A Multi-Query Optimizer for Monet*

Stefan Manegold, Arjan Pellenkoft, and Martin Kersten

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{Stefan.Manegold,Arjan.Pellenkoft,Martin.Kersten}@cwi.nl

Abstract Database systems allow for concurrent use of several applications (and query interfaces). Each application generates an "optimal" plan—a sequence of low-level database operators—for accessing the database. The queries posed by users through the same application can be optimized together using traditional multi-query optimization techniques. However, the commonalities among queries of different applications are not exploited.
In this paper we present an efficient inter-application multi-query optimizer that re-uses previously computed (intermediate) results and eliminates redundant work. Experimental results on a single CPU system and a parallel system show that the inter-application multi-query optimizer improves the query evaluation performance significantly.

1 Introduction

Much effort has been spent on designing and implementing algorithms for database query optimization. Almost all current query optimizers are targeted at finding the best (or at least a good) execution plan for a single query at a time [SAC+79, IK90, GLPK94, VM96]. This is a reasonable approach for ad-hoc querying and traditional applications firing isolated, but rather complex queries at a time.

Modern database applications, such as data mining, however, strongly interact with the DBMS by sending a stream of query batches. This stream typically reflects a kind of navigation through the solution space of the data mining algorithms. Its batches consist of rather simple queries to be solved. Depending on the results of one step—consisting of a single query, or of a set of queries—an interactive user or an automated mining algorithm decides how to proceed. Typically, only a few parameters are changed in order to have a closer, i.e., more detailed look at a certain part of the database. Hence, it is very likely that subsequent data mining steps are similar and can easily benefit from re-using previously created intermediate results. This property can be used by applications to optimize access to a database at the cost of replicating parts of the query optimizer code within each application.

Further, data mining systems are typically multi-user systems, i.e., several users operate on the same database via the same or different mining applications.

* This work has been supported by the HPCN-CONQUER project.

Although the users act independently, formulating different queries, it is not unusual that the database system has to execute identical basic (but expensive) operations several times to satisfy the different requests. Unfortunately, this property cannot be exploited by a single product/application, as it is outside its scope of control.

Obviously, there are two sources of optimization that can be exploited if multiple queries are considered: re-use of previously calculated (and cached) intermediate results and elimination of redundant work. This calls for a new type of *inter-application* multi-query optimizer that is able to detect and exploit such optimization opportunities in a stream of individually optimized queries that originate from various applications.

The issue itself, *multi-query optimization* (MQO), has received limited attention in the database research community. As query optimization was shown to be NP-complete [IK84, SM97] it is not surprising that the problem of MQO is also NP-complete [SG90]. MQO can therefor only be achieved using heuristics [Jar85] or probabilistic techniques.

Early works [Fin82, Sel88] show that ad hoc queries can benefit from using materialized results generated by earlier queries, even if only equivalent expressions are considered. The savings can be considerable when compared to single query processing. Shim et al. [SSN94] propose improved heuristics to search for the global optimum in the state space that models all alternatives for evaluating a batch of queries. Chakravarthy and Minker [CM86] use a multi-query graph for representing multiple SPJ queries. Again, heuristics are used to identify common subexpressions and transform the graph into an evaluation strategy without applying a search algorithm. Chen and Dunham [CD98] improve these heuristics by, o.a., considering partial overlapping selection predicates.

All previous work on MQO considered a single application which allows for a unified query abstraction level to perform common subexpression elimination. However, a multi-query optimizer at the inter-application level, as we study in this paper, cannot take advantage of these techniques.

In [RC88, AR92], frameworks for analyzing the MQO problem are proposed. One of the issues addressed is that each type of (multi) query optimization should be done at the appropriate level of abstraction — e.g., one level for determining the appropriate join order and one to determine the join implementation. Furthermore, they point out that a multi-query optimizer should at least perform as well as a single query optimizer. Illustrative for this approach is the paper [KDB94], where the MQO is done at the algorithm-level to exploit the re-use of (temporary) hash tables.

In this paper we introduce a novel architecture to bring inter-application query-optimizer back into the mainstream of research. The prime innovation is to organize the query optimization problem into three tiers: *Strategic optimization, Tactical optimization* and *Operational optimization*. At each tier, different sources of optimization are exploited. The strategic tier uses the application (-model) knowledge, such as foreign-key dependencies, semantic integrity constraints, and user-application focus to derive a query execution plan.

