UMIACS-TR-93-106
CS-TR-3159
October 1993

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

Chungmin Melvin Chen   Nick Roussopoulos

Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland, College Park
E-mail: min@cs.umd.edu, nick@cs.umd.edu

Abstract

In this paper, we describe the design and the evaluation of the ADMS optimizer. Capitalizing on a structure called Logical Access Path Schema to model the derivation relationship among cached query results, the optimizer is able to perform query matching coincidently with the optimization and generate more efficient query plans using cached results. The optimizer also 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.

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 (intermediate) results. Applications of this technique can be found in [Fin82, LY85, DR92, Sel87, Rou91, Jhi88, HS93, AL80]. In [Fin82, LY85, Rou91], cached query results were used in relational database systems to avoid repeated computations. [Sel87, Jhi88] addressed the problem

*This research was sponsored partially by NSF under grant IRI-9057573 and GDR-85-00108 and by NASA/USRA under contract FCPO-550-81. This paper also appears in *Proceedings of the 4th International Conference on Extending Database Technology, 1994.*

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 cached data are explored. Aside from the above work which focused on the problem of cache maintenance and 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. The first time where query matching and optimization are integrated was 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 of the Cache&Match Optimizer (CMO) on the ADMS [1] database management system, and present a comprehensive performance evaluation. We model the cached query results with a structure called *Logical Access Path Schema (LAPS)* [Rou82a], and based on this, 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. The advantages of CMO are exhibited by variety of experiments on ADMS. These experiments were designed and 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,

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

which capitalizes on the LAPS 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 query languages always allow users to save the final query results in relations [S^+79, SWK76]. Under certain conditions, for example, when sorting is performed or nested queries are present, query intermediate results must also be produced to facilitate the query computations. Though theses intermediate results are mostly retained only within the computation of a query, it is not hard to keep them over a longer time for potential reuse. Throughout this paper, we assume intermediate results are generated during query computations. Intermediate and final query results are not differentiated and are both referred as *temporaries*. Cached temporaries are collected and maintained by the cache manager. Several key issues regarding cache management and the corresponding alternative approaches are discussed in the following:

How to store the cached temporaries ?

Temporaries can be stored as actual data in relations [AL80, BLT86], which are called *materialized views*. 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. This is a pointer based approach for caching, variations of this approach have been proposed in [Rou82b, Val87] and called *ViewCache* in [Rou91].

While limited disk space prohibits unlimited data caching, pointer caching is more space-effective since each tuple is represented by a small number of fixed length pointers. However, extra page references to higher level relations or temporaries are required when materializing the tuples from pointer caches, in contrast to materialized data caching.

From the view of query matching, the pointer caching is more attractive than the 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. This makes pointer caching more versatile than data caches. Nevertheless, both pointer and data caching are evaluated by the CMO processor.

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 the suggested heuristic cache replacement strategies in the cache manager and experimented with them under a bounded cache space environment. The purpose here, however, is not to compare amongst the different replacement strategies, but rather observe the performance change of CMO under different available cache spaces, though the results might shed a light into the choice of proper replacement strategy under certain query load.

How to update outdated cached temporaries ?

Cached temporaries become outdated when their composing 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. For the second update method, logs must be maintained to support differential computations.

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 the CMO mechanism 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, the LAPS [Rou82a] is used 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 captures the logical and derivation relationships among them. The integration of new cached temporaries and logical access paths into the LAPS is fairly direct and has been developed in [Rou82a]. A LAPS subcomponent has been embedded in the CMO and allows the coexistence of multiple and equivalent caches which may have been derived from different paths.

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]. The solution usually involves a *theorem proving* algorithm whose computation complexity, in general, is exponential. However, restrictive algorithms were proposed in [Fin82, LY85] for formulas that contain only attributes, constants, comparison operators $(<, >, =)$, and logical connectives (and, or, not). We have extended the method from [Fin82] for more general 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. More detail is described in the next section.

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 mentioned 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 generate an optimal plan. Without elaborate pruning, this approach is not satisfactory because the search space for both phases is extremely large, and even when only a few number of revised queries are produced from the first phase, it can still duplicate the search space for the following phase.

A better approach is to integrate the matching steps with the optimization and thus, unify the search spaces and avoid duplicate effort. [J$^+$93] first described this approach and used a *state transition network* to model the space, along with some pruning heuristics. The CMO optimizer we implemented here is also one phase.

