The ADMS Project: Views "Я" Us *

Nick Roussopoulos   Chungmin M. Chen   Stephen Kelley
Department of Computer Science
and
Institute of Advanced Computer Studies
University of Maryland
College Park, MD 20742
{nick,min}@cs.umd.edu, skelley@umiacs.umd.edu

Alex Delis
Department of Information Systems
Queensland University of Technology
Brisbane, Australia
ad@icis.qut.edu.au

Yannis Papakonstantinou
Stanford University
Department of Computer Science
Stanford, CA 94305
yannis@DB.Stanford.EDU

Abstract

The goal of the ADMS project is to create a framework for caching materialized views, access paths, and experience obtained during query execution. The rationale behind this project is to amortize database access cost over an extended time period and adapt execution strategies based on experience. ADMS demonstrates the versatility of the views and their role in performance, data warehousing, management and control of data distribution and replication.

1 Introduction and Motivation for ADMS

The main goal of the *Adaptive Database Management System (ADMS)* project is to capitalize on the reuse of retrieved data and data paths in order reduce execution time of follow-up queries. From its inception, ADMS is trying to satisfy a need that has been neglected, that is, learning to perform better with experience obtained by query execution. Although database management systems use sophisticated query optimization techniques and access methods, they neither gather nor retain any experience from query execution and/or obtained results. For example, the cost of an execution plan generated by the optimizer is not compared to the actual cost incurred during the execution in order to adapt strategies during follow-up plans. Similarly, data retrieved during query execution has a very short life span in the buffer area of the database systems and is not retained on disk for future use. ADMS's goals include capturing knowledge that affects query optimization, such as attribute value distributions, selectivities, and page faults, and creating a framework for caching materialized views, access paths, and amortizing their maintenance cost.

*This research was partially sponsored by the National Science Foundation under grants IRI-9057573 and GDR-85-00108, by NASA/USRA under contracts 5555-09 and NAG 5-2926, by ARPA under contract 003195, by the Institute of Systems Research, and by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

Query execution cost can be reduced by reusing intermediate and/or final query results cached in *Materialized View Fragments (MVFs)*. These MVFs can then be accessed efficiently at only a fraction of the cost of their initial generation. Access paths can also be cached in the form of *ViewCache* pointers [Rou91] which are trail markers captured during the execution of queries. They point to base relation tuples and/or to lower level ViewCache entries satisfying a query and may be used by the system during subsequent queries to walk through the same data paths without search.

Subsumption is a common technique for discovering useful MVFs and ViewCaches which contain (subsume) the results of a given query. Such a facility is necessary in any large data warehouse in which the catalog cannot be manually browsed. Subsumption goes hand-in-hand with validation techniques such as cache coherence [FCL92] and relevant update [BLT86]. These are techniques for deciding whether or not cached data is affected by updates. However, since these techniques were developed for short-lived caches, they are of rather limited value for the long term disk caching of a warehouse. To remedy this, ADMS uses *incremental access methods* [RES93] for propagating updates to the MVFs and ViewCaches and, thus, prolongs their useful life span. The ADMS optimizer [CR94b] uses a subsumption algorithm for discovering applicable MVFs and ViewCaches, incremental access methods for updating MVFs and ViewCaches [Rou91], and an amortized cost model in evaluating alternative execution plans.

The adaptive functionality of ADMS captures not only access paths, intermediate and final results, but also the experience obtained during their use in query execution. The ADMS buffer manager observes page faults during execution of queries and builds a "page fault characteristic curve" which predicts page faults under different buffer availability situations [CR93]. These predictions are then used by the system for selecting global buffer allocation strategies. Similarly, ADMS exploits knowledge captured inside MVFs and ViewCaches. The selectivity of operators and cardinality of predicates are harvested during the creation, use, and update of ViewCaches and MVFs, and are fed back to an "adaptive curve-fitting" module which obtains accurate attribute value distributions with minimal overhead [CR94a].