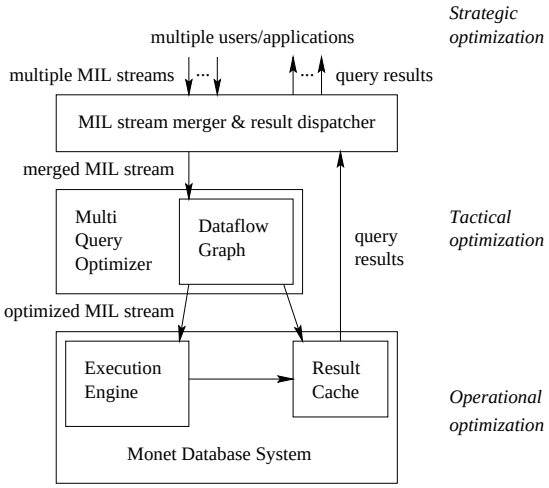

Figure 1. System Architecture

The tactical optimizer is geared at balancing the resources amongst competing queries. This involves both recognition of commonalities amongst (inter- and intra-application) requests and methods to exploit potential parallelism and replication at the database back-ends.

The operational optimizer decides at run time which is the most suitable algorithm for performing low-level database operation, e.g., it chooses between a hash join or a nested loops join. Also the re-use of dynamically created hash tables is done at this level.

In this paper, we present a multi-query optimizer for the middle tier (tactical optimization) that is focused on but not limited to query workloads generated by a specific, commercial data mining application. The optimizer keeps a history of calculated intermediate results to re-use them in subsequent queries and it detects common subexpressions of multiple queries to avoid redundant work.

Section 2 provides a short overview of the system and our focal data mining product. Section 3 illustrates by an extensive example the opportunities for inter- and intra-application in a data mining context. Section 4 reports on the results obtained using the DD Benchmark, a metric for judging the capabilities of a DBMS for interactive data mining. We conclude with a short outlook on the extensions planned for the tactical optimizer.

2 System Architecture

Our system architecture in depicted in Figure 1. The Monet database engine [BK95] is used as back-end server for multiple applications. The applications use the Monet Interpreter Language (MIL) [BK99], an algebraic query language,

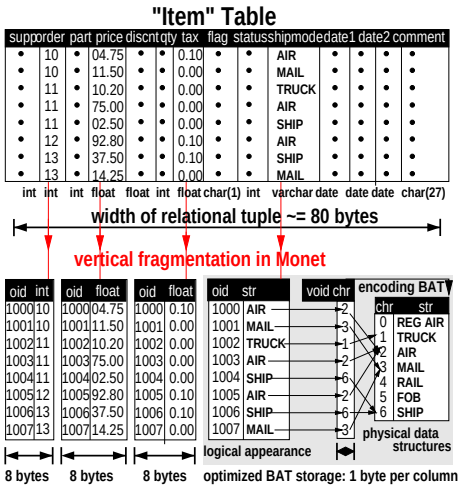

Figure 2. Vertically Decomposed Storage in BATs

to communicate with the back-end engine. Each application first optimizes its query individually on a logical level, e.g., algebraic operators are re-ordered to minimize the data volume to be handled. Then, the application translates each query into a sequence of MIL statements.

Our multi-query optimizer is placed between the applications and Monet. It takes the unified stream of MIL statements from the applications as input and produces an optimized stream of MIL statements to be sent to Monet for execution. We will describe the multi-query optimizer in detail in Section 3.

2.1 Monet

Monet is a main-memory database kernel developed at CWI. It is targeted at achieving high performance on query-intensive workloads, such as created by OLAP or data mining applications. Monet uses the Decomposed Storage Model (DSM) [CK85], storing each column of a relational table in a separate binary table, called a Binary Association Table (BAT). A BAT is represented in memory as an array of fixed-size two-field records [OID,value], or Binary UNits (BUN). The OIDs in the left column are unique per original relational tuple, i.e., they link all BUNs that make up a original relational tuple (cf. Figure 2). The major advantage of the DSM is that it minimizes I/O and memory access costs for column-wise data access which occurs frequently in OLAP and data mining workloads [BRK98, BMK99].

2.2 Data Surveyor

Most existing data mining tools employ specialized data structures and algorithms to manipulate mass data outside the DBMS. Data Surveyor of Data

Distilleries, however, integrates data mining with a DBMS using a 3-tier architecture:

GUIs taking the form of Java applets. Apart from a powerful expert data mining interface, Data Surveyor provides pre-cooked user interfaces tailored to special end-user requirements.