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

In this section, we describe a matching optimizer for the class of *PJS-queries*— queries which involve only relational algebra operators: projection, selection and join. A graph search based algorithm [Nil80] (referred as state transition network in [J$^+$93, LW86]) is used to find the optimal plan. The input to the optimizer is 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. The cost of the access path for this computation or cache is estimated and accumulated in the successive state. Thus, starting from the initial state, successive states are generated until a final state, which represents the totally computed query is reached, and the path with the lowest cost is selected as the optimal plan. We formalize the framework in the following.

A PJS-query q is expressed in SQL as:

$$\texttt{select } \bar{a}_q \texttt{ from } \bar{r}_q \texttt{ where } f_q,$$

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 attributes projected from the relations, and f_q is a boolean formula for which the resulting tuples must satisfy. We can therefore represent any query q as $q = (\bar{a}_q, \bar{r}_q, f_q)$, and view $\bar{a}_q, \bar{r}_q$ and f_q as *sets* as well as *lists*. 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 where $\bar{R}_q$ now contains base relations only. By using 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. We call this the *incremental notation* for v. Similarly, the expanded notation for v is given by $v = (\bar{A}_v, \bar{R}_v, F_v)$.

3.1 The Logical Access Path Schema

The LAPS is a directed graph whose nodes, which reference to base relations and cached temporaries, are connected with edges that represent derivation paths.

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. $x \in N,\ \textit{for all}\ x \in \bar{r}_v$, 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, it is augmented by integrating the cached temporaries along with their logical access paths. The integration steps are straight forward and were described in [Rou82a].

3.2 Query Matching

Given a query, the optimizer must be able to identify the cached temporaries that can be used to compute the query. A temporary is useful if it can be used alone to compute the results of a sub-query of the given query. Formally, we say a (sub-)query q is *derivable* from 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 and content of q and v under instance d, respectively. In the following, we describe without proof the conditions under which a temporary v is *sufficiently* qualified to be a match of a query q.

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

Rather than using a looser condition $\bar{R}_v = \bar{R}_q$, this condition 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 (Qualification Coverability) $\forall x_1, x_2, \ldots, x_n\ (f_q \rightarrow f_v)$, and, there exists a *restricting* formula f^r on v such that $\forall x_1, x_2, \ldots, x_n\ (f_q \leftrightarrow f_v \wedge f^r)$.

$x_1, x_2, \ldots, x_n$ are the attributes appearing in the corresponding formulas, '$\rightarrow, \leftrightarrow$' are symbols for *logical implication* and *logical equivalent.* This condition 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.

The problem of testing $\forall x_1, x_2, \ldots, x_n(f_q \rightarrow f_v)$, known as the *satisfiability problem*, is in general NP-hard [RH80]. However, restrictive algorithms have been proposed for formulas involving only attribute variables, constants, comparison operators and logical connectives [Fin82, LY85]. We have extended the method from [Fin82] to a more general one used in our implementation, detail is given in the Appendix.

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.

This condition 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.

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.*

So far we only talked about how to compute a query by directly selecting from a single temporary, we have not, however, said anything about how to use multiple temporaries in computing a query. This will be achieved by embedding the optimizer with a matching algorithm built based on the knowledge given above. We describe it in the following subsection.

3.3 Integrating Matching with Optimization

The input to the optimizer includes a query and the LAPS. The query is represented by a *query graph*, whose nodes initially correspond to relations, and edges correspond to the query predicates. 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 or an existing cached temporary found in the LAPS. Thus, starting from the initial query graph, successive reduced query graphs (which represent partially computed queries) are generated until the final state, which consists of only one node and no edges, is reached. We use *intermediate results* to refer to those intermediate objects manipulated during the optimization which are not actually computed and cached yet, and formalize the query graph and reductions in the following.

Definition 2 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 $f(e)$.

The query graph is used to model the original query as well as any partially processed queries during the optimization. More precisely, a query graph (N, E) 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 $f_q = \wedge_{e \in E} f(e)$. We say e is a *k-connector* if it connects k nodes, i.e. $|N(e)| = k$. An edge is a *join edge* if $k \geq 2$, it is a *selection edge* if $k = 1$. Since any formula can be transformed into a conjunction of sub-formulas [CL73], the initial query can always be represented by a query graph[2].

