The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching*

Chungmin Melvin Chen and Nicholas Roussopoulos

Department of Computer Science
University of Maryland, College Park
MD 20742, USA

Abstract. In this paper, we describe the design and implementation of the ADMS query optimizer. This optimizer integrates query matching into optimization and generates more efficient query plans using cached results. It features data caching and pointer caching, alternative cache replacement strategies, and different cache update methods. A comprehensive set of experiments were conducted using a benchmark database and synthetic queries. The results showed that pointer caching and dynamic cache update strategies substantially saved query execution time and, thus, increased query throughput under situations with fair query correlation and update load. The requirement of the disk cache space is relatively small, and the extra optimization overhead introduced is more than offset by the time saved in query evaluation.

1 Introduction

The technology of caching query (intermediate) results for speeding up subsequent query processing has been studied widely in previous literature. The benefit of this technique is obtained from saving (part of) the subsequent query computations by utilizing the previous cached results. Applications of this technique can be found in different areas. In [Fin82, LY85, Rou91], cached query results were used in relational database systems to avoid repeated computations. [Sel87, Jhi88] addressed the problem of caching query results to support queries with procedures, rules and functions. In a client-server environment, caching query results on local workstation can not only parallelize query processing among clients, but also reduce the bus contention and the server request bottleneck [DR92]. Recently, this technique was also suggested to support query computations in extensible or object-oriented database systems where expensive computations are more likely to happen [HS93].

Different issues concerning the caching technique have also been studied. [AL80, Rou82b, Val87, Rou91] proposed alternative methods for storing the cached data, [Sel88, Jhi88] discussed the problem of selective caching and cache replacement, and in [RK86, Han87, BLT86], different strategies for updating

* This research was sponsored partially by NSF under grant IRI-9057573 and GDR-85-00108, by NASA/USRA under contract FCPO-550-81, by ARPA under contract 003195, and by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

cached data are explored. Aside from the above work on cache management, the problem of how to identify the useful cached data for computing queries, referred as *query matching* problem, was addressed in [Fin82, LY85]. In their work, however, query optimization was not considered inclusively. This is not satisfactory because blindly using cached data in query computations without optimization may result in a query plan even worse than the one optimized from the original query without utilizing any cached results. Therefore, it is necessary to consider optimization at the same time of query matching. This issue was first discussed in [J$^+$93]. However, since their work emphasized on supporting transaction time using differential techniques, the matching and optimization problem was not addressed sufficiently, and no performance evaluation was reported.

In this paper, we describe the design and implementation of the ADMS [2] Cache&Match Optimizer (CMO), and present a comprehensive performance evaluation. By caching previous query results on the disk, the CMO is able to perform the matching coincidently with the optimization, and generate an optimal plan using cached results. The integrated work is also enriched from the previous work that now it (1) can use multiple cached results in computing a query, (2) allows dynamic cache update strategies, depending on which is better, and (3) provides options for different cache management strategies. A variety of experiments were conducted to evaluate the performance of CMO, under alternative strategies and different situations. The results showed that with appropriate strategies, CMO can always improve substantially the performance.

The rest of this paper is organized as following. In Section 2, we discuss the framework of CMO and related issues. Section 3 describes the integration of query matching and optimization. Section 4 presents the experiment results from implementation. And finally in Section 5, we give the conclusion and future research direction.

2 The CMO Framework and Related Issues

The CMO mechanism consists of two major functional components: the *query matching optimizer* and the *cache manager*. Incoming queries are optimized through the matching optimizer, which capitalizes on a structure called LAPS (Logical Access Path Schema) in finding pertinent cached results, in order to generate more efficient plan. Query or intermediate results are saved on the disk and maintained by the cache manager, which keeps track of all the cached data and decides which to replace when the cache space is full. In the following, we discuss the concerned design issues for both modules, review the related work, and describe the approaches we adopted in the implementation.

2.1 Cache Management

Conventional relational database systems always allow users to save the final query results in relations [S$^+$79, SWK76]. It is not uncommon that intermediate

[2] ADMS, the Advanced Database Management System, is a DBMS developed at the Department of Computer Science, University of Maryland, College Park [RES93].

results might also need to be produced to facilitate the query computations. Throughout this paper, we assume intermediate results are generated during query computations. Both intermediate and final query results are referred as *temporaries*. Though temporaries are mostly retained only within a query computation, it is not hard to keep them on disk over a longer time for potential reuse. In the following, we discuss some key issues regarding the management of cached temporaries.

How to store the cached temporaries ? Temporaries can be stored as actual data in relations called *materialized views* [AL80, BLT86]. Another approach is to store for each resulting tuple of the temporary, a number of *Tuple Identifiers (TID)*, instead of materialized values, which point to the corresponding tuples in the base relations, possibly through several levels, that constitute the resulting tuple. Variations of this *pointer based* approach have been proposed in [Rou82b, Val87, Rou91]. In contrast to materialized data caches, materializing pointer caches requires extra page references to higher level relations or temporaries. However, pointer caching is more space-effective since each tuple is represented by a small number of fixed length pointers. Besides, from the view of query matching, pointer caching is more versatile than data caching because (1) more temporaries can be retained in a limited cache space, and (2) unlike data caches which have only projected attributes, pointer caches virtually serve as indices to the base tuples and, thus, can select any attributes of the underlying relations whether or not used in the queries. Nevertheless, both pointer and data caching are implemented and evaluated in CMO.