This philosophy of caching query results has been extended in the *ADMS±* Enhanced Client-Server database architecture [RK86b, RK86a, RD91, RES93, DR92, DR93]. MVFs are dynamically downloaded from multiple heterogeneous (commercial) server DBMSs to clients running ADMS–, a single user client version of ADMS. ADMS– creates its own data warehouse by caching downloaded results in replicas, incorporating them in locally processed queries, and making incremental update requests from the servers holding their primary copies. ADMS– allows the user to create composite views from multiple heterogeneous DBMSs and enables him to integrate them with local and proprietary data. An extension of the ADMS subsumption algorithm is to find a "best fit" set of MVFs residing on multiple clients for answering a given query. In this technique, the cost is affected by the number of fragments used and the negation predicates which preclude duplicates from the final result [Pap94].

In this paper, we outline the motivation, rationale, concepts, techniques, and the implementation of the University of Maryland ADMS prototype. Section 2 of this paper describes the view engine of ADMS. Section 3 outlines the ADMS optimizer. Section 4 describes the distributed *ADMS±* Client-Server architecture and the management of replicas. Section 5 contains concluding remarks, a brief historical account of ADMS, and current developments.

2 The ADMS View Engine

ADMS uses traditional relational storage organization and standard access methods including sequential scan, B-trees, R-trees and hashing. The ADMS catalogs store names, locations, cardinalities, selectivities, etc. for all databases connected to the warehouse, relations in them, attributes, and indexes. They are the core resource for ADMS query validation and optimization, and are maintained in base relations so they can be queried through ADMS's standard SQL interface. The two novel extensions of the core ADMS system are its ViewCache storage organization and its adaptive buffer manager.

2.1 The ViewCache Storage and the Incremental Access Method

The ViewCache [Rou91] is the most innovative feature and has the most far reaching consequences of any caching technique in ADMS. It is a persistent storage structure consisting of pointers to data objects which are either base relation tuples or pointers in lower level ViewCaches.

The ViewCache storage structure and its incremental access methods are built around a single relational operator we call *SelJoin*. *SelJoin* corresponds to a join with concurrent selections on its arguments. The outer relation can be empty in which case a *SelJoin* reduces to a simple selection. Three join methods have been implemented for *SelJoin*, nested loop, index, and hash join. Each ViewCache corresponds directly to a single application of *SelJoin* and is, therefore, a 1- or 2-dimensional array of pointers (TIDs) depending on the number of arguments of *SelJoin*. The TIDs point to records of the underlying relations or other views necessary to make up the result of the *SelJoin*. A multiway join view is then expressed as a tree of *SelJoin* ViewCaches with the base relations at the leaves. Every ViewCache in the hierarchy is maintained for the life of the view and the expression which defines each subtree is inserted into the catalogs so that ViewCache fragments can be reused, possibly in other views. Since ViewCaches contain only TIDs, they are relatively small in size and can be constructed, quickly materialized (dereferenced), and actively maintained with little system overhead.

All other relational operators such as projection, duplicate elimination, aggregate, ordering, and set theoretic operators are performed on the fly during output, using masking, hashing, and main memory pointer manipulation routines. The advantages of having a single underlying operator are manyfold: Firstly, the optimizer does not have to consider pushing selections or projections ahead of joins or vice versa. Secondly, subsumption on *SelJoin* ViewCaches is more general when no attributes have been projected out. Lastly, the incremental algorithms for *SelJoin* ViewCaches are simple and require no heavy bookkeeping as opposed to incremental update algorithms for projection views or views containing other aggregate operators which have significant complexity and require sophisticated bookkeeping and expensive logging.

ViewCaches are maintained in ADMS by its *Incremental Access Method (IAM)* which amortizes their creation and update costs over a long period of time (indefinitely). IAM maintains update logs which permit either eager or deferred (periodic, on-demand, event-driven) update strategies. The on-demand strategy allows a ViewCache to remain outdated until a query needs to selectively or exhaustively materialize the underlying view. The IAM is designed to take advantage of the ViewCache storage organization, a variation of packed R-trees [RL85]. This organization attempts to reduce the number of intermediate node groupings in the R-tree. This number is the most significant parameter in determining the cost of materializing the ViewCaches. Both ViewCache incremental update and tuple materialization (dereferencing) from it (ViewCache) are interleaved using one–pass algorithms. The interleaved mode avoids the duplication of retrieving the modified records to be updated and then materialized again for the remaining of the query processing.

