Dynamic Materialization of Query Views for Data Warehouse Workloads

Thomas Phan [1], Wen-Syan Li [2]

[1] *Yahoo!, Inc.*
thomas.phan@acm.org
Work done while with IBM Research

[2] *IBM Almaden Research Center*
wsl@us.ibm.com

***Abstract*— A materialized view, or Materialized Query Table (MQT), is an auxiliary table with precomputed data that can be used to significantly improve the performance of a database query. Previous research efforts have focused on finding the best candidate MQT set, with a common static heuristic being to greedily pre-materialize the MQTs prior to executing the workload. While this approach is sound when the size of the MQT set on disk is small, it will not be able to pre-materialize all MQTs and indexes when faced with real-world disk limits and view maintenance costs, and thus a static heuristic will fail to exploit the potentially large benefits of those MQTs not selected for materialization. In this paper we present an automated, dynamic MQT management scheme that materializes views and creates indexes in an on-demand fashion as a workload executes and manages them with an LRU cache. In order to maximize the benefit of executing queries with MQTs, the scheme makes an adaptive tradeoff between the MQT materializations, the base table accesses, and the benefit of MQT hits in the cache. To find the workload permutation that produces the overall highest net benefit, we use a genetic algorithm to search the N! solution space, and to avoid materializing seldom-used MQTs, we prune the set of MQT candidates. We ran our dynamic management on a TPC-H workload and found that our scheme produces higher benefit across a variety of scenarios; we demonstrate over 60% improvement in MQT benefit under harsh conditions when the size of available MQTs with indexes is larger than the cache.**

I. Introduction

A materialized view, or Materialized Query Table (MQT), is an auxiliary table with precomputed data that can be used to significantly improve the performance of a database query [1], [2]. With its MQT matching capability, a database query optimizer can explore the possibility of reducing the query processing cost by appropriately replacing parts of a query with existing and matched MQTs.

Because MQTs are required in OLAP (Online Analytical Processing) applications in which the query workloads tend to have complex structure and syntax, a Materialized Query Table Advisor (MQT Advisor) is often required to recommend MQTs (e.g. [3], [4]). In the rest of this paper, when we refer to an MQT, we will assume that it includes both materialized views (with their indexes) as well as indexes on base tables. This approach is consistent with other work such as [5].

An MQT Advisor takes a workload (the read and write queries to the database system) and the database space size allocated for MQTs as the input. It first performs workload compression to remove those insignificant queries which are inexpensive or infrequent and multi-query optimization [6] to derive common parts in the workload and generates candidate MQTs. The MQT Advisor then: calculates the benefits of these candidate MQTs in terms of resource time reduction; calculates the overhead (in terms of resource time) for refreshing/updating MQTs and then estimates their sizes; calculates the utility of each MQT by dividing net benefit (i.e. benefit minus overhead of creating the MQT) by the size of the MQT and its index size; and finally recommends the MQTs whose utility values are higher than a given threshold. In this paper, we limit our scope to analytical workloads and assume that the MQTs recommended by the MQT Advisor are read-only.

Many commercial database vendors have an MQT Advisor product, such as those in Microsoft SQL Server [3], [7], IBM DB2 [4], [8], and Oracle 10g [9]. These advisors typically deploy a static approach to managing MQTs: the MQTs are quantitatively evaluated and then greedily pre-materialized prior to executing the workloads. While this approach is sound when the size of the MQT set on disk is small, it will not be able to materialize all MQTs when faced with real-world constraints (such as disk space limits, view maintenance costs, or time spent on MQT matching during query compilation) and thus will fail to exploit the potentially large benefits of those MQTs not selected for materialization.

Previous industry and academic research efforts in this area have concentrated on the aspect of finding the best candidate MQT set to pre-materialize during the MQT Advisor stage. Our work takes a complementary approach. *Instead of pre-materializing MQTs in our system, we provide an automated MQT management scheme that uses an LRU cache to dynamically materialize and drop the MQTs as a workload executes. To achieve a high cache hit rate, we permute the query order of the workload, and to avoid materializing seldom-used MQTs, we prune the set of MQT candidates managed by our system.*

Figure 1 shows how our approach fits into an existing database framework. Our system's key components are an MQT cache and a Dynamic MQT Scheduler (DMS) that operates on the candidate MQTs recommended by an existing MQT Advisor. To maximize the benefit of executing queries with cached MQTs, the DMS must make an adaptive tradeoff between the cost of MQT materializations, the cost of accessing base tables in lieu of the MQTs, and the benefit of MQT cache hits. We use the term "cache" to refer to disk space or

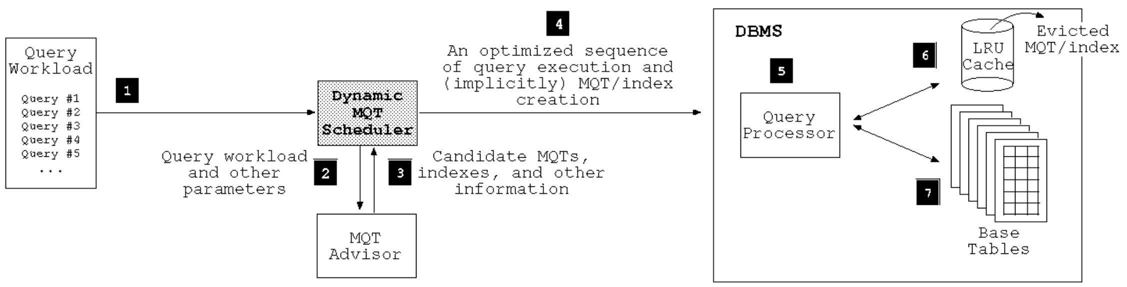
Query Workload
Query #1
Query #2
Query #3
Query #4
Query #5
...
1
Dynamic MQT Scheduler
4
An optimized sequence of query execution and (implicitly) MQT/index creation
Query workload and other parameters
2
3
Candidate MQTs, indexes, and other information
MQT Advisor
DBMS
5
Query Processor
6
LRU Cache
Evicted MQT/index
7
Base Tables

Fig. 1. A high-level overview of the relationship between our dynamic MQT management system and an existing DBMS framework. (1) A query workload is submitted to our Dynamic MQT Scheduler (DMS). (2) The DMS gives the queries to an MQT Advisor product, which (3) returns a set of candidate MQTs and associated indexes. (4) The DMS runs a search algorithm against possible query order permutations, producing an optimized sequence of queries and MQT/index materializations and drops (which are dictated by LRU semantics). (5) This optimized sequence is executed by the query processor, which runs the queries against (6) MQTs which are materialized into and evicted from an LRU cache as well as (7) the base tables. Previous research has focused on improving the effectiveness of the MQT Advisor stage.

the database. Also, an "MQT cache hit" refers to the event where a query is routed to an MQT that has been materialized in the cache; "MQT cache miss" is defined in the same terms.