A state is reduced by assigning an access path to the sub-query induced by one of its join edges. Given a state $q = (N, E)$ 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

1. $\bar{a}_{q_e} = a(N(e)) \cup (\alpha(N(e)) \wedge \alpha(E^e_{ext}))$,

2. $\bar{r}_{q_e} = N(e)$,

3. $f_{q_e} = f(e) \wedge f(E^e_s)$,

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 formulas labelled on the edges in E^e_{ext}.

There are two different reductions, the *selJoin-reduction* corresponds to computing the induced sub-query directly from the operands of the sub-query; the *match-reduction* corresponds to using a matched cached temporary. 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 dashed rectangle in q_0, is replaced by a new node $i1$ in q_2 which corresponds to a new intermediate result. The access path to evaluate the sub-query is optimized depending on the primitives provided by the underlying DBMS and the cost estimation. Formal definition is given in the following.

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'.

[2]Whenever possible, we transform the initial query into a query graph consisting of only 1-connectors and 2-connectors. However, for some rare predicates, higher degree connectors are required, e.g., 'R1.a + R2.b = R3.c' can only be represented by a 3-connector.

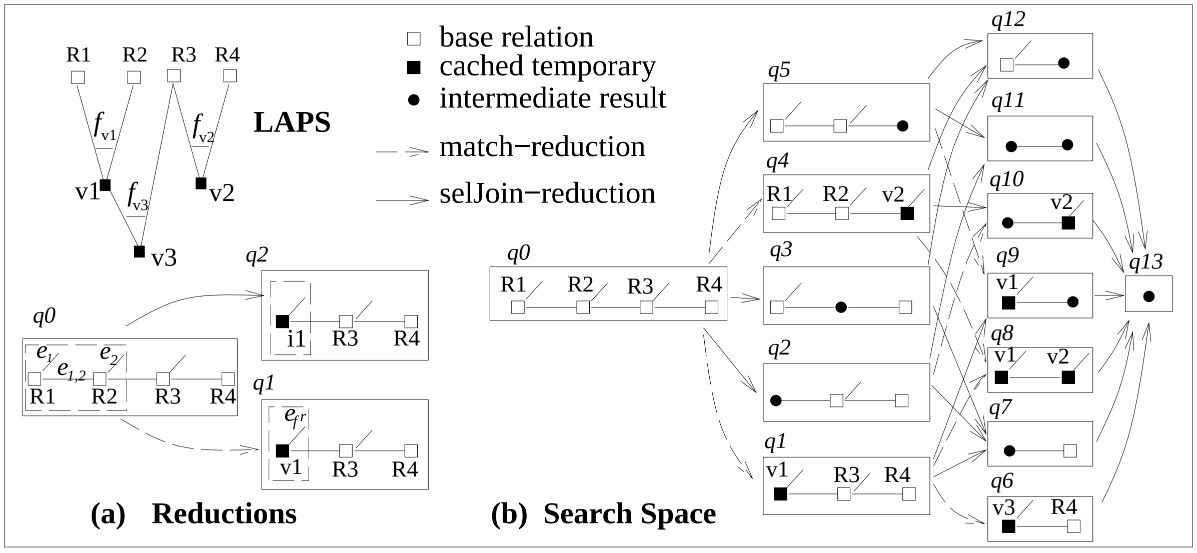

Figure 1: State Reductions and Searching

A match-reduction is also shown in Figure 1.(a), 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 and a selection edge e_{f^r}, suppose $v1$ is a match of $q_{e_{1,2}}$ and f^r is the corresponding restricting formula. Note that $\alpha(v_1)$, the schema of v_1, must contain attributes appearing in both f^r and $f(e_{2,3})$, since they are not evaluated yet. This is assured in the attribute coverability check (Condition 3) of $v1$ against $q_{e_{1,2}}$. The formal definition follows.

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 upon n_v and is labelled with f^r.

We have customized our optimizer so that the LAPS is traversed top-down in parallel with the selJoin-reductions and match-reductions, and no any node in the LAPS is visited more than once. This saves the effort of searching for potential matches in two ways: (1) once a temporary is known as not being a match, all its descendants can be rejected automatically without further checking, (2) the qualification coverability checks are performed in an amortized style so that only the incremental formulas f_v (probably with some propagated restricting formulas) instead of the expanded F_v, are involved.