What to cache ? In a system which provides unbounded disk space, we can simply cache everything generated and leave the decision of using these cached temporaries to the query optimizer. However, a more realistic situation is when we bound the available space for caching. In this situation, a *cache replacement strategy* must be employed to decide which temporaries to replace when the cache space is full. The problem of choosing a good replacement strategy so that the most profitable results can always be cached was addressed in [Sel88]. We incorporated some of the suggested cache replacement heuristics into CMO and experimented with them under different available cache space.

How to update outdated cached temporaries ? Cached temporaries become outdated when the underlying base relations are updated, and thus must be updated before they can be further used. Different strategies regarding *when* to update the outdated caches include: (1) *immediate update* (i.e., when relevant base relations are changed), (2) *periodical update*, and (3) *on-demand update* (i.e., only when they are requested). As for the cache update method, aside from updating via *re-execution*, the technique of *incremental update (*or *differential computation)* [LHM86, BLT86, Rou91] can efficiently update a temporary if only a little part of it is changed.

It was analyzed in [Han87] that none of the combinations of update strategy and update method is superior to all the others under all situations. As it is practically prohibitive to experiment with all possible combinations, on-demand strategy has been adopted in our implementation because it can batch consecutive updates into a single update (and thus reduce the excessive overhead of multiple smaller updates) and always prevents the unnecessary updates to never-used caches. The CMO, however, dynamically chooses between re-execution and incremental computations, depending on their corresponding estimated costs. The performance of CMO under different levels of relation update loads is evaluated in detail in the experiments.

How to keep track of the cached temporaries ? To facilitate the searching and matching against the cache pool, a LAPS structure is used in CMO to keep track of all the cached temporaries efficiently. Instead of recording each cached temporary independently, the LAPS integrates the cached temporaries along with their logical access paths which capture both the access methods and the derivation relationships. The integration of new cached temporaries and logical access paths into the LAPS is fairly direct and has been developed in [Rou82a].

2.2 Query Matching and Optimization

The task of generating the optimal plan, which may or may not use the cached temporaries, for a given query can be conceptually divided into two parts: matching and optimization.

Matching The problem of detecting if a cached temporary is useful for the computation of a query has been investigated in [Fin82, LY85]. We have adapted the method from [Fin82] to our use in the CMO optimizer. Besides, rather than using only one matched cache in a query, the CMO optimizer is capable of using multiple matched temporaries to answer the query more efficiently.

Optimization Optimization is required not only because there may be different combinations of matched caches from which the query can be computed, but also because it is not always beneficial to use caches. A possible solution, as suggested in [Fin82], is a *two phase* approach; during the first phase, the query is transformed into a number of equivalent queries using different cached temporaries, and during the second phase, all the revised queries are fed to a regular optimizer to produce an optimal plan. Without elaborate pruning, this approach may navigate an extremely large search space, even when only a few revised queries are produced from the first phase. A better alternative is to integrate the matching with the optimization and thus, unify the search spaces and avoid duplicate effort [J+93]. The CMO optimizer we implemented here belongs to the second approach.

In summary, the one-phase CMO provides the options of using different cache replacement strategies, data and/or pointer caches, and incremental and/or re-execution updates. In the next section, we describe the integration of matching and optimization in more detail.

3 Integrating Query Matching and Optimization

The matching mechanism only handles the class of *PJS-queries*— queries which involve only projection, selection and join. A PJS-query q is expressed in SQL as: `select` $\bar{a}_q$ `from` $\bar{r}_q$ `where` f_q, where $\bar{r}_q = r_1, r_2, \ldots, r_k$ are operand relations,

$\bar{a}_q = a_1, a_2, \ldots, a_l$ are projected attributes, and f_q is a boolean formula for which the resulting tuples must satisfy. We can therefore represent any query q as $q = (\bar{a}_q, \bar{r}_q, f_q)$. If $\bar{r}_q$ contains any derived relations, the query can be expanded to an expression which involves only base relations, we use $q = (\bar{A}_q, \bar{R}_q, F_q)$ to denote such *expanded* expression. With the same notations, we use $v = (\bar{a}_v, \bar{r}_v, f_v)$ to emphasize that temporary v is *directly* computed from the operand set $\bar{r}_v$ *without* producing any intermediate results in between. Similarly, the expanded notation for v is given by $v = (\bar{A}_v, \bar{R}_v, F_v)$.