To maximize the net benefit of MQT usage, we implemented two key features to help manage the MQTs. First, to get a high number of cache hits, the DMS permutes the workload's query order, and to find the permutation that produces the overall highest benefit, the DMS uses a self-adapting genetic algorithm [10] to search the $N!$ permutation solution space. (Query order permutation further differentiates our work from caching done in other domains, such as memory-level caching).

Second, for large workloads, the MQT Advisor may actually recommend too many MQTs, including MQTs that are not used very often; while dynamically materializing MQTs may improve the cache hit rate, it also causes these non-beneficial MQTs to be built, so to reduce the likelihood of this event, we further prune the set of MQTs recommended by the MQTA by performing a binary search on the size of this set.

The overall result is that the DMS generates an optimal schedule that dictates both the query execution order and (indirectly by way of the LRU cache) the MQT/index creation order. We ran our scheme against an extended TPC-H workload of 133 queries, and our results show that our dynamic MQT management scheme produces higher benefit across a variety of scenarios compared with the typical static approach deployed in existing advisors.

The key contributions of this paper are the following:

- We identify the mutually-beneficial combination of dynamic MQT management, query order permutation, and intelligent MQT set pruning (Section II).
- We describe implementations of the above features using, respectively, an LRU cache, a genetic search algorithm, and a binary search pruning heuristic (Section III).
- We provide two dynamic MQT management schemes that can handle batch workloads. The **dynamic simple** scheme aggressively materalizes all MQTs into the cache, while the **dynamic pruned** scheme intelligently reduces the materializations by pruning the MQT set. They do not require modifications to the query processor and can complement an existing DBMS (Section III).
- We measure our dynamic schemes against a typical static scheme and show that our approach is adaptive to varying conditions (Section IV).

	Without query reordering		**With query reordering**
Static MQT pre-materialization	(1) **Current state-of-the-art** (0/1 knapsack problem)	≅	(2) **Current state-of-the-art** (0/1 knapsack problem)
Dynamic MQT materialization and replacement	(3) Complementary to our work (future work)		(4) **Research work presented in this paper**

Fig. 2. Classification of MQT management scenarios. The current state-of-the-art in commercial DBMSes is to pre-materialize MQTs; with this approach, query reordering makes no difference, so (1) and (2) are equivalent. Our work falls into (4), where dynamic MQT materialization and query reordering provide the most flexibility.

II. Scope of Problem

In Figure 2, we classify MQT management scenarios into four categories based on two conditions, namely whether or not query workloads can be reordered and whether or not MQTs can be materialized and replaced dynamically. Note that we use the term MQT to refer collectively to materialized views, materialized views and their indexes, and indexes on base tables.

Using current MQT advisors, such as described in [4], MQTs are recommended and created to fill the disk space that users specify. The recommended MQTs (and their indexes) are materialized in advance before workloads are executed. During the execution of the workloads, the MQTs are not replaced. Because the MQTs are fixed during the workload execution, whether or not the workload is reordered will not make any difference. These are scenarios (1) and (2); they can be considered knapsack problems.

When the size of the candidate MQT set is small and can fit on disk, a static pre-materialization strategy is sound. However, not all beneficial MQTs should be pre-materialized and maintained all the time due to several real-world considerations:

- With a smaller set of MQTs, the application can perform better by loading all MQTs into the main memory, thus reducing the I/O cost of storing the MQTs on disk.
- The view maintenance cost may be too high: MQTs in the cache or in the database still need to be refreshed, so when there are many MQTs in the cache, these MQTs may have a high overhead.
- Having too many MQTs available could make compilation expensive because the optimizer must match the

queries with all candidate MQTs.

Scenarios (3) and (4) in Figure 2 represent *dynamic* materialization and replacement of MQTs. In (3) there is no possibility of workload reordering, so we materialize an MQT, say MQT_x, as long as its net benefit (i.e. total benefit minus materialization cost) is positive before it is replaced by another MQT_y at T_0. This scenario is addressed in [5] where materializations are interleaved among queries and in DynaMat [11] where query results are cached.

Scenario (4) more specifically considers the possibility of reordering the query workload. If there are new queries that arrive after T_0 and can benefit from MQT_x, we may want to move these new queries ahead so they can execute before MQT_x is swapped out. Therefore, in scenario (4) we are given the highest flexibility for managing MQTs to minimize query workload response time. In this paper, we handle scenario (4) by focusing on dynamic MQT management, query reordering, and intelligent MQT set pruning, a mutually-reinforcing combination not considered by other research (e.g. [5]).

Our work models three MQT management approaches:

- In the **static model**, a subset of the candidate MQT set is greedily chosen and then pre-materialized before the workload begins so that the subset can be used throughout the duration of the workload. If a query requires an MQT that was not pre-materialized, the query must access the base table. This greedy approach represents the current state-of-the-art in existing commercial DBMSes and corresponds to box (1) in Figure 2.
- In the **dynamic simple model**, all MQTs in the candidate MQT set are managed through an LRU cache. If a query requires an MQT that is not in the cache (i.e. upon cache miss), the MQT is materialized on-demand. If there is not enough room in the cache, existing MQTs may be dropped following LRU rules.
- The **dynamic pruned model** is a compromise between the previous models. MQTs are still managed via an LRU cache as in the simple model, but these MQTs are taken from a pruned subset of all the candidate MQTs. If a query requires an MQT that is not in this pruned subset, then the query must access the needed base table.

The dynamic models correspond to box (4) in Figure 2. As we show in our experimental results later, the dynamic simple model is too aggressive because it can dynamically materalialize *all* MQTs, including those that are not used often enough to warrant being materalized. This issue is addressed in the dynamic pruned model, which reduces materializations by reducing the set of possible cache-managed MQTs.

Our dynamic approach has the following characteristics:

- To the best of our knowledge, ours is the first work to identify the mutually beneficial use of dynamic MQT management, query re-ordering, and MQT set pruning.
- Our work applies not only to the situation where disk space is an issue but also to where maintaining a large number of MQTs has high overhead.
- The implementation of the dynamic schemes (in particular the dynamic pruned) can be outside of the core query processor and can be applied to existing DBMSes. Other work such as [5] and [12] require modifications to query processing, MQT matching, or cache management.
- Since we reorder the query, our approach has not only a higher hit rate but also lower MQT maintenance costs: MQTs can be dropped as soon as they are used or MQTs can be marked with a shorter time-to-live.