Based on these two reductions, a *dynamic programming* searching strategy is used to find the optimal plan[3]. It performs a breadth-first search and restricts the state space by eliminating *isomorphic* states. Formally, two states $q_x(N_x, E_x), q_y(N_y, E_y)$ are isomorphic, denoted as $q_x \cong q_y$, if there exists a 1-to-1 mapping $\psi : N_x \rightarrow N_y$ such that for each pair of $v \in N_x$ and $\psi(v) \in N_y$, they either denote the same base relation or cached temporary, or they are both intermediate results and $\bar{R}_v = \bar{R}_{\psi(v)}$[4]. The cost of a state, *cost*(q), is computed as the total cost of the path leading from the initial state to state q. The cost model of the ADMS optimizer 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 the following,

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 $q' \in T$ such that $q' \cong q_1$, then $T := T \cup \{q_1\}$, otherwise if $cost(q') > cost(q_1)$ then $T := T - \{q'\} \cup \{q_1\}$. Do the same thing for q_2.

Step 3 Output the 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 plan of using two matched temporaries v_1 and v_2. Isomorphic states are reflected by those arrows that come into the same state.

4 Experiments: Performance Evaluation

A CMO component has been incorporated in the ADMS. It sits on top of the storage access module which provides a single *selJoin* operator. A *selJoin*, with an appropriate environment support, replaces the three selection, projection and join operators. In essence it is a join operator in which the inner relation may be empty, in which case, it becomes a straight forward selection operator on the outer relation. Projection of a subset of attributes and duplicate elimination is supported on the fly during output using main memory hashing. In today's large main memory computing environments, duplicate elimination can be done more

[3]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 and finding the "real" optimal plans.

[4]Different definitions of isomorphism were implemented and experimented in ADMS, the one given here turned out to have a manageable searching overhead while not sacrificing too much in the quality of the output plan.

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

efficiently and very often at no I/O cost. Two access methods, sequential and index access, are provided for each relation. Three join methods: nested loop, index, and hash join are available for $selJoin$.

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.

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. The set of shorter version is obtained from the regular one by eliminating the last two string attributes, which are 104 bytes in length. Throughout the experiments, each tested database consists of four relations from Table 1. Table 2 lists all the used databases, note that the primes ($'$) indicate the different relation instances of the same relation 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. First, except for the update query frequency, we assigned equal-frequency among single-relation selection and 2 to 4-way join queries. Second, for each query stream against a certain database, the query selectivities are set within a wide range so that the numbers of tuples generated in the query results range from less than a hundred, a couple hundreds and thousands, to several hundred thousands.

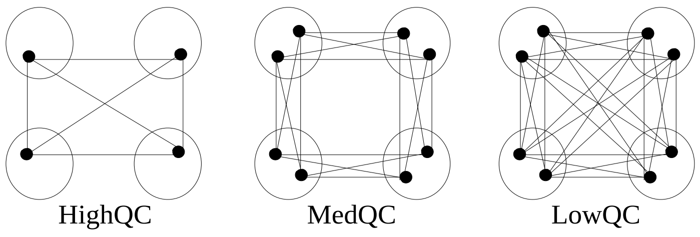

Figure 2: Three Levels of Query Correlations

For each n-way join query, the qualification is formed by $n-1$ two-way join predicates and a few random selection predicates. The generation of these predicates depends on both the selectivities and attribute reference distribution specified for the query stream, where the latter assigns each attribute with two probabilities of being used in a join and selection predicate. 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. Intuitively, the *query correlation* of a query stream can be measured as the number of distinct equal-join predicates appearing in it. Figure 2 shows three classes of query correlations used in generating the tested 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 caching, but requires more space for data caching. Updates are restricted to the last three string attributes to maintain the cardinalities during the experiments. However, the qualifying predicates in the updates 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 tested database and the query correlation are DBMIX and MedQC respectively

Performance Metrics

The total *elapsed time* of a query stream, including query optimization time and query computation time, is taken as the main metric in 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 variations of CMO (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 of this subsection.