Compared to the query modification technique for supporting views that requires re-execution of the definition of the view, and to the incremental algorithms for MVFs [TB88], IAM on ViewCaches offers significant performance advantages, in some cases up to an order of magnitude. The decision about whether or not an IAM on a ViewCache is cost-effective, i.e. less expensive than re-execution, depends on the size of the differentials of the update logs between consecutive accesses of the ViewCaches. For frequently accessed views and for base relations which are not intensively updated, IAM by far outperforms query modification [RES93]. Performance gains are higher for multilevel ViewCaches because all the I/O and CPU for handling intermediate results is replaced with efficient pointer manipulation.

2.2 The ADMS Adaptive Buffer Manager

ADMS uses an adaptive allocation scheme to allocate buffers from the global buffer pool to concurrent queries. Page reference behavior of ViewCache materialization and recurring queries involving MVFs are quantified using page fault statistics obtained during executions [CR93]. This page fault information is fed back to the buffer manager and gets associated with each MVF and/or ViewCache. An "adaptive curve-fitting" module is used to capture the *Marginal Gain Ratio (MGR)* on page faults, i.e. the faults reduction per additional buffer allocated to a query using a ViewCache. ADMS's buffer manager basically identifies two important characteristics, the "critical size", that is the number of buffers beyond which the reduction of faults starts diminishing, and the "saturation size", the number of buffers beyond which no reduction is attained. As queries utilizing MVFs and ViewCaches recur, the buffer manager observes, adapts, and saves their characteristics to continuously capture the effects on page faulting as the database changes in time.

ADMS allocates buffers to these queries according to their page fault characteristics and the global buffer availability. Buffers are allocated to individual queries/relations in proportion to their average Marginal Gain Ratios (MGR) subject to the following constraints: (a) never allocate more than the saturated size (avoid waste), and (b) when the demand for buffers is high, never exceed the critical size of each reference string.

Experimental results in ADMS validated the advantage of MGR over traditional methods such as global LRU and DBMIN [CD85]. Through a comprehensive set of ADMS experiments, we demonstrated that MGR offers significant performance improvement over a pattern prediction-based algorithm [NFS91] and a load control-based algorithm [FNS91]. In all cases of query mixing and under various degree of data sharing, on the average, MGR outperforms the second best strategy by 15% – 30% in query throughput.

The merit of the MGR allocation scheme can be attributed to the feedback of faulting characteristics, which provides more insightful buffer utilization information than the probabilistic analysis-based methods which rely on the infamous crude assumption of uniformity.

3 The ADMS Query Optimizer

The ADMS query optimizer invokes a subsumption matching algorithm to identify relevant ViewCaches and generates alternative query plans utilizing those matched ViewCaches. It then selects between incremental or re-execution update strategies depending on their corresponding projected costs. Another key feature of the ADMS query optimizer is its adaptive selectivity estimation mechanism. This mechanism provides cost-effective and accurate selectivity estimation using *query feedback*. In essence, this feedback is information contained in the MVFs and ViewCaches constructed or updated during prior query processing. Minimal (CPU only) overhead is incurred to compute and adaptively maintain selectivity statistics. With this approach, ADMS completely avoids the overhead of the traditional off-line access of the database for gathering value distribution statistics. The first subsection describes the ADMS query optimizer; the second subsection describes the technique of adaptive selectivity estimation.

