Recommending Materialized Views and Indexes with the IBM DB2 Design Advisor

Daniel C. Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma
IBM Canada Ltd

Guy M. Lohman, Roberta J. Cochrane, Hamid Pirahesh, Latha Colby
IBM Almaden Research Center

Eric Alton, Dongming Liang, Jarek Gryz
York University

Gary Valentin
IBM Haifa Research Lab

Abstract

Materialized views (MVs) and indexes both significantly speed query processing in database systems, but consume disk space and need to be maintained when updates occur. Choosing the best set of MVs and indexes to create depends upon the workload, the database, and many other factors, which makes the decision intractable for humans and computationally challenging for computer algorithms. Even heuristic-based algorithms can be impractical in real systems. In this paper, we present an advanced tool that uses the query optimizer itself to both suggest and evaluate candidate MVs and indexes, and a simple, practical, and effective algorithm for rapidly finding good solutions even for large workloads. The algorithm trades off the cost for updates and storing each MV or index against its benefit to queries in the workload. The tool autonomically captures the workload, database, and system information, optionally permits sampling of candidate MVs to better estimate their size, and exploits multi-query optimization to construct candidate MVs that will benefit many queries, over which their maintenance cost can then be amortized cost-effectively. We describe the design of the system and present initial experiments that confirm the quality of its results on a database and workload drawn from a real customer database.

1. Introduction

Materialized views, or *Materialized Query Tables,* or *MQTs, as they are called* in the IBM® DB2® Information Management products, can improve query performance by orders of magnitude, by avoiding re-computation of a query's expensive operations, such as joins, sorts, etc. However, MQTs redundantly store data that is derivable from other data, so they consume extra disk space and must be updated to maintain their consistency with the source data whenever it changes, either periodically (*deferred or full refresh*) or as part of the same transaction (*immediate refresh*). Furthermore, an MQT, as any other stored table, requires its own indexes for efficient access. The benefit of an MQT relative to its cost is therefore maximized if the MQT benefits many queries, particularly costly queries, or frequently executed queries in the workload.

MQTs, therefore, add many new decisions and trade-offs that a database administrator (DBA) must make to optimize performance. What MQTs should be defined? What indexes on each MQT should be defined? Without the right indexes, accessing an MQT might be more expensive than re-deriving the answer from base tables that do. How often should each MQT be refreshed? What's the trade-off between an index on a base table and an index on an MQT? Solving these issues is too complex for a DBA, so we need to automate. The automation must select a set of MQTs that minimizes the total cost of query evaluation and MQT maintenance, usually under the constraint of a limited amount of resource such as storage space. The large decision space of this problem makes it computationally intractable [HRU96], so virtually all efforts to solve it have been heuristic. Logically, there are two distinct approaches:

- The MQTs are chosen first with some space constraint given, and then the indexes are chosen using the remaining space.
- There may be some iteration between MQT and index selection steps, or the steps are integrated into a larger one to suggest MQTs and indexes at the same time, but with other simplifying assumptions.

Although the second approach is algorithmically harder, it generally yields solutions with superior performance. We chose the integrated version of this second approach in our design.

This paper describes a new algorithm for simultaneously determining the optimal set of MQTs and indexes for a given workload, subject to a disk space constraint. It is an extension of the approach used in the IBM DB2 Index Advisor [VZZLS00], and has been implemented as such. The key idea of our approach is to extend the database engine's query optimizer in order to both: (1) suggest good candidate objects (MQTs and indexes), and (2) evaluate the benefit and cost of those candidate objects. Furthermore, pursuant to (1), we have extended the optimizer to exploit a sophisticated *Multiple Query Optimization* (MQO) technique [LCPZ01] that discovers common sub-expressions among a workload of queries for the definition of candidate MQTs. This approach significantly improves the benefit-to-cost ratio of MQTs that are recommended by the optimizer, and, therefore, of the final solution found by our algorithm.

The remainder of this paper is organized as follows. The related work is presented in Section 2. We describe the overall design for the DB2 Design Advisor in Section 3. Section 4 shows experimental results. Conclusions and future work are summarized in Section 5.