Data Mining Kernel containing tens of data-mining specific algorithms. This component directs the data mining operations and translates a data mining task into multiple DBMS queries.

DBMS back-end can be all SQL-speaking commercial (parallel) DBMSs. Further, Monet can be used as high-performance back-end.

To facilitate the translation of a data mining task into DBMS queries, the Data Mining Kernel uses a unique algorithmic framework that decomposes data mining algorithms in three orthogonal dimensions:

- a *modeling language* for expressing hypotheses,
- a *quality function* for testing the quality of a hypotheses, and
- a *search strategy* for looking for interesting hypotheses.

2.3 DD Benchmark

The Drill Down Benchmark (short: DD Benchmark) is designed to measure DBMS performance on a typical data mining query load. The benchmark is formulated as a typical data mining task, which in turn is translated into DBMS queries. The mining task mimics a customer loyalty application, a common and prototypical data mining problem. In this task, a company wants to find profiles for (un)reliable groups of customers.

The DD Benchmark uses decision rules as the modeling language to describe such customers, where the rules are simple conjunctions of selections on the attributes of the mining table. The quality of such rules is expressed as a confidence interval and a beam-search algorithm is used to find interesting hypotheses.

The mining table contains 1 million customer records, consisting of 100 attributes. Six attributes play a role in the mining task. A detailed description of the DD Benchmark is available in [BRK98].

3 Multi-Query Optimizer

In this Section, we present an overview of the multi-query optimization facilities embodied by our prototype optimizer.

3.1 Concept

Our optimizer mainly focuses on the following optimization potentials:

elimination of common (sub-)expressions Especially in a data mining scenario, it is very likely that several queries to the same database will shared at least some subexpressions. Evaluating identical subexpressions several times (once per query) is of course redundant work. Hence, our optimizer identifies such common subexpressions, schedules each subexpression only once for evaluation and ensures that all queries can use the respective intermediate result without any additional costs once it has been generated.

re-use of cached intermediate results Common subexpression may not only occur among queries that are optimized at the same time. Rather, a query might also require an intermediate result that has already once been calculated for an earlier query. Hence, we keep intermediate results materialized in main memory for later re-use.

parallelization Monet and MIL offer the possibility of parallel query execution. On a shared-memory multi-processor machine, for instance, a multi-threaded Monet engine can evaluate multiple independent MIL statements concurrently. Our optimizer takes care of that by identifying independent statements and scheduling them for concurrent execution.

3.2 Example

As a simple example, consider a relation "customer" with four attributes ("gender","age","marital","reliable") and the following four SQL queries taken from the DD Benchmark. Besides selections, the queries contain groupings and aggregations, the most frequent tasks in data mining.

Q1:
```
SELECT age, reliable, count(*)
FROM customer
WHERE gender = 'f'
GROUP BY age, reliable;
```

Q2:
```
SELECT marital, reliable, count(*)
FROM customer
WHERE gender = 'f'
GROUP BY marital, reliable;
```

Q3:
```
SELECT age, reliable, count(*)
FROM customer
WHERE gender = 'm'
GROUP BY age, reliable;
```

Q4:
```
SELECT marital, reliable, count(*)
FROM customer
WHERE gender = 'm'
GROUP BY marital, reliable;
```

In Monet, the relation is stored in five BATs: "C_gender", "C_age", "C_marital", and "C_reliable". The SQL queries translate to the following four MIL programs. The "Vij" are variables that store the materialized intermediate results.

```
P1: V11 := CTgroup(C_age);
    V12 := select(C_gender,'f');
    V13 := semijoin(C_reliable,V12);
    V14 := CTgroup(V11,V13);
    V15 := histogram(V14);
    print(C_age,C_reliable,V15);
P2: V21 := CTgroup(C_marital);
    V22 := select(C_gender,'f');
    V23 := semijoin(C_reliable,V22);
    V24 := CTgroup(V21,V23);
    V25 := histogram(V24);
    print(C_marital,C_reliable,V25);
P3: V31 := CTgroup(C_age);
    V32 := select(C_gender,'m');
    V33 := semijoin(C_reliable,V32);
    V34 := CTgroup(V31,V33);
    V35 := histogram(V34);
    print(C_age,C_reliable,V35);
P4: V41 := CTgroup(C_marital);
    V42 := select(C_gender,'m');
    V43 := semijoin(C_reliable,V42);
    V44 := CTgroup(V41,V43);
    V45 := histogram(V44);
    print(C_marital,C_reliable,V45);
```