III. Design Approaches

A. General MQT management model

At a high level, the quantitative benefit that a query derives from an MQT is a measure of how much the MQT improves the execution time of the query. This benefit must take into account the cost to materialize the MQT as well as the savings of running the query against the MQT as opposed to the base tables. With this in mind, we define the following variables:

- M: the number of MQTs.
- N: the number of queries.
- B_{ij}: the benefit of MQT i seen by query j.
- B_j: the benefit of all MQTs seen by query j.
- r_{ij}: the probability that MQT i is used by query j.
- $hitprob_i$: the probability that MQT i is materialized.
- $missprob_i$: the probability that MQT i is not materialized ($missprob_i = 1 - hitprob_i$).
- $hittime_i$: the time to run a query *with* MQT i (that is, when the query has been routed to MQT i).
- $misstime_i$: the time to run a query *without* MQT i.

We modeled a query workload's execution and interaction with the MQTs as follows. The workload is represented as a queue of N queries. Our Dynamic MQT Scheduler then takes as input a generated list of M candidate MQTs that an MQT Advisor product has previously computed to be beneficial to the workload. More strongly stated, we assume that the MQT Advisor product (whether a part of DB2, SQL Server, or Oracle) improves the workload by recommending only those MQTs that have a positive benefit. If a potential MQT has a negative benefit (that is, an MQT with a high enough materialization cost that outweighs the MQT's use), then this MQT will not even be recommended. Furthermore, when the MQTA is run, we assume we can turn off its disk space limit enforcer to allow the MQTA to generate all possible positive-benefit MQTs.

The number of queries N is typically larger than the number of MQTs M (for example, in our experiments using an extended TPC-H workload, $N = 133$ and $M = 42$). We further assume that the MQTs are read-only (since we are dealing with analytical workloads) and that the queries are mutually independent. Note that the mapping between the queries and MQTs is many-to-many: each query can use multiple MQTs, and each MQT can be used by multiple queries. The queries in the workload are executed sequentially in the order that they appear in the queue. For each query, the assigned MQTs on disk are in turn accessed sequentially.

The expected value of the benefit that all the MQTs provide for query j is given by a weighted average of the individual MQT benefits for query j:

$$E[B_j] = \sum_{i=1}^{M} r_{ij} \cdot B_{ij} \qquad (1)$$

We assume for simplification that the benefit B_{ij} that MQT i provides for query j can be decomposed into its constituent parts as follows. We note that $hitprob_i$ and $missprob_i$ are affected by the management models (as we discuss soon), but $hittime_i$ and $misstime_i$ are known ahead of time since they are outputs of the analysis performed by the MQT Advisor.

$$B_{ij} = missprob_i \cdot misstime_i - hitprob_i \cdot hittime_i \quad (2)$$

This equation can be rewritten in terms of $hitprob_i$ by using the fact that $missprob_i = 1 - hitprob_i$:

$$B_{ij} = misstime_i - hitprob_i(hittime_i + misstime_i) \quad (3)$$

We can combine equations 1 and 3 into:

$$E[B_j] = \sum_{i=1}^{M} r_{ij} \cdot (misstime_i - hitprob_i(hittime_i + misstime_i)) \quad (4)$$

The expected value of the MQT benefit for any query can be found using the central limit theorem since we can assume that the number of queries and MQTs are sufficiently large (again, we had N=133 queries and M=42 MQTs):

$$E[E[B_j]] = \frac{1}{N}\sum_{j=1}^{N} E[B_j] \quad (5)$$

Our goal is to maximize the expected value in equation 5, so we focus on maximizing $E[B_j]$ from equation 4. As mentioned earlier, $hittime_i$ and $misstime_i$ are properties of the MQTs themselves and were already determined by the MQT Advisor; in fact, the SQL subquery which makes up the MQT was probably already optimized by the query optimizer when the MQT Advisor ran. The probabilities r_{ij} are also a fixed property of the query workload that were decided by the MQT Advisor. The remaining variable is the hit probability $hitprob_i$ that MQT i will be accessed. The differences between the $hitprob_i$ distributions among the static, dynamic simple, and dynamic pruned models capture the respective strengths and weaknesses of these schemes.

B. Static model of MQT management

In the static model commonly used by current commercial databases, a highly skewed MQT hit probability distribution is formed from a methodically greedy approach to choose MQTs to materialize. In this scheme, a subset of the candidate MQTs is chosen to be pre-materialized before the workload is executed. The remaining MQTs not chosen at this point are never used. The static model is shown in Figure 3.

Heuristics to choose the MQTs to place into the candidate MQT set have been the focus of much previous research outside the scope of this paper. Indeed, as we mentioned earlier, we take as input the candidate MQT set produced from an MQT Advisor product.

For the purposes of characterizing a typical implementation of the static model, we assume that that the candidate MQTs produced from the MQT Advisor are typically first scanned and then sorted by decreasing benefit. The difference $misstime_i$-$hittime_i$ is the benefit of one use of MQT i. The benefit of an MQT is then simply the sum of all its benefits

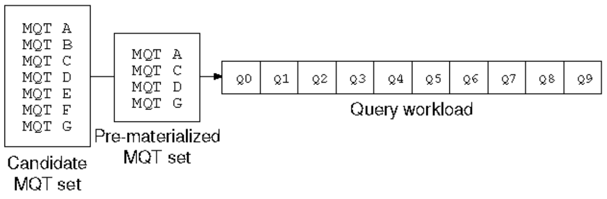

Fig. 3. Static model of MQT management.

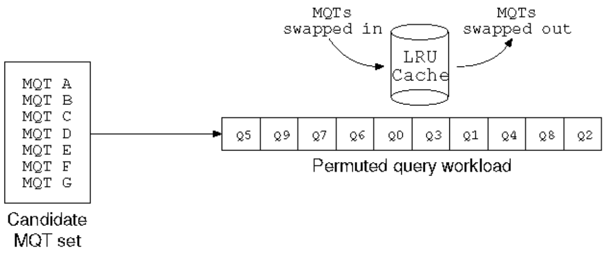

Fig. 4. Dynamic simple model of MQT management.

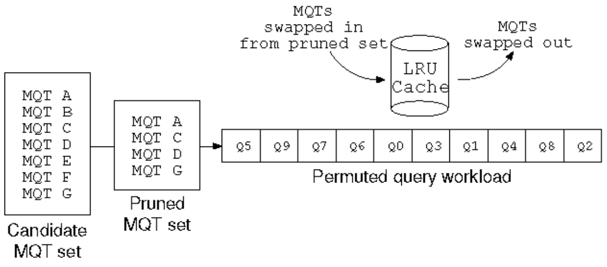

Fig. 5. Dynamic pruned model MQT management.

across all queries in the workload. The MQTs are subsequently sorted based on their calculated benefit.