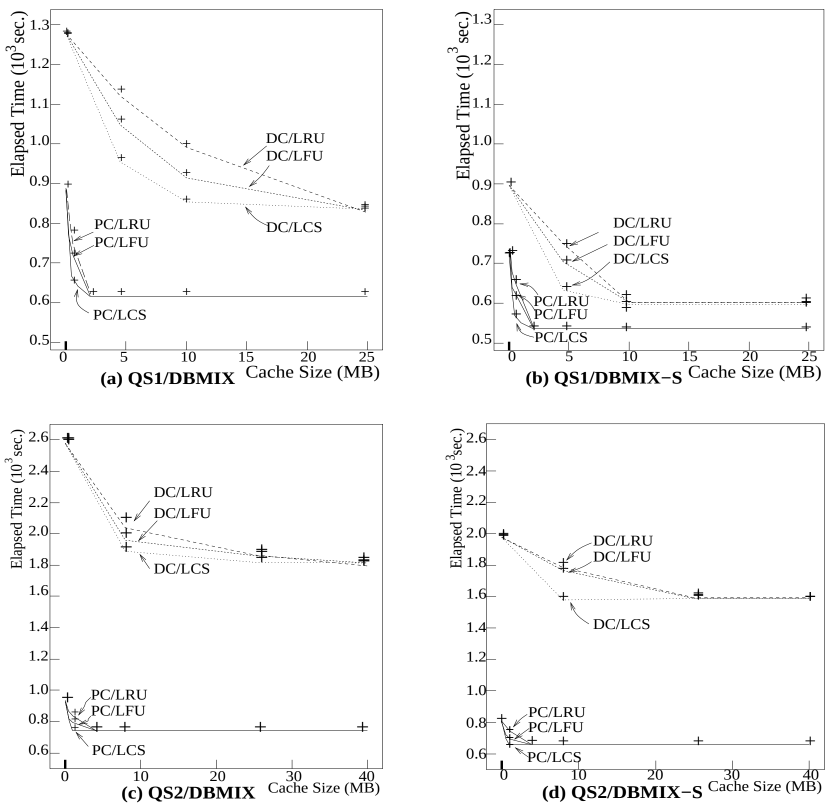

Figure 3: Effect of Cache Management

4.2.1 Effect of Cache Management

We compare the performance of data caching (DC) and pointer caching (PC) under three different 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

These are shown on the six curves labelled accordingly in Figure 3 for each combination of the databases (DBMIX, DBMIX-S) and query streams (QS1, QS2) we tested.

As can be seen all PC curves are roughly the same but among them, PC/LCS is the best. In set QS1/DBMIX, the pointer caching runs faster than the data caching approach 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. As the cache space increases, the PC reduces the query processing time sharply even with a very small cache space. In this case, with 2 MB cache space, every useful temporaries are cached under LCS strategy. For DC, all three replacement strategies reduce the elapsed time as cache space increases, among them, LCS seems to be the best. However, 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 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 PC in performance. Some savings can be achieved 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. The results from another query stream QS2 are also shown in Figure 3.c and d, and as can be seen, are similar to those from QS1.

If cache effectiveness is measured as the time reduced in query computation divided by the cache size, it is clear that pointer caching has much better cache effectiveness than data caching. And for this reason, we believe that pointer caching is the proper choice in implementing a CMO mechanism. In the rest experiments, only pointer caching with LCS replacement strategy 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 the raw stream with 5 different degrees of relation update frequencies and 3 different levels of relation 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).

Four different query streams are experimented, Figure 4 depicts the average throughputs among all four query streams under different situations. At a first glance, the CMO curves, no matter what update strategies are used, perform better than REG in all LS, MS and HS

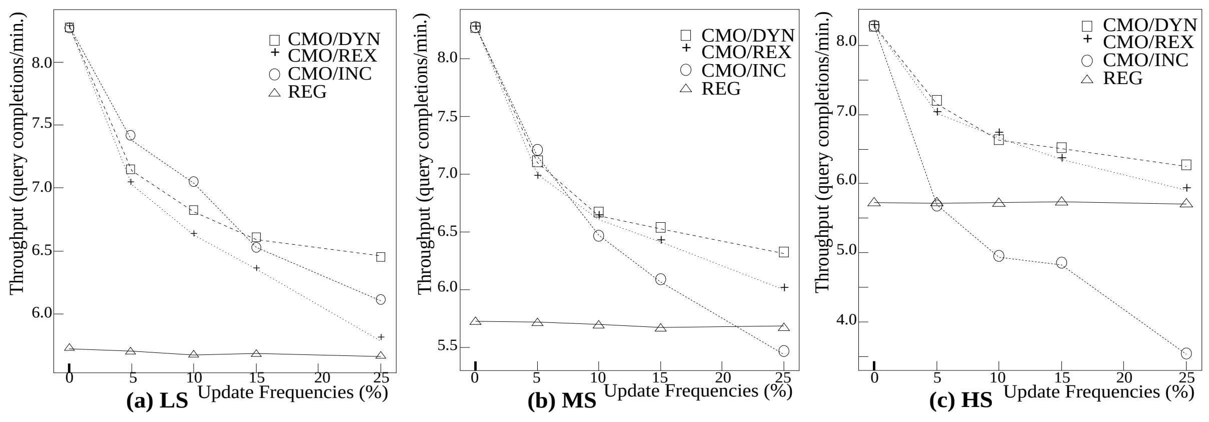

Figure 4: Effect of Relation Updates

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 amortized among those subsequent queries that are able to use it before it becomes outdated again. And of course, as update frequency increase, the cost is amortized among fewer queries and thus the throughput decreases.