In MIL, groupings are materialized in a *cross-table* BAT that holds in the head column identifiers of all objects of interest, and in the tail a unique group identifier. The "CTgroup" operators construct such cross-tables. The unary "CTgroup" is executed on an [OID,value] BAT. It returns an [OID,OID] BAT with the same head column as the input and a group-id in the tail column for each BUN. Each group-id is chosen from the collection of OIDs from the head of its group members. The binary "CTgroup" refines a cross-table by subdividing the groups according to an additional [OID,value] BAT.

The "histogram" operation creates a histogram of the tail values of a BAT. It returns a BAT with each distinct tail value of the input in its head column and the number of occurances of that value in its tail column. Applied on a cross-table, the histogram calculates the group sizes.

The "print" operation finally performs a multi-BAT equi-join on the head columns, printing a multi-column table consisting of the respective tail columns. In our example, "print" creates the required query result, a table that consists of the grouping attributes and the group sizes.

The operations "CTgroup(C_age)", "CTgroup(C_marital)", "select(C_gender,'f')", and "select(C_gender,'m')" occur twice creating pairwise identical results V11≡V31, V21≡V41, V12≡V22, and V32≡V42. Hence, "semijoin(C_reliable,V12)" and "semijoin(C_reliable,V22)" are identical as well

as "semijoin(C_reliable,V32)" and "semijoin(C_reliable,V42)". The multi-query optimizer has to detect these commonalities and avoid redundant work.

3.3 Implementation

The optimizer takes a stream of MIL statements as input. This stream is the merged output of several applications (or multiple mining threads) and contains a set of queries, each optimized in isolation. The optimizer stores the queries in a dependency graph. Each distinct MIL statement makes up a node of the graph. The nodes are connected by directed edges representing the data dependencies between the nodes (i.e., the MIL statements). Hence, the dependency graph forms a directed acyclic graph (DAG).

Elimination of Common Subexpressions At database startup time, the dependency graph consists of a set of non-connected nodes. Each of these nodes represents a persistent BAT stored in the database. When receiving input, the optimizer adds a new node for each distinct MIL statement. The node then represents the intermediate result created by that very MIL statement. Additionally, the optimizer adds edges to the dependency graph, representing the dependency of an intermediate results on the parameters (i.e., persistent BATs and previous intermediate results) of the respective MIL statement. To eliminate common subexpressions, the optimizer identifies identical MIL statements by their signature (i.e., operator name and parameters) and maps them to the same node in the dependency graph. Hence, each distinct intermediate result occurs only once in the dependency graph.

Figure 3 shows the dependency graph for our simple example. The equivalent operations are identified and mapped to a single node.

Parallelization When the execution engine becomes idle, the optimizer scans the dependency graph for independent statements to be executed next. Independent statements are nodes that depend only on persistent BATs or on intermediate results already calculated. In other words, independent statements are ready to be evaluated immediately. For each independent node, the optimizer checks whether there is a linear path starting at the independent node, whose nodes successively become independent as soon as their very predecessor in the path is executed. If such a path exists, all nodes of that path (including the original independent node) are gathered into a single task. Otherwise, the task consists only of the original node. All statements within a task are evaluated sequentially according to their dependencies. Gathering linear paths into sequential tasks ensures that intermediate results are used as soon as possible and preferably by the same thread/CPU that created them. Parallelism is exploited by sending several/all independent tasks to the execution engine to be evaluated concurrently, each by a separate thread.

The grey-shadings in Figure 3 depict sequential tasks and parallel blocks. The optimized MIL program is given in Figure 4. Operations within a sequential task

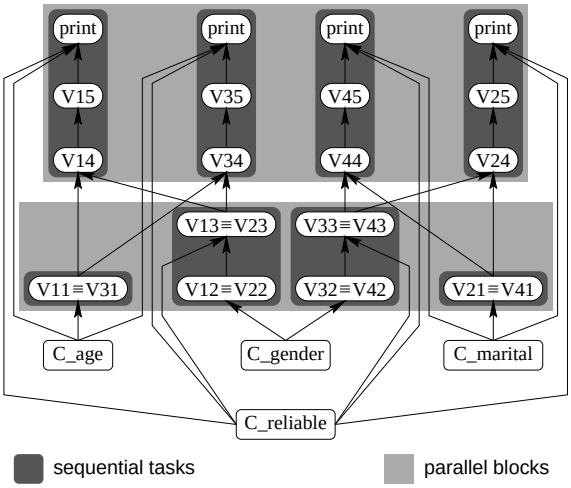

Figure 3. Sample Dependency Graph

("{...}") are are executed one after another. All operations or tasks within a parallel block ("{|...|}") are executed concurrently.