2. Related Work

Because of the power of MQTs, there has been much recent work on the problem of selecting which views to materialize, particularly in a data warehouse environment. [HRU96] provides algorithms to select views to materialize in order to minimize just the total query response time, for the case of data cubes or other OLAP applications, when there are only queries with aggregates over the base relations. In [GHRU97] these results are extended to the selection of both views and indexes in data cubes. [Gup97] presents a theoretical formulation of the general view-selection problem in a data warehouse. All of these papers present approximation algorithms for the selection of a set of MQTs that minimizes the total query response time under a given *space* constraint, but they ignore maintenance costs of those MQTs. Solutions presented in [RSS96, YKL97, TS97] provide various frameworks and heuristics for selection of materialized views in order to optimize the sum of query evaluation and view maintenance time, but without any resource constraint, and they ignore the interaction with indexes. [GM99] first addresses the view selection problem under the constraint of a given view maintenance time. The tools provided by RedBrick/Informix® [RedBrick] and Oracle 8i [Oracle] exclusively recommend only materialized views, ignoring their interaction with indexes [ACN00].

The idea of applying multi-query optimization to the view selection process has been recently explored in [MRSR01]. The main focus of that paper is the efficient maintenance of materialized views by determining new views to add, both permanent and transient (the latter being used only during the actual maintenance process). The authors do not consider any interaction with indexes, nor do they address how the initial set of views is chosen.

The prior work closest to ours is [ACN00]. It provides a tool to simultaneously select materialized views and indexes for a given SQL workload, subject to a disk space constraint, and provides a technique to find common sub-expressions among multiple queries in that workload. However, there are some important differences. First of all, [ACN00] limits the types of materialized views that can be generated to single-block views only; this excludes complex views for CUBE and ROLLUP in OLAP applications, for example. Our algorithm imposes no such syntactic conditions on the candidate views, and considers views that may be exploited for queries requiring "back-joins" (or "compensation"). Secondly, the algorithm in [ACN00] creates views that could be exploited by multiple queries by iteratively "merging" views that have been chosen by an earlier iteration of their algorithm. It is easy to find practical cases where this approach might never produce a good view for two queries because the component views were individually pruned as insufficiently cost-effective. Our algorithm creates common sub-expression views by invoking our MQO algorithm before any pruning is done. Thirdly, [ACN00] mentions the impact of maintenance costs on view selection, but has no explicit cost for maintenance in their cost metric. Our algorithm explicitly includes the cost of refreshing MQTs in its cost metric. Lastly, [ACN00] formulates its views and indexes outside the optimizer, whereas our approach uses the optimizer itself to generate candidates, thereby ensuring candidates that are guaranteed to be exploited by the optimizer and avoiding duplication of logic in both the optimizer and an external program.

Our work is based on [VZZLS00], which pioneered the concept of having the optimizer itself both suggest and evaluate candidate indexes. This paper extends this approach to include the simultaneous, interdependent selection of MQTs as well as indexes on both base tables and the MQTs, and to generate those MQTs using sophisticated multi-query optimization (MQO) techniques [LCPZ01]. We also allow for a wide range of MQTs to be selected and maintained. In particular, we support *full refresh* MQTs (which are updated periodically by the user) and *immediate refresh* MQTs (which are updated whenever the base tables are updated), and impose no restrictions on the complexity of the queries that define these MQTs.

3. Overview of the DB2 Design Advisor

The problem solved by the DB2 Design Advisor can be simply stated. For a workload of any number of SQL statements – which may include UPDATE, INSERT, and DELETE statements -- and (optionally) their corresponding frequency of occurrence, find the set of MQTs and indexes that minimize the total execution time of the entire workload, subject to an optional constraint on the disk space consumed by all indexes and MQTs. The DB2® Universal Database™ offer many ways to automatically gather the queries in the workload and their frequencies, including: (1) the cache of recently-executed queries; (2) the Query Patroller utility for controlling, scheduling, and prioritizing workloads; (3) statements whose plans have been analyzed by the EXPLAIN facility (and hence whose SQL content has been recorded); (4) static SQL that is embedded in application programs; and (5) user-provided (i.e., via cut and paste).

Since the DB2 Design Advisor relies upon the DB2 optimizer to estimate the cost of executing individual queries, it also requires the following information, which is automatically gathered by and already available to the optimizer:

- The database characteristics, including the schema, statistics such as the cardinalities of tables and columns, and other physical database design characteristics (indexes, partitioning, defined views, and materialized views).
- System configuration information, including: the estimated processing power of the processor; the latency, transfer rate, and seek times of the disk drives; and, in a shared-nothing multi-processor environment (DB2 UDB Extended Enterprise Edition), the number of nodes and the bandwidth of the interconnect network.