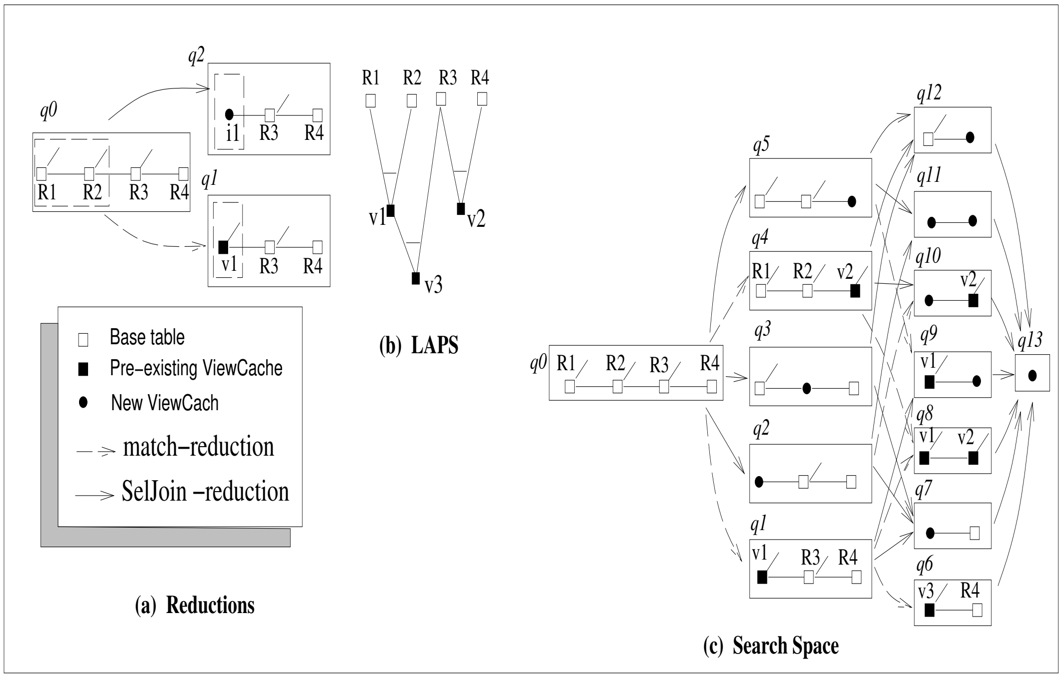

Figure 1: LAPS, Query Graph Reductions, Subsumption Reductions Rules and Search Space

3.1 Subsumption Driven Optimization

The query optimizer uses a dynamic programming graph search algorithm for obtaining an efficient query execution plan. A query graph can be reduced to another one generating the same final result either by a *SelJoin-reduction* or by a *match-reduction*. A *SelJoin-reduction* corresponds to the execution of a *SelJoin* for evaluating a fragment of the query graph; a *match-reduction* corresponds to answering the query graph via a matched, pre-existing ViewCache that subsumes the reduced fragment. Figure 1.a illustrates the two different kinds of reductions that can be applied to a query graph. The Logical Access Path Schema (LAPS) [Rou82] is used to organize the ViewCaches in a directed acyclic graph data structure. This is built from the database catalogs containing the definition of ViewCaches along with their derivation paths. Figure 1.b shows a LAPS with four base relations and three ViewCaches. The search of the query optimizer traverses top-down the LAPS structure and applies *match-reductions* in a breadth-first manner.

Subsumption is used during the *match-reduction* for deciding whether or not a more general ViewCache V "logically implies" a query fragment q. This is in general an undecidable problem, but in the context of database queries it is an NP-hard problem [RH80]. Although several sophisticated algorithms have been proposed [LY85, S$^+$89], we adopted a rather simple subsumption algorithm which extends the "direct elimination" technique proposed in [Fin82] and has worst case complexity of $O(mn)$, where m and n are the number of predicates in V and q respectively. The subsumption algorithm is *sound*, in the sense that it answers positively only when the implication statement is valid, but *not complete*, in the sense that it may not discover all possible subsumable clauses.

For each reduction, the cost of performing the $SelJoin$ from scratch or accessing the matched ViewCache is estimated and accumulated in the reduced query graph. If the ViewCache is outdated, the cost includes both alternatives, the projected cost of an incremental update and the projected cost of a re-execution of the ViewCache. Thus, starting from the initial query graph, the search algorithm generates successive query graphs until a single node graph is obtained which represents the query's result. The path with the lowest cost is then selected for execution. Figure 1.c shows the search space for the query and LAPS given in figure 1.a and b.