In CMO, the query matching and optimization are integrated using a *graph searching* algorithm [Nil80] (referred as state transition network in [J$^+$93, LW86]). The input to the optimizer includes an initial *query graph* (or *state*) which represents the uncomputed query, and a LAPS which models the cached temporaries. A state is *reduced* to a successive state when a part of the query is computed or matched by a cached temporary. Thus, starting from the initial state, successive states are generated until a final state, which represents the totally computed query, is reached. The path with the lowest cost leading to the final state is chosen as the optimal plan. In the following, we first define the LAPS and describe the steps of query matching, and then sketch out the unified CMO matching optimizer. More details about the algorithm and implementation can be found in a complete version of this paper [CR93].

3.1 The Logical Access Path Schema and Query Matching

Definition 1 A *Logical Access Path Schema* (LAPS) is a *directed* graph (N, E) where N is a set of nodes corresponding to base relations and cached temporaries, and E is a set of directed *hyperedges* corresponding to logical access paths such that for any temporary $v = (\bar{a}_v, \bar{r}_v, f_v) \in N$,

1. $x \in N,\ for\ all\ x \in \bar{r}_v$, and
2. there is a hyperedge $e = (\bar{r}_v, v) \in E$ leading from the set of operand nodes $\bar{r}_v$ toward v, and labelled with f_v. □

Initially, the LAPS contains base relations only. When subsequent queries are processed and results are cached, it is augmented by integrating the cached temporaries along with their logical access paths[Rou82a]. The purpose of having LAPS is to facilitate the searching for qualified cached temporaries in parallel with the optimization.

We are now concerned about how to check if a cached temporary can be used in computing a query. A temporary is useful if itself alone can be used to compute a sub-query of the given query. Formally, we say a (sub-)query q is *derivable* from a cached temporary v (or v is a *match* of q) if there exist an attribute set A and a formula F such that, for any database instance d, $q(d) = \pi_A(\sigma_F(v(d)))$, where $q(d), v(d)$ denote the result of q and content of v under instance d, respectively. We list below without proof the conditions under which a temporary v is *sufficiently* qualified to be a match of a query q. In what follows, $x_1, \ldots, x_n$ denote the attributes in the corresponding formula; '$\rightarrow$', '$\leftrightarrow$' denote *logical implication* and *logical equivalent* respectively.

Condition 1 (Operand Coverability) $\bar{r}_v = \bar{r}_q$.

Condition 2 (Qualification Coverability) $\forall x_1, \ldots, x_n\ (f_q \rightarrow f_v)$, and, there exists a *restricting* formula f^r on v such that $\forall x_1, \ldots, x_n\ (f_q \leftrightarrow f_v \wedge f^r)$.

Condition 3 (Attribute Coverability) $\bar{a}_v \supseteq (\bar{a}_q \cup \alpha(f^r))$, where $\alpha(f^r)$ are attributes appearing in f^r.

Rather than using a looser condition $\bar{R}_v = \bar{R}_q$, Condition 1 requires the exactly same set of parent operands. However, this will not lose any generality when we capitalize on the LAPS to integrate the matching and optimization. Condition 2 guarantees that every tuple t in the result of q has a corresponding tuple t' in v such that t is a sub-tuple of t', and there exists a formula f^r through which these t' can be selected from v. Condition 3 assures that temporary v contains all the attributes that are projected in the target list of query q, as well as those required to evaluate f^r. The following lemma is a direct consequence of the above conditions.

Lemma 1. *v is a match of q if all the above three conditions are satisfied, in particular, $q(d) = \pi_{\bar{a}_q}(\sigma_{f^r}(v(d)))$ for all database instance d.*

This lemma basically states how to recognize a single cached temporary from which the result of a (sub-)query can be directly extracted. It does not, however, say anything about how to use multiple temporaries in computing a query. This is achieved by interleaving the optimization with the above steps of match checking in a unified searching algorithm and is described in the following subsection.

3.2 Integrating Matching with Optimization

The CMO optimizer can be viewed as a network of query graph reductions. A *query graph (*or *state)* represents the status of the original query as well as any partially processed queries during the optimization. A query graph is *reduced* to a new one by replacing a connected sub-graph with a single new node which corresponds to either a new intermediate result[3] or a matched temporary found in the LAPS. In the following, we formalize the query graph, reductions, and then describe the searching strategy.

Definition 2 (Query Graph) A *query graph* (or a *state*) is a connected, *undirected* graph $G(N, E)$ where

1. each node $n \in N$ denotes a relation, a cached temporary, or an intermediate result, and is associated with a schema $\alpha(n)$ and a projected attribute list $a(n) \subseteq \alpha(n)$,
2. each *hyperedge* $e \in E$ connects a subset of nodes $N(e) \subseteq N$, and is labelled with a formula $f(e)$. e is a *join edge (selection edge)* if $|N(e)| > 1\ (= 1)$. □

It is not hard to see that a query graph (N, E) actually represents a query $(\bar{a}_q, \bar{r}_q, f_q)$ where $\bar{a}_q = \cup_{n \in N} a(n), \bar{r}_q = N$, and $f_q = \wedge_{e \in E} f(e)$. Also note that since a formula can always be transformed into a conjunction of sub-formulas [CL73], every PJS-query can be represented by a query graph. Before we can define the reductions, we need to define *induced sub-query*. Given a state $q =$

[3] We use *intermediate results* to refer to those which are planned during the optimization but are not actually produced yet.