In 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 attributed to the increase of update logs processing as the update frequency increases. This makes CMO/INC less advantageous. In 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 readers might wonder why CMO/DYN, which is supposed to be theoretically more informed 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 CMO/DYN sometimes chose less efficient paths than CMO/INC and/or CMO/REX. This is due to the inaccuracy of the cost estimation which may cause the wrong choice between incremental and re-execution update when their costs are closed. Even if the optimizer always predicts accurate costs, choosing the optimal plan for individual query does not guarantee the global optimality for all queries. Therefore, CMO/DYN sometimes decides not to use a outdated cache (because of its high update cost) for the current query whose high cost actually can be compensated by the speed-up of some follow-up queries which use it. These two problems of *accurate cost estimation* and *multiple query optimization* actually are general problems to query optimization but not within the scope of this paper. 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.

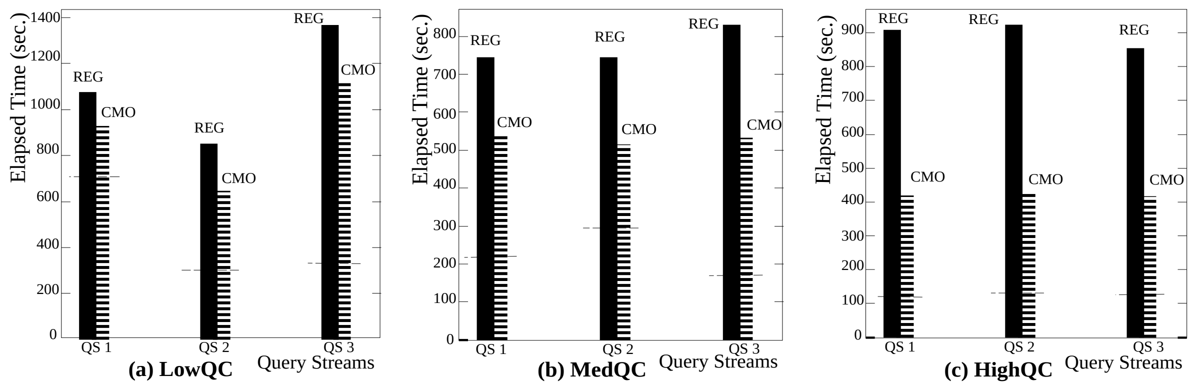

Figure 5: Effect of Query Correlations

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. Figure 6 depicts the *matching rates* for the three classes, where matching rate is computed as the number of matched temporaries used in the queries divided by the total number of $selJoins$ performed in the queries. The curve for each class is obtained by averaging among the three query streams. It can be seen from the figure that the higher the query correlation is, the faster and higher the corresponding matching rate curve grows.

We also observed the effect of query selectivities. Figure 7 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 8 shows the results, where CMO consistently shorten the 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 described above. Though CMO