The performance of the ADMS query optimizer was evaluated by running a comprehensive set of experiments using a benchmark database and synthetic queries. By turning off the subsumption algorithm and the incremental access methods, the ADMS optimizer reduces to the System R [SAC+79], thus allowing us to make comparisons. The experiments showed that ViewCaches with subsumption and dynamic selection between incremental update and re-execution of MVFs and ViewCaches save substantial query execution time, and thus, increase the overall query throughput under a variety of query and update loads [CR94b]. The improvement ranged between 30% - 60% in query throughput under moderate update loads while it suffered no loss under heavy update loads. Although query optimization cost is increased by up to one tenth of a second per query, this overhead is insignificant when compared to query execution reduction of seconds or tens of seconds.

3.2 Adaptive Selectivity Estimation

The most significant factor in evaluating the cost of each query execution plan is *selectivity* – the number of tuples in the result of a relational selection or join operator. Various methods based on maintaining attribute value distributions [Chr83, PSC84, MD88, SLRD93, Ioa93] and query sampling [HOT88, LN90, HS92] have been proposed to facilitate selectivity estimation.

ADMS uses and adaptively maintains approximating functions for value distributions of attributes used in query predicates. We implemented both polynomials and splines[1] and an "adaptive curve-fitting" module which feeds back accurate selectivity information after queries and updates [CR94a]. The CPU overhead of adapting the coefficients of the approximating functions is hardly noticeable but the estimation accuracy of this approach is comparable to traditional methods based on periodic off-line statistics gathering or sampling. However, unlike these methods, the query feedback of ADMS incurs no I/O overhead and, since it continuously adapts, it accurately reflects changes of the value distributions caused by updates in the database.

In all our experiments, the adaptive selectivity estimating functions converge very closely to the actual distribution after 5 to 10 query feedbacks of random selection ranges on the attribute. For a rather staggered actual distribution, it takes almost the same time to converge to a stable curve. In such a case, the resultant curve may not fit seamlessly to the actual distribution due to the smoothness nature of the polynomials, but it represents the optimal approximation to the actual distribution in the sense of least square errors.

4 ADMS±: An Enhanced Client-Server Database Architecture

Commercial Client-Server DBMS architectures, which exemplify primary copy distributed database management, have significant performance and scalability limitations. Firstly, record-at-a-time navigation through their interfaces is way too slow to be functional. Secondly, a dynamic SQL interface is rather restrictive, simply because no optimized plans can be submitted; the user is forced to use the server's optimization services as they are. Finally, these architectures do not scale up because all database accesses (I/O) and processing are done on the server.

The *ADMS±* architecture, [RK86b, RK86a], preceded all known database client-server architectures and introduced the concept of a client DBMS that cooperates with the servers not only for query processing, but also for data accessing from replicated MVFs dynamically downloaded onto their disks.

[1]Splines are piecewise polynomials and give the database administrator the flexibility of choosing parameters which best fit the application, such as the degree and the number of polynomial pieces.

Figure 2 shows the $ADMS\pm$ architecture with two commercial database servers and three clients. The ADMS– client has all the capabilities of ADMS, but is usually run in single user mode (indicated by the – sign). The ADMS+ component on the figure is the gateway layer that runs as an application on top of the underlying DBMS.