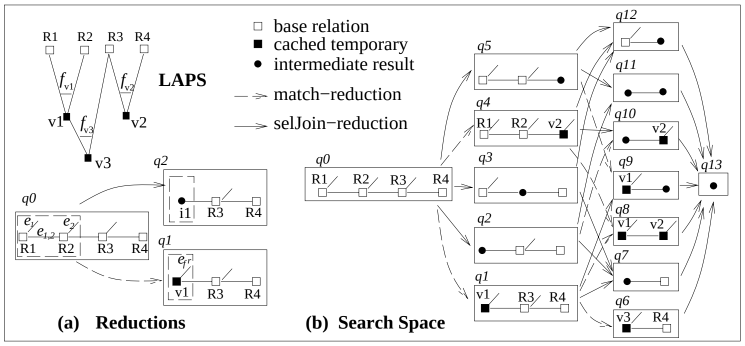

Fig. 1. State Reductions and Searching

(N, E) and an edge $e \in E$, let E^e_{ext} be the set of edges which connect at least one node from $N(e)$ with at least another node *not* from $N(e)$, and E^e_s be the selection edges incident on any node in $N(e)$, then the *sub-query* induced by e is a query $q_e(\bar{a}_{q_e}, \bar{r}_{q_e}, f_{q_e})$ such that

$$\bar{a}_{q_e} = a(N(e)) \cup (\alpha(N(e)) \wedge \alpha(E^e_{ext})), \quad \bar{r}_{q_e} = N(e), \quad f_{q_e} = f(e) \wedge f(E^e_s),$$

where $a(N(e)) = \cup_{n \in N(e)} a(n)$, $\alpha(N(e)) = \cup_{n \in N(e)} \alpha(n)$, $f(E^e_s) = \wedge_{e' \in E^e_s} f(e')$ and $\alpha(E^e_{ext})$ are the attributes appearing in the formula $f(E^e_{ext}) = \wedge_{e' \in E^e_{ext}} f(e')$.

Intuitively, a state is reduced by assigning an access path to the sub-query induced by one of its join edges. There are two different reductions, the *selJoin-reduction* corresponds to computing the induced sub-query directly from the operand relations of the sub-query; the *match-reduction* corresponds to using a matched cached temporary. We define both reductions below and demonstrate them using the example in Figure 1.(a).

Definition 3 (selJoin Reduction) State $q = (N, E)$ is *selJoin-reduced*, on a given join edge $e \in E$, to a new state $q' = (N', E')$ by constructing

1. $N' = N - N(e) \cup \{n'\}$, where $n' \notin N$ and $a(n') = a(N(e))$, $\alpha(n') = \bar{a}_{q_e}$,
2. $E' = E - \{e\} - E^e_s - E^e_{ext} \cup E^e_{ext'}$, where $E^e_{ext'}$ is a new set of edges formed from E^e_{ext} by replacing each occurrence of nodes from $N(e)$ with the new node n'. □

Definition 4 (Match Reduction) State $q = (N, E)$ is *match-reduced*, using a temporary $v(\bar{a}_v, \bar{r}_v, f_v) \in LAPS$ for a join edge $e \in E$, to a new state $q' = (N', E')$ if the induced sub-query q_e is *derivable* from v through a restricting formula f^r (as described in Lemma 1). In particular,

1. $N' = N - n(e) \cup \{n_v\}$, where $n_v \notin N$ and $a(n_v) = a(N(e)), \alpha(n_v) = a_{q_e}$.
2. $E' = E - \{e\} - E^e_s - E^e_{ext} \cup E^e_{ext'} \cup \{e_{f^r}\}$, where e_{f^r} is a new selection edge on n_v and is labelled with f^r. □

A selJoin-reduction is illustrated in Figure 1.(a), where state q_0 is selJoin-reduced to q_2 on edge $e_{1,2}$. Note the induced sub-graph $q_{e_{1,2}}$, bounded by a dashed rectangle in q_0, is replaced by a new intermediate result node $i1$ in q_2. A match-reduction is also shown, where state q_0 is reduced to q_1 on edge $e_{1,2}$.

The induced sub-query $q_{e_{1,2}}$ is replaced by a cached temporary $v1$ from LAPS ($v1$ is a match of $q_{e_{1,2}}$) and a selection edge e_{f^r} where f^r is the corresponding restricting formula.