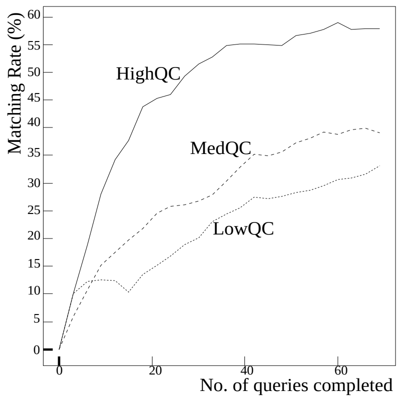

Figure 6: Matching Rates

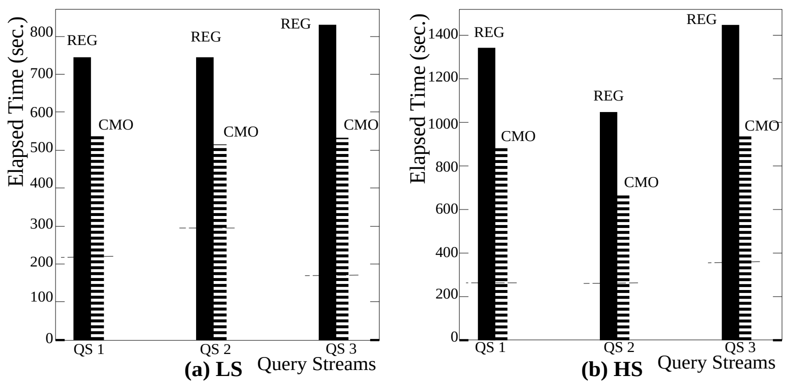

Figure 7: Effect of Query Selectivities

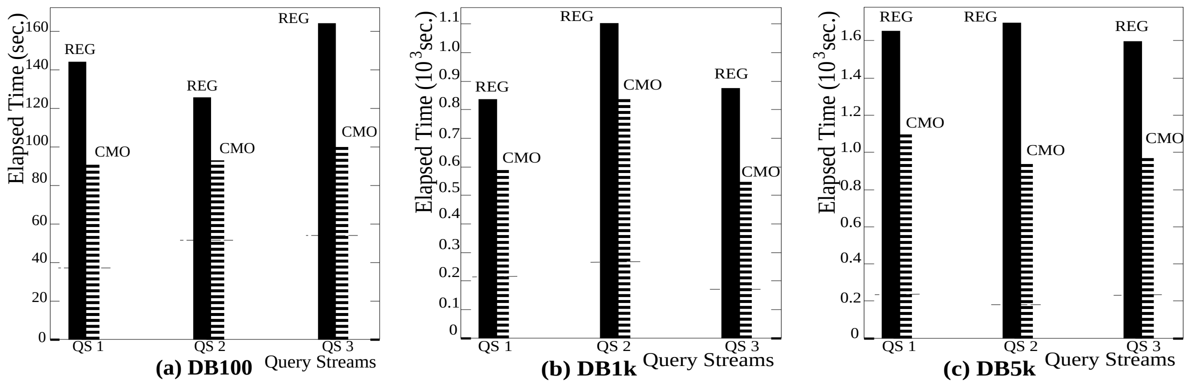

Figure 8: Effect of Database Size

has around 50% – 60% more optimization time than REG, the extra overhead introduced by CMO is relatively small when compared to the time saved in query computations. Note that this has been demonstrated in all the above experiment results where the elapsed time includes both query optimization and computation time.

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

Table 3: Ave. Optimization Time (second/per query)

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 apply the ADMS-CMO techniques to concurrent queries and investigate how results cached from a query (user) can be used in another concurrent query (user). Also we would like to extend the CMO optimizer for 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.

Appendix

The satisfiability problem is to test if $\forall x_1, x_2, \ldots, x_n \ (f_1 \rightarrow f_2)$ is true. That is, for all possible values of attributes $x_1, x_2, \ldots, x_n$, if f_2 always evaluates to true whenever f_1 is true. And if this is the case, then find a *restricting formula* f^r such that $\forall x_1, x_2, \ldots, x_n \ (f_1 \leftrightarrow f_2 \wedge f^r)$. We will consider query qualifications constructed from attributes, constants, comparison operators $(>, =, <)$, and logical connectives $(\wedge, \vee)$. The same problem has been addressed in [LY85], who transformed it into a directed graph problem. However, it was not clear how the restricting formula can be constructed and how efficient the algorithm is. Therefore, we used a more efficient but restrictive algorithm extended from [Fin82].

An *atom* is a predicate of the form $x \ \theta \ y$, where x is an attribute, y is either an attribute or a constant, and θ is a comparison operator. A *clause* is a disjunction of atoms, denoted as $C = A_1 \vee A_2 \vee \ldots A_n$. From elementary logic [CL73], we can transform each formula into a conjunctive form as $f = C_1 \wedge C_2 \wedge \ldots C_m$. The following lemma gives sufficient conditions and a constructive way for the satisfiability check. Note that all universal quantifiers are bound to the attributes appearing in the scoped formula.