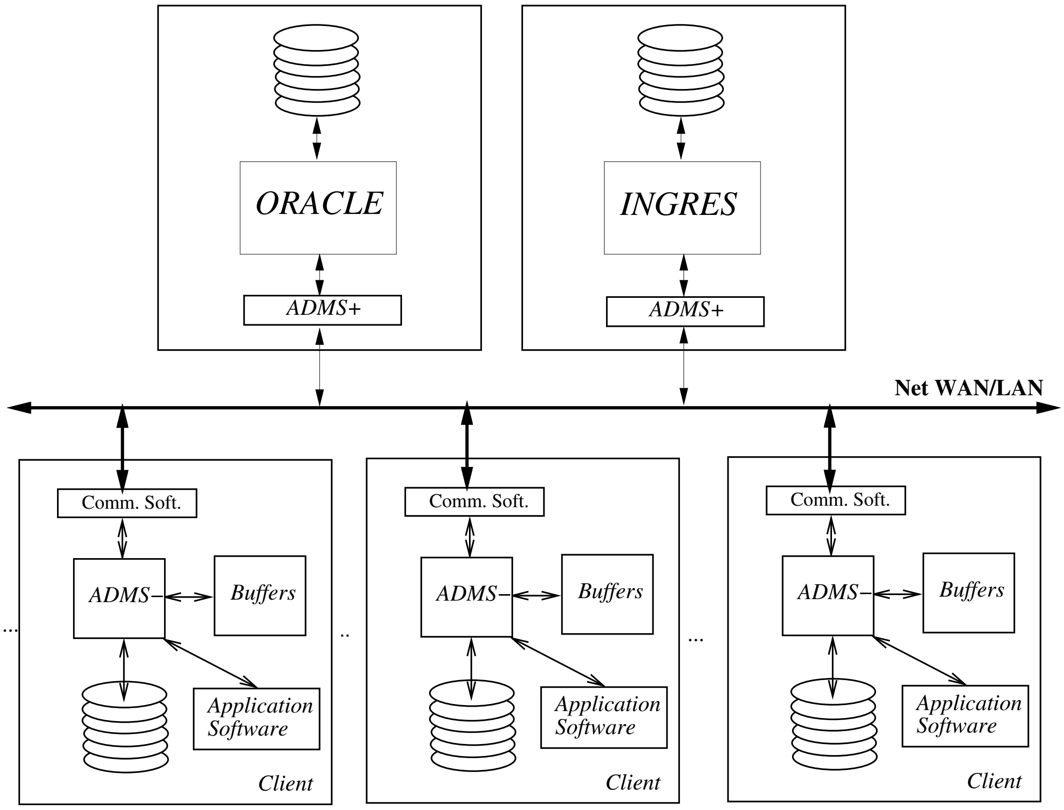

Figure 2: The $ADMS\pm$ Client-Server Architecture

A user on an ADMS– client can connect to a number of commercial DBMS servers and make queries against their databases. Query results are downloaded and incrementally maintained as MVFs on the client ADMS–. Again subsumption of views and incremental access methods provide the foundations for efficient and controlled management of the data replication. Since ADMS– views can be glued from multiple global heterogeneous sources, i.e. server-server joins, combined with possibly proprietary data on the client, i.e. server-client joins, $ADMS\pm$ became the first conceived *Data Warehousing Architecture* and is operational since 1989 [Eco89, RES93].

Updates from a client are sent to the primary copy on the server(s) and find their way to the downloaded MVFs through incremental update algorithms by transmitting the log differences [RES93]. $ADMS\pm$ supports eager and a number of deferred update propagation strategies including on-demand, periodic, and event driven. The deferred strategies allow $ADMS\pm$ to operate under "weak connectivity" mode in which clients can be disconnected and reconnected at later time. Again, MVFs and their incremental update algorithms provide the foundations of this architecture.

The value of the $ADMS\pm$ architecture is three-fold. First, it distributes asynchronous and parallel I/O to the clients, alleviating the I/O bottleneck on the servers [DR92]. This is a significant performance booster because of the exhibited scalability of the architecture. Second, it provides a controlled mechanism for managing replication and update propagation [DR94]. Third, by caching MVFs on the clients it permits client mobility and database access in disconnected mode, whereby a client uses its local disk when some of the database server(s) are unaccessible.

4.1 Performance of *ADMS±*

We extensively studied the *ADMS±* Enhanced Client–Server architecture, [RD91, DR92, DR93] and showed that it (a) outperforms all other client-server architectures including those with equal number of disks replicating their data, (b) scales up very nicely and reaches linear scalability on read- and append-mostly databases. The studies showed that the distribution of the I/O to the client disks and the parallelism obtained this way are the main contributors to the performance and scalability.

In [DR94], we proposed a number of update propagation techniques for the *ADMS±* architecture. We then studied performance under various workloads and scalability as the number of participating clients increases. We showed that, under high server resource utilization, a simple broadcasting strategy for server updates gives better performance than any other update propagation policy. However, when none of the server resources reach full utilization, on-demand update propagation strategy furnishes better results.

4.2 Utilization of MVF's on the Clients

In the distributed environment of *ADMS±*, query requests from a client may be answered by MVFs that reside on other clients instead of retrieving from the server(s). This is very important for performance reasons but even more so for fault tolerance when the connections between some clients and the servers are unavailable.