Based on these two reductions, a *dynamic programming* searching strategy is used to find the optimal plan[4]. It performs a breadth-first search and restricts the searching space by merging *equivalent states* [CR93]. The cost of a state q, denoted $cost(q)$, is computed as the least cost among all the paths that lead from the initial state to q. The cost model uses weighted sum of CPU and I/O time and takes into account the costs of pointer cache materialization and incremental updates. The searching algorithm is outlined in the following.

```
Step 1 Let $q_0(N_0, E_0)$ be the initial state. $T := \{q_0\}$, $S := \emptyset$. Repeat Step 2 for
    $|N_0| - 1$ times.
Step 2 $S := T$, $T := \emptyset$. For each $q(N, E) \in S$, and each join edge $e \in E$, do
    2.1 apply selJoin-reduction to $q$ on $e$, let $q_1$ be the reduced state; apply
        match-reduction to $q$ on $e$ if applicable, let $q_2$ be the reduced state.
    2.2 If there exists no $q' \in T$ equivalent to $q_1$, then $T := T \cup \{q_1\}$, otherwise
        if $cost(q') > cost(q_1)$ then $T := T - \{q'\} \cup \{q_1\}$. Do the same for $q_2$.
Step 3 Output the cheapest path leading from $q_0$ to the single final state in $T$
    as the optimal plan.
```

Continuing on the example of Figure 1.(a), its searching space is given in figure (b) where the selJoin-reduction and match-reductions are drawn in solid and dashed arrows respectively. Three iterations are performed, with a final state generated at the lowest level. Note that in this case, q_1 is further match-reduced to q_6 by using a matched temporary v_3, and q_8 corresponds to the state after using two matched temporaries v_1 and v_2. Merging of equivalent states are reflected by those arrows that come into the same state.

4 Experiments: Performance Evaluation

A CMO component has been incorporated in ADMS. It sits on top of the storage access module which provides alternative access methods to the relations. The access methods include sequential and index scan for single relation, and three join methods: nested loop, index, and hash joins. In the following, we present the experiments that were conducted to evaluate the performance of CMO.

4.1 The Experiment Environment

The experiments were carried out by running a centralized ADMS on a Sun SPARCstation 2. All the experiments were run under a single user stand-alone mode, so that the benefit from CMO can be measured in terms of elapsed time. Different databases and query loads were used throughout the experiments so that the impact from the CMO parameters as well as from the system's environment can be observed.

[4] An A* algorithm was used in an early version of the ADMS optimizer, however, experiments showed that it does not benefit much in reducing the searching space.

Relation	100	1k	2k	5k	10k
Cardinality	100	1,000	2,000	5,000	10,000
Size (KB)	20	208	408	1,008	2,016

Relation	100s	1ks	2ks	5ks	10ks
Cardinality	100	1,000	2,000	5,000	10,000
Size (KB)	10	95	200	496	976

Table 1. Synthetic Relations

Databases	Relations
DBMIX	1k, 2k, 5k, 10k
DBMIX-S	1ks, 2ks, 5ks, 10ks
DB100	100, $100'$, $100''$, $100'''$
DB1k	1k, $1k'$, $1k''$, $1k'''$
DB5k	5k, $5k'$, $5k''$, $5k'''$

Table 2. Five Different Databases

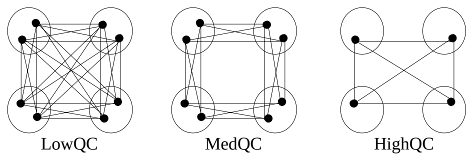

Fig. 2. Three Levels of Query Correlations

Databases Synthetic relations are generated according to the characteristics of the Wisconsin Benchmark [BDT83]. Table 1 outlines the cardinalities and sizes of each relation used in the experiments. Throughout the experiments, each tested database consists of four relations. Table 2 lists all the used databases, note that the primes ($'$) indicate the different relation instances of the same schema, cardinality and attribute value distributions.

Workload and Query Characteristics The query workload is generated by a customized random query generator. By specifying desired query characteristics, different copies of *query streams* can be generated that all satisfy the given characteristics. In the experiment, we restrict the number of join attributes so that common sub-expressions can recur in queries and, thus, the effectiveness of CMO can be observed. Informally, the *query correlation* among the queries of a query stream is measured as the number of distinct equal-join predicates appearing in it. Figure 2 shows three classes of query correlations used to generate the query streams. The circles denote the relations, the nodes denote the join attributes, and the edges denote the possible join predicates. Note that a maximum of 6, 16, and 24 distinct join predicates can be generated in HighQC, MedQC, and LowQC respectively.

Selection predicates are chosen from random attributes subject to the specified query selectivities. To allow best chance of data caching, every query is projected on all attributes of its participating relations. This makes no difference to pointer cache, but requires more space for data cache. Updates are restricted to certain attributes so that the cached temporaries will not change drastically in cardinality. However, the qualifying predicates in the update queries are randomly chosen from all attributes. Throughout the experiments, each query stream contains at least 50 queries. When not mentioned explicitly, the defaults for the database and the query correlation are DBMIX and MedQC respectively.

Performance Metrics The total *elapsed time* of a query stream execution, including query optimization and query computation time, is taken as the main metric for evaluating the performance outcome. Through the whole experiments, each run (query stream) was repeated several times and the average elapsed time was computed.