Lemma 2 *1. $\forall(f_1 \rightarrow f_2)$ if and only if, for each $C_{2,j} \in f_2$, $\forall(f_1 \rightarrow C_{2,j})$*

2. if there exists a $C_{1,i} \in f_1$ such that $\forall(C_{1,i} \rightarrow C_{2,j})$, then $\forall(f_1 \rightarrow C_{2,j})$

3. if for each $A_i \in C_{1,i}$, there exists a $A_j \in C_{2,j}$ such that $\forall A_i \rightarrow A_j$, then $\forall(C_{1,i} \rightarrow C_{2,j})$

What remains to be solved is the testing of $\forall A_i \rightarrow A_j$. This test can be easily checked when A_i, A_j contain no arithmetic operators. First, it evaluates to false if A_i and A_j contain different attributes. If they have the same attributes, then a look-up table, as shown in Table 4, is used. The entry in table (a) indicates the relationship between constants c and c' under which $(x \ \theta_1 \ c) \rightarrow (x \ \theta_2 \ c')$ is true, a blank entry means under no situations can it be true. In table (b), a checked entry indicates that $(x \ \theta_1 \ y) \rightarrow (x \ \theta_2 \ y)$ is true, where both x and y are attributes.

$\theta_1 \backslash \theta_2$	$x = c'$	$<$	$\leq$	$>$	$\geq$
$x = c$	$=$	$<$	$\leq$	$>$	$\geq$
$<$		$\leq$	$\leq$		
$\leq$		$<$	$\leq$		
$>$				$\geq$	$\geq$
$\geq$				$>$	$\geq$

(a) Unary Atoms

$\theta_1 \backslash \theta_2$	$x = y$	$<$	$\leq$	$>$	$\geq$
$x = y$	$\surd$		$\surd$		$\surd$
$<$		$\surd$	$\surd$		
$\leq$			$\surd$		
$>$				$\surd$	$\surd$
$\geq$					$\surd$

(b) Binary Atoms

Table 4: Look-Up Table for $A_i \rightarrow A_2$

When $\forall(f_1 \rightarrow f_2)$ is true, we also need to find a restricting formula so that the tuples satisfying f_1 can be extracted from the tuple satisfying f_2. The simplest restricting formula is f_1 itself since $\forall(f_1 \leftrightarrow f_2 \wedge F_1)$. However, in some cases, a restricting formula with fewer predicates can be computed from f_1. This may benefit the query computation since fewer predicates need to be evaluated. The following lemma tells how to get a restricting formula by eliminating redundant clauses from f_1.

Lemma 3 *Suppose* $\forall(f_1 \rightarrow f_2)$ *is true. If there exist clauses* $C_1 \in f_1, C_2 \in f_2$ *such that* $\forall(C_1 \leftrightarrow C_2)$*, then* $f^r = f_1 - \{C_1\}$ *is a restricting formula, i.e.* $\forall(f_1 \leftrightarrow f_2 \wedge f^r)$*.*

Note that $\forall(C_1 \leftrightarrow C_2)$ can be detected by checking both $\forall(C_1 \rightarrow C_2)$ and $\forall(C_2 \rightarrow C_1)$. An algorithm based on the above lemmas is embedded in the ADMS Cache&Match optimizer. The algorithm is *sound* but *not complete* in the sense that it is always correct when it returns true for the satisfiability check, but might return false when actually the answer should be true. However, this will not error the query processing, except that in some rare cases where query expressions are complicated, potential useful cached temporaries might be ignored. We think this is acceptable since covering all cases needs a more general algorithm of prohibitive computation complexity.

References

[AL80] M.E. Adiba and B. G. Lindsay. Database snapshots. In *Procs. of 6th VLDB*, 1980.

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

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

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

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

[Fin82] S. Finkelstein. Common expression analysis in database applications. In *Procs. of ACM-SIGMOD*, pages 235–245, 1982.

[Han87] E.N. Hanson. A performance analysis of view materialization strategies. In *Procs. of ACM-SIGMOD*, pages 440–453, 1987.

[HS93] J.M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Procs. of ACM-SIGMOD*, 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 14th VLDB*, 1988.

[LHM86] B. Lindsay, L. Haas, and C. Mohan. A snapshot differential refresh algorithm. In *Procs. of ACM-SIGMOD*, 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 11th 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. *To appear in IEEE Trans. on Knowledge and Data Engineering*, 1993.

[RH80] D.J. Rosenkrantz and H.B. Hunt. Processing conjunctive predicates and queries. In *Procs. of 6th VLDB*, 1980.

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS$\underline{+}$. *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 ACM-SIGMOD*, pages 23–34, 1979.

[Sel87] T. Sellis. Efficiently supporting procedures in relational database systems. In *Procs. of ACM-SIGMOD*, 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.