is performed mostly at the Query ReWrite level - not in the Optimizer. This means that the re-routing decisions are made using rules, rather than allowing the Optimizer to evaluate several alternatives according to estimated cost. However, there are efforts underway to allow more re-routing decisions to be made in the Optimizer, as well as efforts to add Mass Query Optimization to the DB2 Optimizer. This will provide the opportunity for more advanced advising that goes well beyond indexes.

Another future direction is to use this method for the recommendation of partitioning keys in a parallel database environment. It is possible to plug alternative partioning keys into the DB2 Optimizer and then evaluate them using an outside engine such as Daniel Zilio's Physical Design recommendation tool [Zilio 98]. At the heart of this technology is the efficiency of using the internal optimizer as much as possible. This avoids doing excessive calls in and out of the RDBMS, and avoids having to duplicate the optimizer's intelligence outside the engine.

We are looking at expanding the concept to include the suggestion of all database-related configuration: Including data layout, data properties such as referential integrity and constraints, partitioning keys, clustering, reorganization, and statistics collection.

Currently, this technology greatly simplifies the process of selecting a set of indexes. But the long-term goal of this technology is that a DBA will not even know what an index is, or what it is used for, and can concentrate on their primary concern: the creation and use of data.

8. Conclusion

The DB2 Advisor is unique in its use of a query optimizer for both suggesting and evaluating potential indexes. Using information that it must derive for optimizing a query anyway, the Optimizer can readily suggest much better candidates for new indexes than can an external routine that must repeatedly invoke the optimizer as it blindly iterates through the numerous possible combinations of columns for potential indexes. The DB2 Advisor suggests multi-column virtual indexes by combining columns from predicates, orders, and index-only access; estimates their attributes; and then evaluates them against other, existing indexes using its usual query optimization logic. Virtual indexes that are chosen by the optimizer are recommended to the user.

For workloads of multiple queries, this RECOMMEND INDEX mode is also used to determine the benefit of each such recommended index, by comparing the estimated cost for each query with and without these virtual indexes. The cost is simply the size of the index in bytes. Treating the index selection problem as an application of the well-known Knapsack Problem, the db2advis utility selects those indexes with the largest benefit-to-cost ratio, which is the optimal solution when the integrality constraint is relaxed. Selection continues until the cumulative size of all indexes chosen exceeds the disk constraint. The solution is refined by iteratively swapping a few indexes that are in the solution with those that are not, to account for the relaxation of integrality.

Both single-query and workload index selection by DB2 Advisor have been implemented in IBM's DB2 Universal Data Base Version 6.1. Performance evaluation has verified that it both finds indexes which significantly improve the execution of complex queries, and that the utility finds these indexes in a time proportional to the number of queries, but can continue to iteratively improve its recommendations.

We believe that exploiting a query optimizer in this way has tremendous potential for efficiently automating other aspects of database design. After all, the cost model of a query optimizer is a sophisticated mathematical model of how a query would perform, given the schema and physical attributes of the database. It therefore provides an ideal way to evaluate the impact of variations in the schema and/or its attributes. We are investigating additional ways for the DB2 Advisor to exploit the DB2 query optimizer to recommend and evaluate alternative database designs.

References

[BPS 90] Elena Barucci, Renzo Pinzani, and Renzo Sprugnoli, "Optimal selection of secondary indexes", IEEE Trans. on Software Engineering, 16(1):32-38, January 1990.

[CBC 93] Sunil Choenni, Henk M. Blanken, and Thiel Chang, "On the Selection of Secondary Indices in Relational Databases", Data & Knowledge Engineering, 11(3):207-233, 1993.

[CFM 95] Alberto Capara, Matteo Fischetti, Dario Maio, "Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design", IEEE Transactions on Knowledge and Data Engineering, 7(6):955-967, December 1995.

[FON 92] Martin R. Frank, Edward R. Omiecinski, and Shamkant B. Navathe, "Adaptive and Automated Index Selection in RDBMS", International Conference on Extendig Database Technology (EDBT), pages 277-292, Vienna, Austria, March 1992.

[GHRU 97] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman, "Index Selection

for OLAP", In Proceedings of the Internatoinal Conference on Data Engineering, pages 208-219, Birmingham, U.K., April 1997.

[ISR 83] Maggie Y. L. Ip, L. V. Saxton, and Vijay V. Raghavan, "On the Selection of an Optimal Set of Indexes", IEEE Transactions on Software Engineering, 9(2):135-143, March 1983.

[CN 98a] Surajit Chaudhuri and Vivek Narasayya, "AutoAdmin 'What-if' Index Analysis Utility", Procs. of the 1998 ACM SIGMOD Conf. (Seattle, 1998), pp. 367-378.

[CN 98b] Surajit Chaudhuri and Vivek Narasayya, "Microsoft Index Tuning Wizard for SQL Server 7.0", Procs. of the 1998 ACM SIGMOD Conf. (Seattle, 1998), pp. 553-554.

[Falkowski 92] Bernd-Juergen Falkowski, "Comments on an Optimal Set of Indices for a Relational Database", IEEE Trans. on Software Engineering 18,2 (Feb. 1992), pp. 168-171.

[FST 88] S. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical Database Design for Relational Databases", ACM Trans. on Database Systems 13, 1 (March 1988), pp. 91-128.

[GN 72] Robert S. Garfinkel and George L. Nemhauser, Integer Programming, John Wiley & Sons, New York (1972), pp214-241.

[Whang 85] Kyu-Young Whang, "Index Selection in Relational Databases", Proc. Intl. Conf. on Foundations on Data Organization (FODO) (Kyoto, Japan), May 1985, pp. 369-378. Also reprinted in Foundations of Data Organization, Sakti P. Ghosh, Yahiko Kambayashi, and Katsumi Tanaka (eds.), Plenum Press (1987), pp. 487-500.

[Zilio 98] Zilio, Daniel C., "Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems", PhD Thesis, University of Toronto, 1998.