Mathematically, this problem is an example of the Knapsack problem, in which the objects to be included in the knapsack or not are candidate indexes and MQTs, each of which has a "benefit" (the improvement it engenders in the workload, less its maintenance cost) and a "cost" (the disk space it consumes). The objective function is to maximize the sum of the "benefit" of included objects, subject to a constraint on the "cost" (i.e., disk space) and an integrality constraint (partial MQTs or indexes contribute no benefit). By relaxing this integrality constraint, a provably optimal solution can be efficiently found (in *O(nlogn)* time) by ordering all objects in decreasing benefit-to-cost order, and filling the knapsack until the constraint is met [VZZSL00]. This solution to the relaxed problem is used to generate an excellent initial solution to the initial problem with the integrality constraint. Additional constraints must be introduced for indexes on MQTs, however, because it makes no sense to include in the solution indexes on MQTs that were excluded from the solution.

What is new in the DB2 Design Advisor is: (1) the optimizer must be extended to suggest good MQT candidates, and (2) the algorithm from the DB2 Index Advisor had to be significantly modified to deal with the interaction of MQTs and indexes. The overall design of this new algorithm is presented in Figure 1. Depending on what the user requests, the DB2 Design Advisor could output: (1) the recommended MQTs, (2) the recommended indexes, on base tables only,, or (3) both (1) and (2), including indexes on MQTs. In the first three steps, a set of good candidate objects (MQTs and indexes) is generated by invoking the DB2 optimizer under new EXPLAIN modes. To estimate the size (e.g., row width, cardinality, etc.) of each candidate object, the algorithm by default uses estimates provided by the optimizer, but optionally the data can be sampled to obtain much more reliable size estimates.

Next, we execute our selection algorithm to determine an optimal set of objects. This is where much of the work is done, starting with the initial solution suggested by the relaxed Knapsack solution, and trying different possible feasible solutions by invoking the DB2 optimizer to evaluate each solution for the entire workload. After the Knapsack algorithm, we have a random swapping algorithm as in {VZZLL00] to iterate between candidate objects that were not chosen before to find a better candidate set. The iteration continues either until a time limit is exceeded or we did not change the result in the last eight iterations. In the final "filtering" step, the algorithm examines the query plans that result with the resulting solution, and it removes any candidate indexes and MQTs not used anywhere in the workload. The filtering was needed as the knapsack algorithm potentially selects indexes or MQTs that compete in the same optimized query plan, and we run the winning candidates through the optimizer in the filtering to determine which subset was finally chosen. This last step is particularly useful when making selections for other database systems in a heterogeneous DBMS environment, whose behavior is somewhat unpredictable.

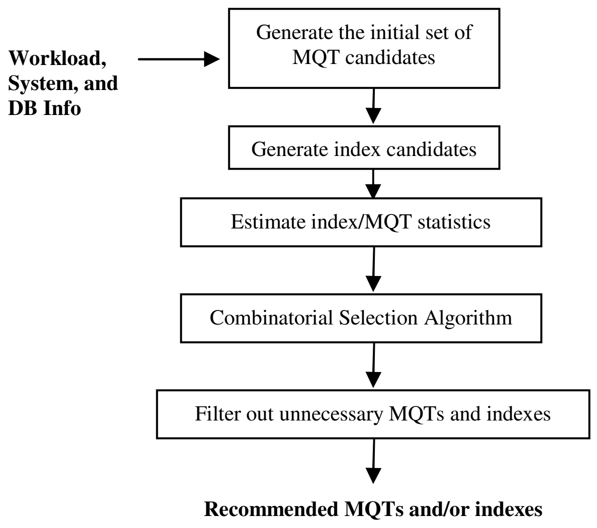
Workload, System, and DB Info
Generate the initial set of MQT candidates
Generate index candidates
Estimate index/MQT statistics
Combinatorial Selection Algorithm
Filter out unnecessary MQTs and indexes
Recommended MQTs and/or indexes

Figure 1. Design overview of the DB2 Design Advisor

3.1 MQT Candidate Generation

This module is used to generate an initial set of candidate summary tables. These candidates will be the set of MQTs used by the selection algorithm as input to determine the subset that gives the best performance benefits.