4.2 Experiment Results

Three major experiment sets were run to evaluate the CMO. The first set compared with different cache management strategies under different degrees of cached space availability. The second set of experiments were conducted to observe the performance degradation of CMO, under three different strategies of cache updates and different degrees of update selectivities and frequencies. And finally, we evaluated the performance impacts from the factors of database sizes and query characteristics. The overhead of CMO is also shown at the end.

4.2.1 Effect of Cache Management We compare the performance of data caching (DC) and pointer caching (PC) under three replacement strategies:

LRU: the least recently used — in case of a tie, LFU, LCS
LFU: the least frequently used — in case of a tie, LRU, LCS
LCS: the largest cache space required — in case of a tie, LRU, LFU

Two databases were tested (DBMIX, DBMIX-S) , the results are shown on the six curves labelled accordingly in Figure 3. As can be seen all PC curves are roughly the same but among them, PC/LCS is slightly better. In set DBMIX, the pointer caching runs faster than the data caching on all cache space ranges from 0 to 25 MB (we assume that disk space is sufficient for retaining intermediate results within a query). This suggests that when fair amount of intermediate results are generated and written to and read from the disk, the materialization cost of PC is compensated by its efficient write cost. Note that in this case, even with cache space more than 10MB, the performance of DC/LCS is still worse than that of PC/LCS with only 2MB. The inferiority of data caching can be attributed to the large overhead in writing and reading the intermediate results. To make it more competitive, the same experiment was performed again on a smaller database DBMIX-S within which the tuple length is now only around half of the original one. The results are shown in Figure 3.b, where DC now becomes closer to, though still inferior to, PC in performance.

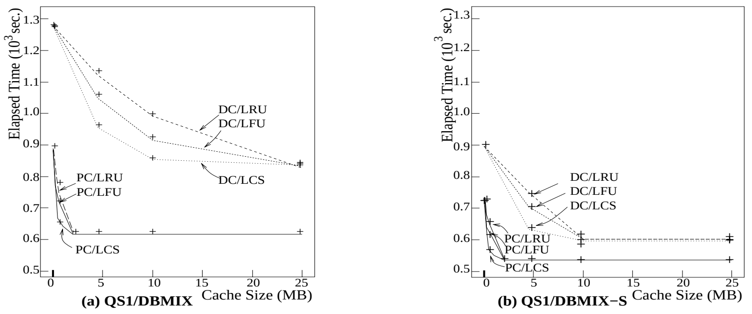

Fig. 3. Effect of Cache Management

It is fair to argue that some savings can be achieved in data caching by projecting out some of the non-useful attributes of the intermediate results. However, such a projection reduces the potential reuse of these intermediate results in other queries which may need these attributes. Thus, for data caching, there is a dilemma between reducing the intermediate size and enhancing the chance of cache reuse. Pointer cache, on the other hand, does not have this problem at all, since all attributes are implicitly inherited in the non-materialized cache. And for this reason, we believe that pointer caching is the proper choice in implementing a CMO-like mechanism. In the rest experiments, only pointer caching with LCS replacement is considered and the cache size is set to 4MB.

4.2.2 Effect of Relation Updates In this experiment, we evaluate the CMO performance degradation under relation updates. Three variations of CMO are evaluated under different degrees of update frequency and selectivities. CMO/INC uses incremental update only, CMO/REX uses re-execution update only, and CMO/DYN allows both methods and leave the decision to the optimizer. The performance of a regular optimizer (REG) without caching and matching technique is also included for comparison. Both relation update frequency (no. of update queries / no. of total queries) and update selectivity (no. of updated tuples / relation cardinality) are controlled. For each raw query stream (which contains no update queries), 15 variations are produced by interleaving it with update queries according to each combination of 5 different levels of update frequencies and 3 different levels of update selectivities. The frequencies range from 0%, 5%, 10%, 15% to 25%, and the update selectivities range from LS ($1\% - 5\%$), MS ($6\% - 10\%$) to HS ($6\% - 10\%$ for 2/3 of the update queries, and $40\% - 50\%$ for the other 1/3).