Given a list of these MQTs sorted on benefit score, the system greedily materializes the MQTs in decreasing benefit order until a disk usage limit is reached. In our experiments we set the disk limit on the order of GBytes. These pre-materialized MQTs are kept on disk throughout the execution of the workload. If a query requires an MQT that has not been materialized, the query must access a base table, which can have a substantially negative impact in workload performance.

It can be seen that in the static model, the hit probability $hitprob_i$ that MQT i is accessed by they queries is highly irregular among the MQTs. The MQTs chosen to be pre-materialized will always have a hit probability of 1.0, whereas those not pre-materialized will have a hit probability of 0.0. *The static approach thus will fail to exploit the potentially large benefits of those MQTs that were not selected for materialization.*

C. Dynamic models of MQT management

In our dynamic models, the MQTs are not pre-materialized. Instead, they are materialized on-demand when queries execute and are managed in an LRU cache. Such an approach makes a tradeoff between the negative cost of materialization time and the positive benefit of MQT hits in the cache which obviate the need to access base tables. With respect to the *hitprob* distributions of the MQTs, the use of an LRU cache provides non-zero hit probability to all the candidate MQTs (in the case of the dynamic simple model) or to a subset of the candidate

MQTs (in the case of the dynamic pruned model). The overall result is that the dynamic models provide more hits across all the MQTs, as will be shown in Section IV.

The rationale for the dynamic models can be seen with the following intuitive example. Suppose a workload comprises five queries, each of which access the same MQT. The time to materialize the MQT is 2000 units, the time to execute the query with the MQT is 100 units, and the time to execute the query without the MQT is 500 units. If all five queries execute without the MQT, the workload execution time is $500 \times 5 = 2500$ units. On the other hand, if the MQT is used and materialized on-demand in the cache, then the execution time is $2000 + 100 \times 5 = 2500$, which represents a one-time materialization cost and five successive hits in the cache. It can be seen that if there are six or more hits of the MQT in the cache, then the on-demand materialization approach provides greater benefit than running the workload without the MQT materialized because MQT hits in the cache makes accessing the base tables unnecessary. *In the case of the static approach, not having a needed MQT available can be likely if the disk limit was already reached during the pre-materialization phase.*

The dynamic models execute by sequentially running the queries in the queue. Each query has its own MQT set, and for each MQT in the set, the MQT's benefit is calculated based upon whether or not the MQT is present in the cache. If there is a hit, the MQT is accessed. If there is a miss, the MQT is materialized at that moment and placed into the cache. If an MQT must be removed from the cache to make room, eviction follows LRU policy; however, if a cached MQT is to be evicted but is in the set of required MQTs for the current query, then the MQT is kept in the cache.

In the **dynamic simple model**, shown in Figure 4, all the MQTs in the candidate MQT set are managed via the LRU cache: MQTs are *always* materialized on demand when there is cache miss. However, for large workloads, the MQT Advisor may produce too many MQTs. Having too large a candidate MQT set causes problems because the dynamic simple approach is too aggressive in its materializations: for some particular MQTs, no global query ordering could be found that allowed these MQTs to be hit enough times in the cache to warrant their multiple materializations and evictions. The end result for these MQTs is that their net benefit across the workload is negative.

We thus also considered a **dynamic pruned model**, shown in Figure 5, where only a pruned subset of the candidate MQT set is managed in the cache. Pruning the MQT set allows us to reduce the total number of materializations while still providing the advantage of dynamic MQT management. In the case where a query requires an MQT not in the pruned set, the query accesses the base table.

The pruned subset is chosen by following the algorithm in Algorithm 1. At line 10, the workload with the candidate MQT set (suggested by the MQTA) is initally estimated once with a genetic algorithm (as described in the next subsection). A resulting negative net benefit of the MQTs implies that some MQTs are being materialized but are not used often

Algorithm 1 Candidate MQT set pruning

```
 1: FUNCTION Prune
 2: IN: $W$, workload queue
 3: IN: $M$, candidate MQT set suggested by MQTA
 4: OUT: $Pruned$, pruned candidate MQT set
 5: BEGIN
 6:
 7: MQTSet $Pruned := M$
 8: int $netbenefit :=$ GeneticAlgorithm($W$, $Pruned$)
 9: while $(netbenefit < 0) and (Pruned.size > 1)$ do
10:     sort $Pruned$ by descending benefit
11:     $Pruned :=$ top half of $Pruned$
12:     $netbenefit :=$ GeneticAlgorithm($W$, $Pruned$)
13: end while
14: return $Pruned$
15: END
```

enough. In this case, a binary search is performed on the size of the candidate MQT set (spanning lines 12 to 16). For these reduced sizes, the candidate MQTs are sorted by their net benefit from the previous estimation round and are selected for the current pruned set based on decreasing order. As we show later in Section IV, the dynamic pruned model consistently produces a better query workload execution than either the static model (which produces too many base table accesses) or the dynamic simple model (which produces too many materializations).

In both dynamic models, the hit probability distribution for each MQT is a consequence of the workload's query order. Ordering is important due to the cache's LRU replacement policy: it is desirable to have as many cache hits as possible, but this situation requires MQT accesses be grouped together to exploit temporal locality before MQT eviction.

Given these observations, the complexity of maximizing the benefit attained via the dynamic models reduces to the problem of finding the optimal permutation of the workload queue that produces the highest expected benefit. Note that although we use an LRU cache to manage the MQTs, this selection of replacement policy is only a matter of choice: because the common nature of replacement policies is to exploit locality of reference in the access stream, the fundamental problem is finding an optimal permutation of the workload that can take advantage of whatever policy is being used.

With a queue size of N queries, there are $N!$ permutations to search, and the resulting combinatorial search against this solution space is NP-hard. We next describe our use of a genetic search heuristic for finding the optimum permutation.

D. Permutation search with a genetic algorithm

Given a search space of $N!$ permutations of the query workload, the problem is to find the optimal query order that produces the highest MQT benefit. To search this solution space, we use a genetic algorithm (GA) search heuristic [10], [13] that finds a tradeoff between maximizing MQT cache hits, minimizing MQT materializations, and minimizing base table accesses. The GA's output is a query ordering that also indirectly controls the MQT/index materialization and drop order (which are dictated by the LRU behavior).