We generate MQT candidates using the following methods:

- Use the queries supplied in the workload themselves as candidate MQTs. For each query, all SELECT-FROM-WHERE-GROUP BY combinations that appear in the query are treated as MQT candidates (the GROUP BY is optional). For example, the simplest case is to define query Q itself as a candidate:

  CREATE SUMMARY TABLE <name> AS <Q>

  This is essentially what the Red Brick and Oracle tools do. We also use an algorithm that analyzes each query in the workload to change local predicates so as to generalize the MQT. An example of this would be to change the predicate "A=5" in a query to be "GROUP BY A" and add A to the SELECT clause.
- Use the (non-materialized) logical (shorthand) views defined by the user as candidate MQTs. Since users typically create logical views as shorthand for frequently referenced pieces of a query, they are likely to be referenced by multiple queries in the workload, and hence are excellent candidates for materialization. This option is easy to implement and inexpensive to execute, but requires that the user do much of the work.
- Utilize MQO (Multiple-Query Optimization) to suggest candidates. We exploit the sophisticated MQO techniques of [LCPZ01] for finding common sub-expressions among multiple queries. The internal graphical representations of all queries in the workload are merged together in a single graph. Using a bottom-up traversal, the operations between query blocks are matched level by level in terms of the objects referenced, predicates, grouping expressions, etc. Also, using the existing MQT matching techniques in DB2 UDB, suitable compensation for unmatched portions are added on top of the common sub-expression (CSE). As many CSEs may compete with each other, the MQO algorithm has various stages to reduce the sets of possible candidates. MQO includes generalizing local predicates in finding common expressions. We transform CSEs found in MQO to MQTs. One example of how MQO works is from using queries Q1 and Q2, which are shown below. MQO can detect the commonality in the following distinct queries, and uses the commonality to produce candidate MQTs that are usable by both queries. Note that MQT1 is derived from the common subquery, and MQT2 combines the two queries.

Q1: SELECT A FROM R WHERE B>5 AND C IN (SELECT D FROM S WHERE E<10)
Q2: SELECT A FROM R WHERE B<25 AND C IN (SELECT D FROM S WHERE E<10)
MQT1: SELECT D,E FROM S GROUP BY E
MQT2: SELECT A,B FROM R WHERE C IN (SELECT D FROM S WHERE E<10) GROUP BY B

In another MQO example, two queries Q3 and Q4 are matched. There is an extra table "Cust" in one query. If we assume there is a referential integrity relationship between Cust and Trans based on cust_id, the matching algorithm will use the join of all 3 tables as the CSE. The predicates and the grouping expressions are also matched and compensated accordingly and this allows us to suggest the CSE shown below as a suitable MQT candidate. Alternate candidates may involve the join of Trans and Store only.

Q3: SELECT store_name, cust_name, SUM(sales) as ss FROM Trans T, Store S, Cust C WHERE T.store_id = S.store_id AND T.cust_id = C.cust_id AND T.year = 2001 GROUP BY store_name, cust_name

Q4: SELECT store_name, year, SUM(sales) as ss FROM Trans T, Store S WHERE T.store_id = S.store_id AND T.year >= 1998 GROUP BY store_name, year

CSE: SELECT store_name, cust_name, year, SUM(sales) as ss FROM Trans T, Store S, Cust C WHERE T.store_id = S.store_id AND T.cust_id = C.cust_id AND T.year >= 1998
GROUP BY store_name, year, cust_name

Compensation for Q3:
SELECT store_name, cust_name, SUM(sales) as ss FROM **CSE** WHERE year = 2001
GROUP BY store_name, cust_name

3.2 The Selection Algorithm

As described earlier, the initial feasible solution is obtained by ordering all objects by decreasing *weight,* which is defined as "benefit" divided by "cost". High weight suggests that an MQT or an index with that weight is a good candidate for selection. The benefit of an MQT is the sum over all queries of the frequency of a query, times the performance change to the query (and/or update) when using the MQT as opposed to not using the MQT. (The weight of an index is defined in a similar way.) Formally, the weight *w(A)* of the MQT *A* is:

$$w(A) = \frac{\sum_{q \in Q} f(q) * B(q)}{D(A)}$$

where *Q* is the set of queries in the workload that make use of the MQT *A*, *f(q)* is the frequency of query *q*, *B(q)* is the performance change for the query *q*, and *D(A)* is the disk space consumed by *A*. These components are calculated as follows:

1) Use the standard EXPLAIN facility of the optimizer to compute the original, "status quo" estimated execution cost *E(q)* for each query q in the workload with existing MQTs and indexes.
2) Run each of the queries in the workload in a new EXPLAIN mode that adds the candidate MQTs as "virtual MQTs" to the existing MQTs. The usual exploitation of MQTs by the optimizer will determine the best MQT for that query from existing and candidate MQTs. The estimated cost *M*(*q*) from the resulting plan for query *q* will represent the minimal cost possible for that query, since it had all candidate MQTs to choose from.
3) Estimate the performance change of each query *q* as *B(q) = E(q) - M(q)*
4) Iterate through each MQT *A*, and compute its benefit by adding the performance change of any query *q* that uses that MQT. Its cost *D(A)* is its estimated size, i.e., its estimated cardinality times its width, in bytes.

As mentioned earlier, we allow the workload to contain both queries and updates. When more indexes and MQTs are introduced, queries perform better, but updates incur a performance penalty for maintaining them, either on each update statement (for indexes and immediate refresh MQTs) or periodically (for deferred refresh MQTs). The maintenance cost of updates and the benefit to queries when adding MQTs is included as part of our overall cost metric. The maintenance cost associated with an *immediate refresh* MQT can be determined in two ways:

- Use the estimated number of rows for all MQTs and the estimated frequencies and number of updated rows for each update statement to calculate a rough update cost. Computing this increases the number of optimizer calls in our algorithm by only *U*, where *U* is the number of update statements in the workload.
- For each MQT and update statement, determine the difference in estimated time for each update statement between using and not using the MQT. Computing the update cost this way would increase the number of calls to the optimizer in our algorithm by *U*(|A|+1)*, where |*A*| is the number of candidate views.

For each full refresh MQT A, we add to the "benefit" of A a penalty C(A) for materializing all of A with a frequency of g(A). A default setting for g(A) is 1, effectively including the cost of at least one materialization of A during the time interval for which we are optimizing. Thus, the weight for a full refresh MQT A, including maintenance, is therefore estimated by:

$$W(A) = \frac{\sum_{q \in Q} f(q) * B(q)}{D(A)} - g(A) * C(A)/D(A)$$

To recommend both indexes and MQTs together, our **ADD_COMBINE** algorithm (shown below) treats both indexes and MQTs as objects in the same Knapsack problem having a single space constraint. Note that indexes created on MQTs will be dependent on them; this dependency is recorded before entering the selection algorithm. The algorithm starts with the empty list of candidates. We repeat the following iteration until we exceed the disk space constraint. In each iteration, we try to add into the list the next object with the highest weight. To consider the dependence between an index and an MQT, if the object chosen is an index on an MQT, and that MQT is not in the chosen set, we add the MQT to the list, so long as it too fits within the space constraint. Time complexity of **ADD_COMBINE** is *O(nlogn),* where *n* is the number of candidate MQTs. This arises from the cost of sorting the candidate MQTs by weight.

```
Algorithm ADD_COMBINE
 /*  Input: A – a set of candidate objects (MQTs or
     indexes), space – the disk space for
     recommendations
     Output: S – the set of objects to be
     recommended */

FOR each object o (MQT or index),
     Calculate its weight W(o)
END FOR
Re-order the objects by descending weight W(o)
S = { }
WHILE (space > 0 && A is not empty) DO
      Pick o from A  such that o has the highest
         weight W(o)
      IF  W(o)  <  0  THEN space = 0
      ELSE
            C = { o }
            IF (o is an index on MQT a) && ($a \notin S$)
            THEN
              /* Add in the MQT a, since it is not in
                 the solution yet  */
              $C = C \cup a$
            END IF
            IF ( space - D(C) ) >= 0  THEN
              /* There is space for the objects in C,
                 so add them to the solution. */
              space  =  space – D( C )
              $S = S \cup C$
              $A = A - C$
            END IF
      END IF
END WHILE
Perform iterative random swapping phase where
   MQT and index winners in S are randomly
   swapped with other candidates in A
RETURN S as the set of objects picked by
   ADD_COMBINE
```