We have extended the ADMS subsumption algorithm to discover *unions* of MVFs that subsume a client's query posed on the network as a *range* query [Pap94]. A query may be subsumed by the union of a number of MVFs residing on different clients even when none of the fragments does. Duplicates resulting from the union are filtered out by constructing additional range predicates.

Union subsumption can be used in several ways. First, we can improve performance of a query by replacing expensive join operators with simpler selections. Second, given that the MVFs may be located at client workstations we can reduce the contention for the server resources. Third, we can use union subsumption to parallelize query processing with fragments residing on different workstations. Note, even if we are not able to subsume the client query, we may be able to subsume some part of the data that are necessary for the computation of the client query; i.e., we may be able to subsume some nodes of the query graph by unions of MVFs.

Assuming that the MVFs are not indexed, the optimizer attempts to minimize the total size of the MVFs – equivalently, the total retrieval time – and also attempts to distribute the work evenly to all clients. However, the decision of an optimal set that subsumes Q is an NP-hard problem. Thus, we use a greedy polynomial best-fit optimization algorithm that selects at every step the "most promising" MVF. In addition, the selected set must not have more than a few hundred MVFs, because otherwise, the cost of applying the filters becomes greater than the cost of retrieving the MVF from the disk.

5 Conclusions

This article presented the motivation, concepts, ideas, and techniques of the ADMS Project. Many of the ideas pioneered in this project are finding their way into the commercial world. For example, Oracle is now offering limited incremental refresh of select only MVFs. Also, both Sybase and Ingres use incremental techniques for maintaining replicated data. Similarly, the recent flurry of research activity on views and their management is gratifying. We are content that our long-term conviction and persistence on view maintenance and replication have paid off, and that our ideas are finally receiving appropriate attention.

The ADMS design document was written in 1984, and implementation began on a SUN 3 workstation at that time. It has gone though several major revisions, such as when the ViewCache storage organization was evolving 1985, 1986, 1987, and when the SQL parser and cost-based optimizer were added in 1991. It has now migrated and been ported to SUN SPARC, DEC MIPS, HP SNAKE, and IBM POWER architectures running their various flavors of UNIX. The unified source tree consists of approximately 120,000 lines of "C" code.

The $ADMS\pm$ client-server architecture was designed during the fall of 1985 but the first implementation of the prototype began only in late 1987. $ADMS\pm$ is now in its third incarnation which includes new client-server and server-server join strategies, enhanced server catalogs (for selectivity estimation), and a robust TCP/IP based communication layer. Last year, we ran our first trans-antlantic joins between an Oracle database at the University of Maryland and an Ingres database at the National Technical University in Athens, Greece.

And the ADMS saga goes on. We are targeting our energy towards adaptive and intelligent techniques capable of learning from running queries against the database and fine tune their processing. We have developed a query optimizer for the ADMS– client. It has an adaptive cost estimator which exhibits excellent learning capability over foreign commercial DBMSs. An experiment is being conducted as of this writing and we will report the results in the near future.

References

[BLT86] J.A. Blakeley, P.A. Larson, and F.W. Tompa. Efficiently Updating Materialized Views. In *Proc. of the 1986 ACM SIGMOD Intern. Conference*, pages 61–71, August 1986.

[CD85] H. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Procs. of the 11th Intl. Conf. on VLDB*, pages 127–141, 1985.

[Chr83] S. Christodoulakis. Estimating Record Selectivities. *Inf. Syst.*, 8(2):105–115, 1983.

[CR93] C.M. Chen and N. Roussopoulos. Adaptive Database Buffer Allocation Using Query Feedback. In *Procs. of the 19th Intl. Conf. on Very Large Data Bases*, 1993.

[CR94a] C.M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[CR94b] C.M. Chen and N. Roussopoulos. The implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *Procs. of the 4th Intl. Conf. on Extending Database Technology*, 1994.

[DR92] A. Delis and N. Roussopoulos. Performance and Scalability of Client–Server Database Architectures. In *Proc. of the 19th Int. Conference on Very Large Databases*, Vancouver, BC, Canada, August 1992.