A GA emulates Darwinian natural selection by having population members compete against one another over successive

Algorithm 2 Genetic search algorithm

```
1: FUNCTION Genetic algorithm
2: BEGIN
3: Time $t$, Population $P(t)$
4:
5: evaluate($P(t)$)
6: while ! done do
7:     recombine and/or mutate P(t)
8:     $t := t + 1$
9:     select the best $P(t)$ from $P(t-1)$
10:    evaluate($P(t)$)
11: end while
12: END
```

generations in order to converge toward the best solution. As shown in Algorithm 2, chromosomes evolve through multiple generations of adaptation and selection.

We note that we use a GA simply as a tool to find a solution in the combinatorial problem. There are many possible approaches to finding a good solution, such as local-search stochastic hill-climbers and global-search simulated annealing, but we chose a GA due to our familiarity with the framework. Since we are trying to find an optimal query permutation, potential solutions can be naturally formed as a string of integers, a well-studied representation that allows us to leverage prior GA research in effective chromosome recombination (e.g. [14]). Furthermore, it is known that a GA provides a very good tradeoff between exploration of the solution space and exploitation of discovered maxima [13].

GA execution. A GA proceeds as follows. Initially a random set of chromosomes is created for the population. The chromosomes are evaluated (hashed) to some metric, and the best ones are chosen to be parents. In our problem, the evaluation produces the net benefit of executing the workload, accessing MQTs, and materializing/evicting MQTs in the cache. The parents recombine to produce children, simulating sexual crossover, and occasionally a mutation may arise which produces new characteristics that were not available in either parent. In our work, we further implemented an adaptive mutation scheme whereby the mutation rate is increased when the population stagnates (i.e. fails to improve its workload benefit metric) over a prolonged number of generations. The children are ranked based on the evaluation function, and the best subset of the children is chosen to be the parents of the next generation, simulating natural selection. The generational loop ends after some stopping condition is met; we chose to end after 1000 generations had passed, as this value was empirically the best tradeoff between execution time and thoroughness of the search. Note that converging toward and finding the global optimum is not guaranteed because the recombination and mutation are stochastic.

GA recombination and mutation. The chromosomes are permutations of unique integers. Figure 6 shows a recombination of chromosomes applied to two parents to produce a new child using a two-point crossover scheme [14]. Using this approach, a randomly chosen contiguous subsection of the first parent is copied to the child, and then all remaining items in the second parent (that have *not* already been taken from the first parent's subsection) are then copied to the child

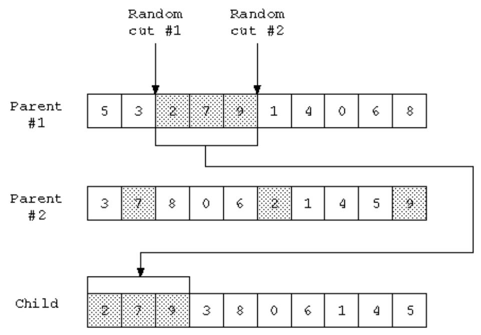

Fig. 6. An example showing how a child chromosome is produced from parents using two-point crossover recombination. Each chromosome represents a permutation string. To maintain the requirement that all the integers must be unique in the string, the recombination first takes a contiguous subsection of the first parent (chosen as the piece between two randomly chosen slices), places this subsection at the start of the child, and then picks all remaining values in the second parent not included from the first.

in order of appearance. The uni-chromosome mutation scheme chooses two random items from the chromosome and reverses the elements between them, inclusive. We will look at other recombination and mutation schemes in the future.

GA evaluation function. An important GA component is the evaluation function. Given a particular chromosome representing one workload permutation, the function deterministically calculates the net benefit of using MQTs managed by an LRU cache during the workload. The evaluation function follows the pseudocode in Algorithm 3. There are three key branches, namely if an MQT is in the candidate MQT set that have been reduced by pruning (line 13), if there was a cache hit (line 18), and if there was a cache miss (line 20). Each of these events results in the appropriate penalty. Note that the evaluation function can be easily replaced if desired; for example, other evaluation function can model different cache replacement policies or execution of queries in parallel.

IV. Experiments and Results

To show the benefit of the dynamic MQT management models in improving workload execution time, we ran the GA implementation of these models along with the static model in several experimental scenarios. We implemented our Dynamic MQT Scheduler using standard C++ and ran it with the publicly-available versions of the IBM DB2 database and MQT Design Advisor. DB2 ran on an AIX PPC workstation, while our DMS ran on a Fedora Core Linux box running at 2.8GHz with 1 GB of RAM. The candidate MQT list produced by the MQT Advisor was fed into our DMS, which in turn issued its resulting optimized schedule to DB2.

A. *Experimental setup*

We ran our experiments with a TPC-H workload that was extended from an original set of 22 queries to 133 queries, with the added 111 queries defined by a Business Intelligence (BI) performance team[1]. These additional queries were defined specifically to simulate BI applications with complex queries and result in an average query processing cost equivalent to

[1]This workload is available for other researchers. Please email us if interested.

Algorithm 3 GA evaluation function

```
1:  FUNCTION evaluate
2:  IN: $W$, workload queue
3:  IN: $M$, candidate MQT set
4:  OUT: $netbenefit$, execution net (w/MQT vs. w/o MQT)
5:  BEGIN
6:  LRUCache $cache$, Query $query$
7:  MQTSet $mqtset$, MQT $mqt$
8:  int $netbenefit$ := 0
9:  $cache$.empty()
10: for (each $query \in W$) do
11:     $mqtset$ := $query$'s set of MQTs
12:     for (each $mqt \in mqtset$) do
13:         if ($mqt$ ! $\in$ $M$) then
14:             {$mqt$ is not in the candidate MQT set. This is relevant to the
                pruned algorithm.}
15:             $netbenefit$ -= query time without mqt
16:         else
17:             $result$ := $cache$.access($mqt$)
18:             if ($result = hit$) then
19:                 $netbenefit$ += (query time without mqt -
                    query time with mqt)
20:             else if ($result = miss$) then
21:                 $netbenefit$ -= (mqt materialisation cost -
                    query time with mqt)
22:             end if
23:         end if
24:     end for
25: end for
26: return $netbenefit$
27: END
```

that of the original 22 queries. We used this extended TPC-H set to allow the MQT Advisor to recommend more data to be materialized, resulting in a bigger challenge since it represents a larger search space in which to find an optimal solution. Specifically, for the 133-query workload, the MQT Advisor recommended 21 MQTs and 21 base table indexes, whereas for the 22-query workload, the MQT Advisor recommended only 9 MQTs and 26 base table indexes. To further allow the MQTA to over-recommend the candidates (so we can search through more of them), we turned off the MQTA's disk space limit. The total amount of TPC-H data was 20GB. Each query accessed between 1 to 3 MQTs on average.