Re-use of Cached Intermediate Results The execution engine keeps all intermediate results materialized in main memory. Hence, they are instantly available for later re-use.[1]

By annotating the nodes in the dependency graph appropriately, the optimizer keeps track of which intermediate results are already available and which statements still need to be executed. Thus, the optimizer can easily detect, when a new statement requests an intermediate result that has already been calculated earlier.

4 Experiments

To analyze the benefits of multi-query optimization in a data mining scenario, we run experiments using the DD Benchmark [BRK98]. The DD Benchmark creates a typical Data Mining workload. It consists of 5 batches of queries, 133 queries altogether. All queries perform selections, groupings, and aggregations on a subset of the attributes of a single relational table. The query batches mimic the behavior of a beam-search algorithm to generate decision trees.

To run the DD Benchmark against the Monet database, we use the MIL programs as generated by Data Distilleries' mining tool Data Surveyor. In this form, the whole DD Benchmark consists of some 1200 MIL statements altogether.

[1] Currently, we implicitly assume an unlimited memory capacity. Cache management facilities are to be added in the near future (cf. Section 5).

```
{|
    {
        V11 := CTgroup(C_age);
    }{
        V12 := select(C_gender,'f');
        V13 := semijoin(C_reliable,V12);
    }{
        V32 := select(C_gender,'m');
        V33 := semijoin(C_reliable,V32);
    }{
        V21 := CTgroup(C_marital);
    }
|}{|
    {
        V14 := CTgroup(V11,V13);
        V15 := histogram(V14);
        print(C_age,C_reliable,V15);
    }{
        V34 := CTgroup(V11,V33);
        V35 := histogram(V34);
        print(C_age,C_reliable,V35);
    }{
        V24 := CTgroup(V21,V13);
        V25 := histogram(V24);
        print(C_marital,C_reliable,V25);
    }{
        V44 := CTgroup(V21,V33);
        V45 := histogram(V44);
        print(C_marital,C_reliable,V45);
    }
|}
```

Figure 4. Optimized MIL Program

Table 1 compares the performance of executing the non-optimized and the optimized MIL program running a single-threaded Monet server on an Intel PentiumII 400 MHz based PC with 512 MB main memory.

The results show, that our optimizer is able to detect overlap among the queries and eliminate common subexpressions (i.e., redundant work) efficiently. The total optimization overhead is approximately 400 milliseconds, i.e., negligible. The improvements increase with each additional batch of queries, as the execution can then benefit from re-using previously calculated intermediate results. In batch 4, the number of instructions that is actually executed is reduced by factor 5.1, the elapsed time it even reduced by factor 17.3. The overall improvement for the whole benchmark is factor 3.7.

Table 1. Experimental Results: sequential Monet Server on PC

batch	non-optimized		optimized		improvement (factor)	
	# stat's	time [ms]	# stat's	time [ms]	# stat's	time
0	14	1,940	14	1,940	1.0	1.0
1	18	4,864	12	3,339	1.5	1.5
2	345	43,350	135	17,444	2.6	2.5
3	444	38,621	114	7,059	3.9	5.5
4	447	27,802	87	1,608	5.1	17.3
0-4	1,268	116,578	362	31,898	3.5	3.7

Table 2. Experimental Results: sequential Monet Server on Origin2000

batch	non-optimized		optimized		improvement (factor)	
	# stat's	time [ms]	# stat's	time [ms]	# stat's	time
0	14	1,664	14	1,664	1.0	1.0
1	18	3,926	12	2,615	1.5	1.5
2	345	39,811	135	15,574	2.6	2.6
3	444	36,429	114	6,382	3.9	5.7
4	447	26,889	87	1,422	5.1	18.9
0-4	1,268	108,720	362	27,795	3.5	3.9

We ran the same experiments on an SGI Origin2000 with 24 MIPS R12000 CPUs (300 MHz) and 48 GB of main memory. Table 2 shows the results using a single-threaded Monet server. The improvements are similar to those on the PC.