After the selection of the initial feasible solution by the above algorithm, the selection algorithm uses an iterative improvement phase that randomly swaps a few MQTs that are in that solution with a few of those that are not, as was done in [VZZLS00]. We execute this phase for either a user-specified time limit, or until we cannot improve the workload performance further (over a given number of consecutive swaps). This phase exists to compensate for three simplifying assumptions in our algorithm that could lead to sub-optimal solutions. First, the performance benefit for a query Q is assigned to all candidate indexes and MQTs used by the query. This does not account for how much each candidate itself contributed to the performance improvement. As such, the benefit assigned to each candidate is optimistic. Second, the weighting of competing candidates is calculated independently. There is no concept of weighting an MQT A, say, given that another MQT, B, was not selected. Third, the ordering by cost/benefit weights is provably optimal only for the relaxation of the general knapsack algorithm, but may be sub-optimal under the original integrality constraint. The *random swap phase* for this combined selection algorithm is more complicated than in [VZZLS00] when we swap a handful of chosen items with a handful of not chosen items. If a new index is swapped in, we also have to add in any MQT on which that index depends. For similar reasons, if an MQT is swapped out, the indexes that depend on it also have to be swapped out.

4. Experiments

We tested our algorithm to determine whether MQO provided useful MQT candidates, and to measure the overall quality of its results. The experiments used a simulation of an IBM DB2 UDB customer's database that has an OLAP-type (star) schema. Though we did not have access to the customer's proprietary data, we could simulate their database and environment by importing its schema and statistics. The originating system was an IBM DB2 UDB EEE system with 10 processing nodes, which is used to keep track of product sales information for the company in various markets, in various geographic territories, for various customers, and for different times. There were more than 15 tables in the database, comprising roughly 400 GB of data. The workload we used contained 12 OLAP-like business queries to the star schema, each of which had an equal frequency of execution.

The purpose of our first set of experiments was to: 1) measure the actual performance improvement from MQTs recommended by the Advisor; and 2) compare the benefit of MQO candidates with the benefit of candidates drawn from both MQO and the queries of the workload. To do this, we first executed the advisor using MQT candidates drawn from both MQO and directly using the queries of the workload. In the second run, we limited the MQT candidates to be only ones from MQO. Each invocation of the Advisor was limited to at most 5 minutes.

The results of these runs are shown in Table 1. These results include the total workload estimated times (WETs) with no MQTs, the total workload estimated times when MQTs are recommended by the Advisor, the percentage improvement in the estimated run time gained by exploiting the recommended MQTs, and the number of MQTs recommended by the Advisor. The results indicated that our Advisor recommends a small set of high-quality MQTs that improve performance by almost 30%. The results also indicate that MQTs derived by our MQO algorithm provided most of this benefit, and MQTs from queries provided surprisingly little improvement.

Type of MQT Selection	WET without MQTs	WET with MQTs	% diff in WETs	Num. of MQTs
MQTs from MQO and queries	493.7 seconds	353.0 seconds	28.5%	7
MQTs from MQO only	493.7 seconds	352.0 seconds	28.4%	4

Table 1: Experiment output for MQT-only selection.

Our second experiment selected MQTs and indexes together. The purpose of this experiment was to: 1) measure the combined performance improvement from MQTs and indexes recommended by the Advisor; and 2) demonstrate that selecting MQTs with indexes provides even greater performance improvements than just selecting only MQTs. In this experiment, the WET without recommendations was 493.7 seconds, as before. The Advisor recommended 7 MQTs and 18 new indexes, 8 of which were on the recommended MQTs. For this recommended configuration, the WET was only 51.4 seconds, for an 89% improvement. The fact that the Advisor recommended indexes on base tables as well as candidate MQTs indicates that the algorithm correctly handled the interaction of indexes and MQTs. And exploiting indexes as well as MQTs recommended by the Advisor gives far better performance than just MQTs alone, as can be seen by comparing the results of our second to those of our first experiment.