The GA ran using the values for MQT materialization cost, query execution without MQTs, and query execution with MQTs that were all produced by the MQT Advisor. These are the measurements that constitute the $hittime$ and $misstime$ variables from Section III. We generated the candidate MQT set by allowing the MQT Advisor to produce MQTs without disk limit. When the GA ran in our trials, we varied the MQT cache as explained next.

B. Cache load factor

An important term that we use throughout the experiments is the *cache load factor*, which is akin to a hash table's load factor. We define the cache load factor to be the size of the candidate MQT set (in bytes) divided by the size of the cache (in bytes). For example, if the load factor is 3.0, then the candidate MQT set is 3 times larger than that of the cache. If the load factor is less than 1.0, then all the candidate MQTs can fit into the cache. For a given cache size, n candidate MQTs, and the individual sizes of the MQTs, we calculate the cache load factor as follows:

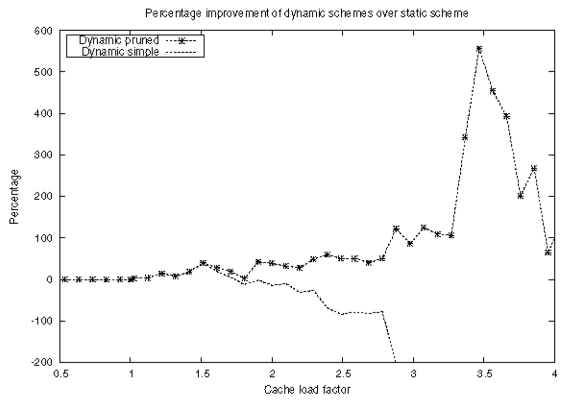

Fig. 7. The percentage improvement in MQT benefit for the dynamic pruned and dynamic simple schemes over the static scheme. MQT benefit is the net difference in execution time between running a scheme with MQTs and running without MQTs.

$$Cache\ load\ factor = \frac{\sum_{i=1}^{n} size\ of\ MQT_i}{cache\ size}$$

Note that we use the term "cache" for both the static and dynamic models; in the former, the cache is the set of pre-materialized MQTs created before the workload executes, while in the latter, the cache is the LRU-guided storage.

C. Results

1) Improvement of dynamic MQT schemes: In this experiment we compared the three MQT management schemes. For each scheme, we measured the 133-query workload execution while running with MQTs. The net benefit of a given scheme was then computed to be the improvement in running time with the scheme over execution of the workload without any MQTs. In Figure 7 we graph the percentage improvement of the two dynamic schemes over the static scheme as a function of increasing cache load factor. Along the x-axis, an increasing cache load factor represents an increasing likelihood of a cache miss because a higher load factor indicates that there are more MQTs than can fit into the cache. In this experiment and those that follow, all results were averaged over 50 runs.

Several observations can be made about this graph. First, when the cache load factor is less than 1.0, the dynamic models provide no improvement over the static model; this result is due to the fact that in this region, all the candidate MQTs can fit into the cache, so reordering the queries in the workload has no effect. Second, for the region between approximately 1.0 and 1.7, the dynamic simple model provides a modest improvement over the static model (reaching 39% improvement at around a cache load factor of 1.5), but at around 1.7, its falls off dramatically and becomes much worse than the static scheme; as we show later, this performance is due to its aggressive on-demand materialization of all MQTs, including those MQTs that are not used often enough to outweigh their materialization cost. Third, the dynamic pruned model is consistently better than the static model (with approximately an 18% improvement with a cache load factor of 2.0 to about a 62% improvement at 4.0) and significantly better than the dynamic simple model. The sudden spikes

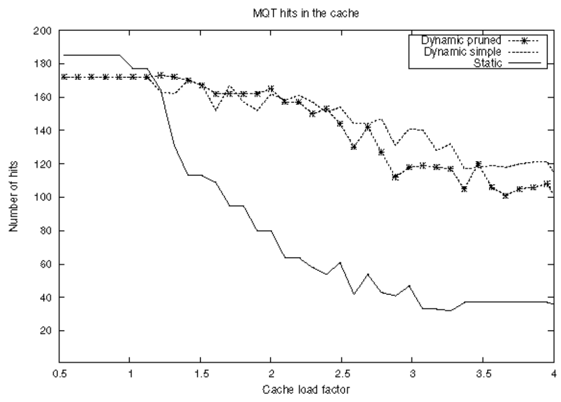

Fig. 8. The number of cache hits for the dynamic and static models. For the dynamic models, the hits are in the MQT cache, whereas for the static algorithm, the MQT hits are on the pre-materialized MQTs. Figures 8 to 10 are from the same experiment as Figure 7.

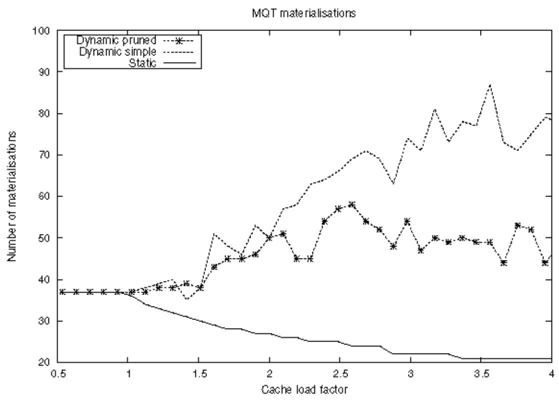

Fig. 9. The number of MQT materializations for the dynamic and static models. The dynamic models' materializations occur during workload execution. The static model's materializations are the pre-materializations.

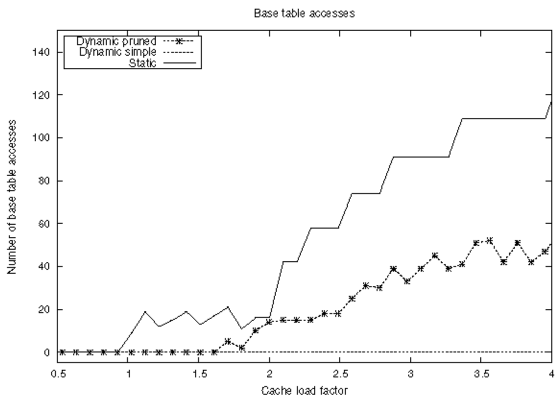

Fig. 10. The number of base table accesses for the dynamic and static models. The dynamic simple model never accesses the base tables because MQTs are always materialized on cache miss. On other hand, a miss in the static model always results in a base table access.

are due to the relatively large running time of the static algorithm that occur in bursts at high cache load factors; we are continuing to study this behavior. The dynamic pruned model's improvement is due to the fact that it reduces the set of available candidate MQTs managed in the cache and thus makes unnecessary the materialization of nonbeneficial MQTs as was explained in Section III-C.