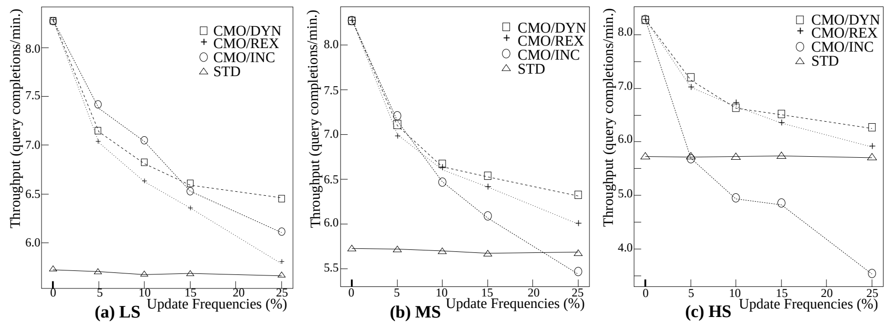

Fig. 4. Effect of Relation Updates

Four different query streams are experimented, Figure 4 depicts the average throughputs among all four query streams under different situations. The CMO curves, no matter what update strategies are used, perform better than REG in all LS, MS and HS sets (except CMO/INC in set HS), and decline elegantly as update frequency increases. This is no surprise since the cost of updating a outdated temporary is compensated by those subsequent queries that are able to use it before it becomes outdated again. In set LS, CMO/REX performs worse than the other two since re-execution update do not take advantage of incremental computation. CMO/INC is better than CMO/DYN at 5% and 10% update frequencies, but as the frequencies increase, it is outperformed by CMO/DYN. This is due to the increasing overhead of update log processing in CMO/INC. In set MS, except at 5% frequency, CMO/DYN performs the best; CMO/INC now swaps position with CMO/REX from LS. And finally in HS, CMO/INC declines drastically, and performs even worse than REG for frequencies greater than 10%. CMO/DYN still performs the best in this set.

The reader might wonder why CMO/DYN, which is supposed to be theoretically the best under all circumstances, is inferior to CMO/INC at update frequencies 5% and 10% in LS. We analyzed the statistical profile and found out that it is due to the problems of *inaccurate cost estimation* and the lack of *global (multiple) query optimization*, which somtimes cause CMO/DYN to chose less efficient paths than CMO/INC and/or CMO/REX. Overall, though, CMO is cost effective in most environments where queries arrive in no ad hoc manner and, thus, there is no good way to predict what queries will appear in the stream.

4.2.3 Effect of Query Correlation and Database Sizes In this set of experiment, we observe the impact of the database environment on the CMO performance. Figure 5 shows the performance improvement of CMO over REG under three classes of query correlations: LowQC, MedQC, and HighQC. For each class, the results of three random generated query streams (QS1, QS2, QS3) are presented each of which consists of 70 queries. The portion below the horizontal dashed line denotes the total elapsed time of those queries that do not find any matched caches for use. It shows that CMO reduces the total elapsed time in all classes with a significant amount. And as query correlation increases, the improvement increases because of higher chance of match.

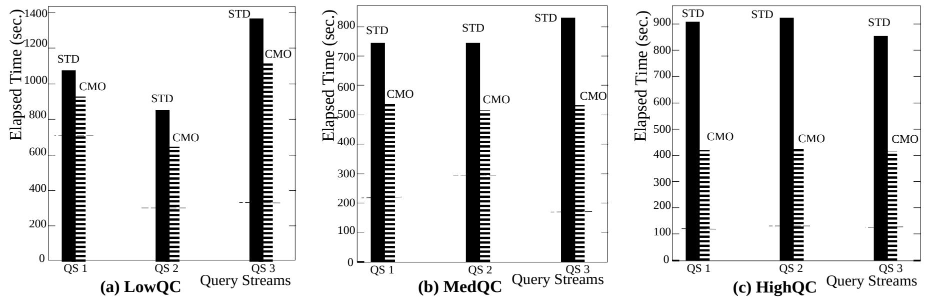

Fig. 5. Effect of Query Correlations

We also observed the effect of query selectivities. Figure 6 compares the results between two classes of query selectivities: low selectivities of $0.0001 - 0.05$ (LS) and high selectivities of $0.0001 - 0.3$ (HS). Intuitively, query matching should have been more advantageous in a large database with small selectivity queries, because it tends to save a large computation by caching a small amount of results. However, our results show that for the higher selectivity queries (HS), the relative improvement of CMO over REG is still as good as that in the lower selectivity one (LS), though elapsed time has almost doubled in HS.

To see the effect of database size, three different size databases were experimented in another set. We have adjusted the query selectivities for each query stream so that the query result sizes do not diverge too much among all three database sizes. The purpose of doing so is to observe the improvement trend for different size databases supposed that the query result sizes are fairly small and unchanged. Figure 7 shows the results, where CMO consistently shorten the

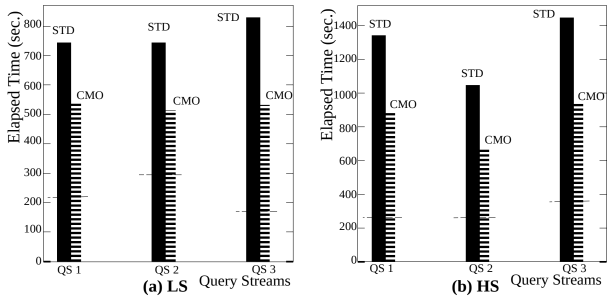

Fig. 6. Effect of Query Selectivities

	REG	CMO
Exp. 4.2.1	0.084	0.133
Exp. 4.2.2	0.087	0.135
Exp. 4.2.3	0.088	0.149

Table 3. Ave. Optimization Overhead (sec./ per query)