5. Conclusion and Future work

This paper has presented a novel and efficient algorithm for simultaneously determining the set of MQTs and indexes that minimize the total execution cost for a given workload of SQL statements, given a single disk space constraint for both. The execution cost explicitly includes the maintenance cost that is due to updates, as well as the time to execute queries. We also detailed extensions to the database engine's optimizer that exploits sophisticated multi-query optimization techniques to suggest superior candidate MQTs that will benefit multiple queries. Incorporating these techniques in the optimizer saves duplicating the optimizer's cost model in the design tool, and ensures consistency with the optimizer's choice of MQTs and indexes when the recommended items are subsequently created. By modeling the MQT and index selection problem as a variant of the well-known Knapsack problem, the DB2 Design Advisor is able to optimize large workloads of queries in a reasonable amount of time. The new algorithm, and the extensions to perform multi-query optimization in the optimizer, have been implemented in IBM DB2 Universal Database (UDB) for Linux, UNIX®, and Windows®, , and have demonstrated significant improvements in the workloads tested to date.

References

[ACN00] S. Agrawal, S. Chaudhuri, V. Narasayya. Automated Selection of Materialized Views and Indexes for SQL Databases. *VLDB*, 2000.

[BDD+] R. G. Bello, K. Dias, A. Downing, et al. Materialized Views In Oracle. In *Proc. VLDB*, 1998.

[BLT86] J. A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. *SIGMOD*, 1986.

[CN99] S. Chaudhuri and V. Narasayya. Index Merging. *ICDE*, 1999.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection in OLAP. *ICDE*, 1997

[GM95] A. Gupta, and I. S. Mumick. Maintenance of Materialized views: Problems, Techniques, and

Applications, *Data Engineering Bulletin*, June 1995.

[GM98] A. Gupta, and I. S. Mumick. Editors. *Materialized Views*: MIT Press. 1998.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. *ICDT*, Delphi, Greece, 1997.

[GM99] H. Gupta, and I. S. Mumick. Selection of Views to Materialize under a Maintenance Cost Constraint. *ICDT*, Jerusalam, January 1999

[HGW+95] J. Hammer, H. Carcia-Molina, J.Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehouse Project. *IEEE Data Eng. Bulletin, Specail Issue on Materialized Views and Data Warehousing*, 1995.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. *SIGMOD*, June 1996.

[Kim96] Kimball, R., *The Data Warehouse Toolkit*, Join Wiley, 1996.

[KR99] Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouse. *SIGMOD*, 1999

[LCPZ01] W. Lehner, B. Cochrane, H. Pirahesh, M. Zaharioudakis. Applying Mass Query Optimization to Speed up Automatic Summary Table Refresh. *ICDE* 2001.

[LQA97] W. J. Labio, D. Quass, and B. Adelberg. Physical Database Design for Data Warehouse. *ICDE*, 1997.

[MRSR01] H. Mistry, P. Roy, S. Sudarshan, K. Ramamritham. Materialized View Selection and Maintenance Using Multi-Query Optimization. *SIGMOD* 2001.

[Mum95] I. Mumick . The Rejuvenation of Materialized Views. In *CISMOD*. 1995.

[Oracle] http://www.oracle.com/

[R+95] N. Roussopoulos, et al. The ADMS project: Views "R" Us. In *IEEE Data Engineering Bulletin*, June 1995.

[RedBrick] http://www.informix.com/informix/solutions/dw/redbrick/vista/

[RK86] N. Roussopoulos and H. Kang. Preliminary design of ADMS+-: A workstation-mainframe integrated architecture for database management systems. *VLDB*, 1986.

[Rou91] N. Roussopoulos. An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis. *ACM TODS*, 1991.

[RSS96] K. A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. *SIGMOD*, 1996.

[SSV96] P. Scheuermann, J. Shim, and R. Vingrlek. WATCHMAN: A Data Warehouse Intelligent Cache Manager. *VLDB*, 1996.

[TS97] D. Theodoratos and T. Sellis. Data warehouse configuration. *VLDB*, 1997.

[VZZLS00] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes. ICDE, San Diego, CA, Feb. 2000.

[WB97] M. C. Wu, A. P. Buchmann. Research Issues in Data Warehousing, *BTW*'97, Ulm, March 1997.

[Wid95] J. Widom. Research problems in data warehousing. In *Proc. Of the Intl. Conf. On Information and Knowledge Management*, 1995.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. *VLDB*, 1997.

[ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. ACM *SIGMOD* 1995, pages 316-327.

Trademarks