2) Differences between dynamic and static models: Figure 8 graphs the hits for the same experiment from the previous subsection. To reiterate, a cache hit corresponds to a query being routed to an MQT that exists in the cache. This figure displays the inherent differences in the cache hit characteristics of the schemes, as mentioned in Section III. The static model starts off with more hits because it begins with MQTs already pre-materialized in its cache. The dynamic simple model registers more hits than the dynamic pruned model because the latter has fewer candidate MQTs that can be materialized.

Although the dynamic and static models share the same trend of decreasing hits with an increasing cache load factor, their behaviors diverge due to their treatment of cache misses. The impact of cache misses on materialization is shown in Figure 9. For the static model, the number of materializations decreases because materialization occurs only during the pre-materialization phase. For the dynamic simple model, materializations increase dramatically, revealing its aggressive materialization behavior on every cache miss. Finally, for the dynamic pruned model, materializations are not as aggressive, as expected, due to its pruned candidate set.

The base table accesses are shown in Figure 10. As mentioned, a cache miss in the static model incurs a base table access, resulting in a substantial increase with an increasing number cache load factor. The dynamic simple model never accesses the base tables. Again as expected, the dynamic pruned model finds a compromise.

The general observation from these graphs is that the three models lie on different regions of the MQT management spectrum with the difference between them being most evident under harsh conditions when the cache load factor is high. The static model, with fewer available pre-materialized MQTs, produces a large number of base table accesses. The dynamic simple model never accesses the base tables, resulting in a large number of materializations. The dynamic pruned model falls between the two extremes and finds a mix of base table accesses and materializations that results in better performance than the other two, as was shown in Figure 7. Furthermore, the GA can find a permutation that increases the number of hits and reduces the number of misses, thereby providing a net benefit consistently higher than that of the static model when the cache load factor is greater than 1.0.

In the future we look to improve the dynamic pruned model further. We anticipate that instead of performing a binary search to find the optimum size of the reduced candidate MQT subset, we can encode this parameter into the GA chromosome itself and allow it to naturally converge toward an optimum.

It is important to note that the performance and cache results shown in this and the previous subsection are based on one set of experimental parameters, the most important of which are the MQT net benefit and the MQT materialization cost. As we discuss in the next two subsections, varying these two parameters produces the same trends but with different absolute results.

3) Effect of varying MQT benefit: Here we compared the workload performance produced by the dynamic pruned model and the static model when the net benefit of the MQTs varies. An individual MQT can produce a higher net benefit if the query execution time with the MQT decreases with respect to

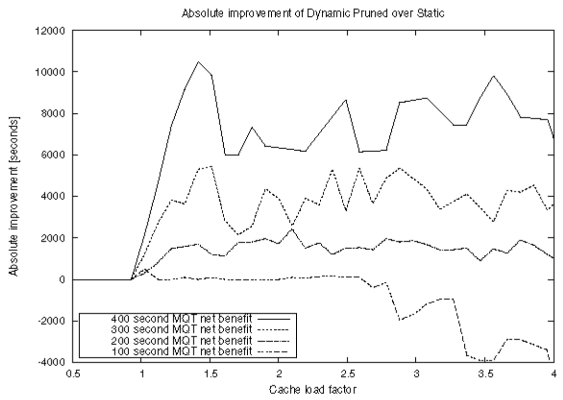

Fig. 11. Synthetic workload execution showing the absolute improvement of the dynamic pruned model over the static model. The four curves represent different MQT net benefits.

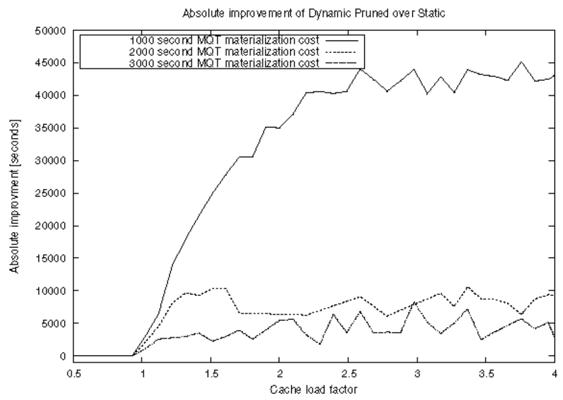

Fig. 12. Synthetic workload execution showing the absolute improvement of the dynamic pruned model over the static model. The three curves represent three different MQT materialization costs.

a constant query execution time without the MQT.

The MQT benefits in the TPC-H workload were already fixed from the MQTs' definitions produced by the Advisor. Thus, to gauge changes in MQT benefit at a fine-grained scale, we ran a **synthetic workload** that allowed us to carefully control the MQTs' benefit characteristics.

In this synthetic workload (which is also used in the next subsection), the number of MQTs per query is randomly chosen for each query based on a non-uniform distribution ranging from 0 MQTs/query to 4 MQTs/query. The mapping between queries to MQTs was determined with a uniform random distribution. (We considered modelling this assignment mapping with a Zipf distribution but could not find any prior research that could sustain this assertion.) The number of MQTs was 42, and the number of queries in the workload was 133, similar to the TPC-H setup.

As shown back in Figure 7, the dynamic pruned model produces a better workload execution time. Our expectation is the difference between the dynamic pruned and static models will widen or narrow depending on the MQTs' net benefits, which is indeed shown in Figure 11, which graphs the absolute difference between the net benefits of the dynamic pruned and the static models. The net benefit for an individual scheme is the difference between the query costs with and without an MQT; this figure graphs the differences betwen these net benefits. When the difference is 0, there is no workload execution time difference between the two models. Four different MQT net benefits are shown as four curves. The materialization cost was set to be 2000 seconds in all.

As can be seen, a higher net benefit per MQT produces an overall higher benefit from the dynamic pruned model. With a high enough net benefit per MQT, the dynamic pruned model consistently improves upon the static model. Note that when the net benefit is small (as shown by the net benefit of 100), the difference between the two models decays rapidly and in fact results in the static model performing better; the materialization cost in this scenario outweighs the small MQT net benefit to such a degree that there is no query permutation that can provide a positive benefit.

4) Effect of varying MQT materialization cost: We also studied the impact of changing the MQT materialization cost. Again, since the materialization costs of the MQTs in the TPC-H workload were already fixed when the MQTs were defined, we used a synthetic workload to control this variable. Here we fixed the net benefit at 300 seconds.

With a lower materialization cost, the overall net benefit will increase, while a higher cost will incur the opposite reaction. Figure 12 shows this effect. As shown in the figure, a high materialization cost incurs less absolute improvement for the dynamic approach.