Table 3 depicts the results using a multi-threaded Monet server, i.e., using parallel execution. In the non-optimized version, each query forms a sequential task. All queries within one batch are independent and can be evaluated concurrently. With increasing degree of parallelism, the improvement of the optimized version over the non-optimized version slightly decreases. The reason being that most improvement is gained from the fact that several statements can re-use an intermediate result that is created only once. The Origin2000 is a ccNUMA machine with distributed shared memory. Thus, memory access costs differ significantly between local and remote memory access. With higher parallelism, it becomes more likely, that a thread re-uses a result that has been created by another thread, and is hence stored on another (remote) CPU board. But even with 9 threads, the improvement is still factor 2.5.

Finally, Figures 5 and 6 show the speedup curves for the non-optimized and the optimized version, respectively. Both figures show 6 speedup curves, 5 of them representing the individual performance of each batch and the last one representing the overall performance.

In both cases, batch 0 and batch 1 show rather limited speedup. This is due to the fact that the two batches contain only 7, respectively 6, queries. The other batches consist of more that 30 queries each, i.e., there is sufficient

Table 3. Experimental Results: parallel Monet Server on Origin2000

threads	non-optimized time [ms]	optimized time [ms]	improvement (factor)
1	108,720	27,795	3.9
2	54,944	15,598	3.5
3	39,089	12,221	3.2
4	30,288	9,451	3.2
5	25,874	8,893	2.9
6	21,934	7,549	2.9
7	19,793	7,670	2.6
8	18,587	7,072	2.6
9	18,808	7,436	2.5

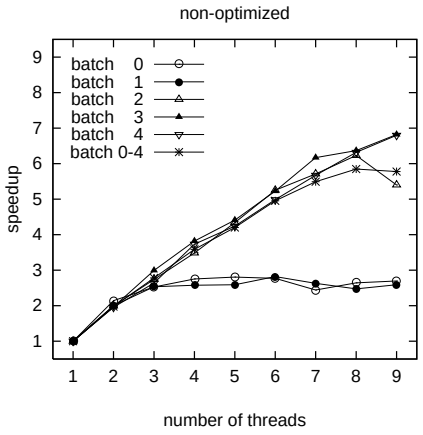

Figure 5. Speedup: non-optimized

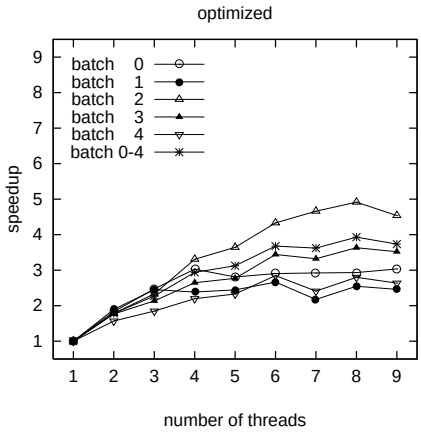

Figure 6. Speedup: optimized

potential for parallelism. Doing redundant work on local data in parallel, the non-optimized version achieves near-linear speedup for the remaining batches and for the overall performance. The optimized version, however, avoids redundant work. Hence, several threads might re-use the same intermediate result, causing remote memory access (see above). Although there is sufficient potential for parallelism, the extra costs for remote memory access limit the achievable speedup on a ccNUMA machine.

5 Conclusion and future research

In this paper, we showed that data mining applications provide some inter-query optimization potentials that traditional query optimizers cannot exploit. We proposed and implemented an application-independent inter-application multi-query optimizer. The optimizer avoids redundant work by eliminating common

subexpressions and re-using cached intermediate results. Performance experiments with the Drill Down Benchmark showed, that the optimizer yields significant improvements of up to factor 4, while causing hardly any optimization overhead.

The architecture provides an outlook on the novel three-tier optimization scheme under development for the Monet DBMS. The underlying hypothesis is that by breaking the optimizer into distinct tiers, we can both simplify the optimization process at each level and further benefit from the inter- and intra-application dependencies. This approach has already been shown beneficial at the operational level in [BK95, BK99] and at the tactical level in this paper.