[DR93] A. Delis and N. Roussopoulos. Performance Comparison of Three Modern DBMS Architectures. *IEEE–Transactions on Software Engineering*, 19(2):120–138, February 1993.

[DR94] A. Delis and N. Roussopoulos. Management of Updates in the Enhanced Client–Server DBMS. In *Proccedings of the 14th IEEE Int. Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

[Eco89] N. Economou. Multisite Database Access in $ADMS\pm$. Master's thesis, University of Maryland, College Park, MD, 1989. Department of Computer Science.

[FCL92] M. Franklin, M. Carey, and M. Livny. Local Global Memory Management in Client–Server DBMS Architectures. In *Proc. of the 18th Int. Conference on Very Large Data Bases*, Vancouver, Canada, August 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.

[FNS91] C. Faloutsos, R. T. Ng, and T. Sellis. Predictive Load Control for Flexible Buffer Allocation. In *Procs. of the 17th Intl. Conf. on VLDB*, pages 265–274, 1991.

[HOT88] W. Hou, G. Ozsoyoglu, and B. K. Taneja. Statistical Estimators for Relational Algebra Expressions. In *Procs. of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 276–287, 1988.

[HS92] P. Haas and A. Swami. Sequential Sampling Procedures for Query Size Estimation. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 341–350, San Diego, CA, 1992.

[Ioa93] Y.E. Ioannidis. Universality of Serial Histograms. In *Procs. of the 19th Intl. Conf. on VLDB*, Dublin, Ireland, 1993.

[LN90] R. J. Lipton and J. F. Naughton. Practical Selectivity Estimation through Adaptive Sampling. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 1–11, Atlantic City, NJ, 1990.

[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.

[MD88] M. Muralikrishma and D. DeWitt. Equi-depth Histograms for Estimating Selectivity Factors for Multi-dimensional Queries. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 28–36, Chicago, Illinois, 1988.

[NFS91] R. T. Ng, C. Faloutsos, and T. Sellis. Flexible Buffer Allocation Based on Marginal Gains. In *Procs. of 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pages 387–396, 1991.

[Pap94] Y. Papakonstantinou. Computing a Query as a Union of Disjoint Horizontal Fragments. Technical report, Department of Computer Science, University of Maryland, College Park, MD, 1994. Working Paper.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 256–275, Boston, MA, 1984.

[RD91] N. Roussopoulos and A. Delis. Modern Client–Server DBMS Architectures. *ACM–SIGMOD Record*, 20(3):52–61, September 1991.

[RES93] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A Testbed for Incremental Access Methods. *IEEE Trans. on Knowledge and Data Engineering*, 5(5):762–774, 1993.

[RH80] D.J. Rosenkrantz and H.B. Hunt. Processing Conjunctive Predicates and Queries. In *Procs. of the 6th Intl Conf. on VLDB*, 1980.

[RK86a] N. Roussopoulos and H. Kang. Principles and Techniques in the Design of *ADMS±*. *Computer*, December 1986.

[RK86b] N. Roussopoulos and Y. Kang. Preliminary Design of ADMS±: A Workstation–Mainframe Integrated Architecture. In *Proc. of the 12th Int. Conference on Very Large Databases*, August 1986.

[RL85] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *Procs. of 1985 ACM SIGMOD Intl. Conf. on Management of Data*, Austin, 1985.

[Rou82] N. Roussopoulos. The Logical Access Path Schema of a Database. *IEEE Trans. on Software Engineering*, SE-8(6):563–573, 1982.

[Rou91] N. Roussopoulos. The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis. *ACM–Transactions on Database Systems*, 16(3):535–563, September 1991.

[S+89] X. Sun et al. Solving Implication Problems in Database Applications. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 185–192, 1989.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Data Base System. In *SIGMOD–Conference on the Management of Data*, pages 22–34. ACM, June 1979.

[SLRD93] W. Sun, Y. Ling, N. Rishe, and Y. Deng. An Instant and Accurate Size Estimation Method for Joins and Selection in a Retrieval-Intensive Environment. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 79–88, Washington, DC, 1993.

[TB88] F. Tompa and J. Blakeley. Maintaining Materialized Views Without Accessing Base Data. *Information Systems*, 13(4):393–406, 1988.