5) GA performance: The duration that the GA runs can be varied: the longer the GA runs, the better the resulting query order will be produced. We typically ran our GA for under 10 minutes. Given that the cost of running a typical business intelligence workload such as TPC-H can be on the order of minutes to several hours, we believe that the time spent running the GA is almost negligible compared with the benefit gained from query processing reduction.

6) Summary of experiments: We used the cache load factor as a control metric to evaluate the effectiveness of the MQT management schemes. When the cache load factor is 1.0 or less, all MQTs can fit into the cache, and our proposed MQT schemes offer no benefit over a static MQT management scheme. On the other hand, when the cache load factor is over 1.0, the cache cannot hold all the MQTs. In these situations, the static algorithm is more limiting because an early decision must be made to decide which MQTs to materialize.

We make two key conclusions from our experiments:

- First, as shown in Figure 7, when the cache load factor increases, the dynamic models provide net benefit by allowing MQTs to be materialized and dropped on demand.
- Second, while both dynamic models allow for more MQT hits, the dynamic pruned improves upon the dynamic simple by pruning away less-beneficial MQTs that are not used often enough to warrant materialization.

Figures 8, 9, and 10 summarize the latter point. As the cache load factor increases, the static scheme produces fewer MQT hits, resulting in more base table accesses. The dynamic simple model always opts for MQT hits in the cache, resulting in a high number of materalizations and no base table accesses. The dynamic pruned model makes the most effective compromise by managing a smaller set of candidate MQTs than the dynamic simple model: MQTs in this set are materialized and dropped via the cache, and queries needing data in MQTs not in this set instead access the base tables.

V. Related Work

The problem of *view maintenance* has been extensively studied (e.g. [15], [16], [17]) in the context of materialized views in data warehouses. Since a data warehouse consists of a large view, the main focus of the database research has been to maintain materialized views *incrementally*. Numerous algorithms have been proposed for incremental view maintenance [18], [19]. However, none of this work addresses the issue of dynamic management of MQTs. State-of-the-art MQT Advisors in commercial DBMSes (e.g. [3], [7], [20], [9]) are used to recommend candidate MQTs for pre-materialization. The work described in this paper is complementary to these MQT Advisors because we seek to improve the utilization rate of MQTs by providing a dynamic cache.

The work in [11] and [5] are closest to ours. DynaMat [11] dynamically creates cached query results, but it does not consider query re-ordering of batch workloads. Further, this work focuses on data warehouse workloads with star schemas and multidimensional range queries; our work is more general, as it is designed to fit into an existing DBMS and complement an MQT advisor product. The goal of [5] is to interleave creation of materialized structures (either views or indexes) in between query execution in order to improve workload performance. However, they do not consider query re-ordering or an explicit cache, which we have shown are beneficial.

Another related topic is *query caching* [21], [22], [23], [24]. These solutions guarantee that the stored results are always *consistent* with the underlying data. Instead, view invalidation is determining whether a query is independent of a particular update to the underlying data. [25] proposes a more flexible materialization strategy, dynamic materialized views, that aims at reducing storage space and view maintenance costs. A dynamic materialized view selectively materializes only a subset of rows rather than all the rows defined in the materialized view definition.

VI. Conclusion

In this paper we identified the mutually-beneficial combination of dynamic MQT management, query order permutation, and intelligent MQT set pruning. Existing MQT advisors are commonly designed with a static approach to recommend the best MQT set and then pre-materialize the views to fit a cache space constraint prior to executing the workloads. Our system is an automated, dynamic view management scheme that materializes views on-demand as a workload executes and manages them in an LRU cache. We implemented a self-adapting genetic search algorithm to find optimal workload permutations that leverages MQT cache behaviour to high benefit. Additionally, to prevent the system from materializing seldom-used MQTs, we prune the candidate MQT set further. Our results show that our dynamic scheme produces higher benefit across a variety of scenarios compared to a typical static approach.

In the future we look to extend our work. We will compare our dynamic models with other recent research efforts by evaluating them against a variety of different workloads. Additionally, we will study the problem of handling online arriving queries. Finally, we will explore other search heuristics in addition to more advanced genetic algorithm operators to improve the effectiveness of the search.

References

[1] P. Larson and H. Yang. Computing Queries From Derived Relations. In *Proc. of VLDB*, 1985.

[2] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing Queries with Materialized Views. In *Proc. of ICDE*, 1995.

[3] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated Selection of Materialized Views and Indexes for SQL Database. In *Proc. of VLDB*, 2000.

[4] D. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. Lohman, R. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *Proc. of the Intl. Conf. on Autonomic Computing*, 2004.

[5] S. Agrawal, E. Chu, and V. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *Proc. of SIGMOD*, 2006.

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

[7] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, , and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proc. of VLDB*, 2004.

[8] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proc. of VLDB*, 2004.

[9] B. Dagerville, D. Das, K. Dias, K. Yagoub, M. Zait, , and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *Proc. of VLDB*, 2004.

[10] J. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MIT Press, 1992.

[11] Kotidis Y and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proc. of SIGMOD*, 1999.

[12] P. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *Proc. of ICDE*, 2004.

[13] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Kluwer Academic Publishers, 1989.

[14] L. Davis. Job Shop Scheduling with Genetic Algorithms. In *Proc. of the Int. Conf. on Genetic Algorithms*, 1985.

[15] E. Rundensteiner, A. Koeller, and X. Zhang. Maintaining data warehouses over changing information sources. *Communications of the ACM*, 43(6):57–62, 2000.

[16] L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *Proc. of SIGMOD*, 1996.

[17] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. In *Proc. of IPDIS*, December 1996.

[18] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance in Data Warehouses. In *Proc. of the ACM Int. Conf. on Management of Data*, 1997.

[19] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How to roll a join: Asynchronous Incremental View Maintenance. In *Proc. of SIGMOD*, 2000.

[20] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proc. of ICDE*, 2000.

[21] S. Adalı, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmaninan. Query Caching and Optimization in Distributed Mediator Systems. In *Proc. of SIGMOD*, 1996.

[22] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic Caching of Query Results for Decision Support Systems. In *Proceedings of the 1999 Symposium on Statistical and Scientific Data Base Management*, pages 254–263, 1999.

[23] Q. Luo, J. F. Naughton, R. Krishnamurthy, P. Cao, and Y. Li. Active Query Caching for Database Web Servers. In *Proc. of the WebDB*, 2000.

[24] C. Chen and N. Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Proc. of EDBT*, 1994.

[25] J. Zhou, P. Larson, J. Goldstein, and L. Ding. Dynamic Materialized Views. In *Proc. of ICDE*, 2007.