The research agenda for the tactical optimizer includes the following near-term extensions:

cache management Main memory capacity—although constantly growing—is not unlimited. Hence, the result cache will exceed (real) main memory capacity, eventually. To prevent this, cache management is necessary to decide when and which (old) results to discard in order to create space for new results. We plan to investigate several strategies, mainly focusing on taking into account the costs for (re-) creating the results and the benefits of keeping certain results.

re-using supersets Currently, the optimizer is limited to identify and re-use equivalent intermediate results, only. Additionally, it is also beneficial to re-use the smallest superset of a requested intermediate result in case the equivalent result is not available, provided that using the superset is cheaper that using the original persistent BAT.

pattern rewriting Finally, we will extend the optimizer with 'peephole optimization rules' to detect certain patterns in the MIL sequence, respectively in the dependency graph, in order to replace them with less expensive ones. This approach releases the applications from taking care of any tactical optimization. Instead, applications only need to generate straightforward MIL code. The tactical optimizer then takes care of optimizing the code before actually executing it. In particular, when the Monet engine is extended by a new operator that implements a sequence of operators more efficiently, it will enable arbitrary applications to use the new implementation, without changing the applications; the application even doesn't have to know about the existence of the new operator.

In general, many of the decisions that have to be taken by these extensions at run time cannot be formulated as static rules or heuristics. Hence, we will provide the optimizer with the required cost information to support its decisions.

Acknowledgements. We would like to thank Florian Waas for his contributions to a prototype of the multi-query optimizer.

References

[AR92] J. R. Alsabbagh and V. V. Raghavan. A Framework for Multiple-Query Optimization. In *Proc. Research Issues on Data Eng.: Transaction and Query Processing*, Tempe, AZ, USA, February 1992.

[BK95] P. Boncz and M. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Proc. Basque International Workshop on Information Technology*, San Sebastian, Spain, July 1995.

[BK99] P. Boncz and M. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, 8(2), October 1999.

[BMK99] P. Boncz, S. Manegold, and M. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 54–65, Edinburgh, Scotland, UK, September 1999.

[BRK98] P. Boncz, T. Rühl, and F. Kwakkel. The Drill Down Benchmark. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 628–632, New York, NY, USA, June 1998.

[CD98] F.-C. F. Chen and M. H. Dunham. Common Subexpression Processing in Multiple Query Processing. *IEEE Trans. on Knowledge and Data Eng.*, 10(3):493–499, May/June 1998.

[CK85] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 268–279, Austin, TX, USA, May 1985.

[CM86] U. S. Chakravarthy and J. Minker. Multiple Query Processing in Deductive Databases using Query Graphs. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 384–390, Kyoto, Japan, August 1986.

[Fin82] S. J. Finkelstein. Common Expression Analysis in Database Applications. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 235–245, Orlando, FL, USA, June 1982.

[GLPK94] C. A. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fast, Randomized Join-Order Selection – Why Use Transformations? In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 85–95, Santiago, Chile, September 1994.

[IK84] T. Ibaraki and T. Kameda. Optimal Nesting for Computation N-Relational Joins. *ACM Trans. on Database Systems*, 9(3), September 1984.

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 312–321, Atlantic City, NJ, USA, May 1990.

[Jar85] M. Jarke. Common Subexpression Isolation in Multiple Query Optimization. In W. Kim, D. S. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*, pages 191–205. Springer-Verlag, 1985.

[KDB94] M. H. Kang, H. G. Dietz, and B. Bhargava. Multiple-query optimization at algorithm-level. *Data and Knowledge Engineering*, 14(1), November 1994.

[RC88] A. Rosenthal and S. Chakravarthy. Anatomy of a Modular Multiple Query Optimizer. *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 230–239, 1988.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 23–34, Boston, MA, USA, May 1979.

[Sel88] T. K. Sellis. Multiple-Query Optimization. *ACM Trans. on Database Systems*, 13(1), March 1988.

[SG90] T. Sellis and S. Ghosh. On the Multiple-Query Optimization Problem. *IEEE Trans. on Knowledge and Data Eng.*, 2(2):262–266, Jun 1990.

[SM97] W. Scheufele and G. Moerkotte. On the Complexity of Generating Optimal Plans with Cross Products. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 238–248, Tucson, AZ, USA, May 1997.

[SSN94] K. Shim, T. Sellis, and D. Nau. Improvements on a Heuristic Algorithm for Multiple-Query Optimization. *Data and Knowledge Engineering*, 12(2):197–222, March 1994.

[VM96] B. Vance and D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 35–46, Montreal, Canada, June 1996.