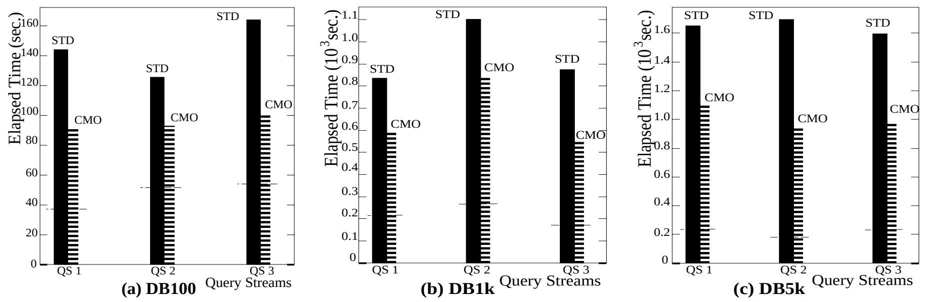

Fig. 7. Effect of Database Size

elapsed time from REG. However, no drastic differences in relative improvement can be told among the three database sizes.

Finally, we compare the optimization overhead of CMO with REG. Table 3 lists the average optimization time per query for each set of experiments we presented above. Though CMO has around $50\% - 60\%$ more optimization time than REG, the extra overhead introduced by CMO is relatively small when compared to and, thus, is compensated by the time saved in query computations.

5 Conclusion

This paper describes the ADMS query optimizer which matches and integrates in its execution plans query results cached in pointers or materialized views. The optimizer is based on the Logical Access Path Schema, a structure which models the derivation relationship among cached query results, for incrementally identifying the useful caches while generating the optimal plan. The optimizer features data caching and pointer caching, different cache replacement strategies, and different cache update strategies.

An extensive set of experiments were conducted and the results showed that pointer caching and dynamic cache update strategies substantially speedup query computations and, thus, increase query throughput under situations with fair query correlation and update load, The requirement of the cache space is relatively small and the extra computation overhead introduced by the caching and matching mechanism is more than offset by the time saved in query processing.

For the future research, we would like to extend the CMO techniques to concurrent queries and investigate how results cached from a query can be used in another concurrent query . Also we would like to adapt the CMO optimizer to the ADMS$\pm$[RK86, RES93] client-server. In this environment, query results are cached in local client workstation disks and used to achieve parallelism in query processing.

References

[AL80] M.E. Adiba and B. G. Lindsay. Database snapshots. In *Procs. of the 6th Intl. Conf. on Very Large Data Bases (VLDB)*, 1980.

[BDT83] D. Bitton, D.J. DeWitt, and C. Turbyfill. Benchmarking database systems, a systematic approach. In *Procs. of the 9th Intl. Conf. on VLDB*, 1983.

[BLT86] J.A. Blakeley, P. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1986.

[CL73] C. L. Chang and C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[CR93] C.M. Chen and N. Roussopoulos. Implementation and performance evaluation of the ADMS query optimizer. Technical Report CS-TR-3159, Dept. of Comp. Sci., University of Maryland, College Park, 1993.

[DR92] A. Delis and N. Roussopoulos. Evaluation of an enhanced workstation-server DBMS architecture. In *Procs. of the 18th Intl. Conf. on VLDB*, 1992.

[Fin82] S. Finkelstein. Common expression analysis in database applications. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 235–245, 1982.

[Han87] E.N. Hanson. A performance analysis of view materialization strategies. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 440–453, 1987.

[HS93] J.M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1993.

[J+93] C. S. Jensen et al. Using differential techniques to efficiently support transaction time. *VLDB Journal*, 2(1):75–111, 1993.

[Jhi88] A. Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *Procs. of the 14th Intl. Conf. on VLDB*, 1988.

[LHM86] B. Lindsay, L. Haas, and C. Mohan. A snapshot differential refresh algorithm. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 53–60, 1986.

[LW86] S. Lafortune and E. Wong. A state transition model for distributed query processing. *ACM TODS*, 11(3):294–322, 1986.

[LY85] P.-Å. Larson and H. Z. Yang. Computing queries from derived relations. In *Procs. of the 11th Intl. Conf. on VLDB*, pages 259–269, 1985.

[Nil80] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Pub. Co., 1980.

[RES93] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A testbed for incremental access methods. *Appeared in IEEE Trans. on Knowledge and Data Engineering*, 1993.

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS$\pm$. *Computer*, 19(12):19–25, 1986.

[Rou82a] N. Roussopoulos. The logical access path schema of a database. *IEEE Trans. on Software Engineering*, SE-8(6):563–573, 1982.

[Rou82b] N. Roussopoulos. View indexing in relational databases. *ACM TODS*, 7(2):258–290, 1982.

[Rou91] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *ACM TODS*, 16(3):535–563, 1991.

[S+79] P. G. Selinger et al. Access path selection in a relational database management system. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, 1979.

[Sel87] T. Sellis. Efficiently supporting procedures in relational database systems. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1987.

[Sel88] T. K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Inform. Systems*, 13(2), 1988.

[SWK76] M. Stonebraker, E. Wong, and P. Kreps. The design and implementation of INGRES. *ACM TODS*, 1(3):189–222, 1976.

[Val87] P. Valduriez. Join indices. *ACM TODS*, 12(2):218–246, 1987.