An Architecture for Recycling Intermediates in a Column-store

Milena G. Ivanova Martin L. Kersten Niels J. Nes Romulo A.P. Gonçalves
{milena,mk,niels,goncalve}@cwi.nl
Centrum Wiskunde en Informatica (CWI)
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands

ABSTRACT

Automatically recycling (intermediate) results is a grand challenge for state-of-the-art databases to improve both query response time and throughput. Tuples are loaded and streamed through a tuple-at-a-time processing pipeline avoiding materialization of intermediates as much as possible. This limits the opportunities for reuse of overlapping computations to DBA-defined materialized views and function/result cache tuning.
In contrast, the operator-at-a-time execution paradigm produces fully materialized results in each step of the query plan. To avoid resource contention, these intermediates are evicted as soon as possible.
In this paper we study an architecture that harvests the by-products of the operator-at-a-time paradigm in a column store system using a lightweight mechanism, the *recycler*. The key challenge then becomes selection of the policies to admit intermediates to the resource pool, their retention period, and the eviction strategy when facing resource limitations.
The proposed recycling architecture has been implemented in an open-source system. An experimental analysis against the TPC-H ad-hoc decision support benchmark and a complex, real-world application (SkyServer) demonstrates its effectiveness in terms of self-organizing behavior and its significant performance gains. The results indicate the potentials of recycling intermediates and charters a route for further development of database kernels.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Design, Performance, Management

Keywords

Caching, Database kernels, Column-stores

1. INTRODUCTION

Query optimization and processing in off-the-shelf database systems is often still focused on individual queries. Queries are optimized in isolation using statistics gathered, analytical models, and heuristic rewrite rules, and ran against a kernel regardless opportunities offered by concurrent or previous invocations.

This approach is far from optimal and two directions to improve upon this situation are being actively explored: materialized views and reuse of (partial) results. Both depend and interact heavily with the underlying architecture, its execution paradigm and opportunities for optimizers to exploit transient information.

The state-of-the-art commercial systems use a tuple-at-a-time pipelined execution model which avoids the overhead of materializing intermediates [9]. However, this paradigm also limits the opportunities for shared and/or reused computations. It requires detection of overlapping query expression trees and temporal alignment of their data flows. One way to deal with this architectural limitation is to use materialized views or function/query result set caches [Oracle, DB2, SQLServer]. Materialized views are derived from query logs and they have been researched extensively in recent years [16, 8, 25, 24]. They represent common sub-queries, whose materialization improves subsequent processing times. The view management component of an optimizer takes them into account while exploring the space of alternative execution plans. Typically, a database administrator supported by workload analysers determines which portions to materialize [1, 4]. Reuse of partial results is also useful in applications with parametrised queries [25, 14, 18].

The operator-at-a-time execution paradigm, where complete intermediates are a byproduct of every step in the query execution plan, calls for an in-depth analysis of its reuse potentials. We believe that this (off-beat) approach, in terms of resource requirements during query execution, can be exploited to speed up query streams significantly in a self-organizing way. In other words, it pays off to recycle intermediate results of relational algebra operations instead of blindly garbage collecting them or avoiding them altogether.

Recycling intermediate results improves response time and throughput when their creation cost and management cost can be kept under control. In the operator-at-a-time setting, only the second cost factor is relevant, because the creation cost is always taken by the execution paradigm.

Recycling is a refinement of operator caching, a technique known for a long time. However, the inter-dependencies between the relational operators in a query plan allow for a variety of policies to capitalize on the algebra semantics. In contrast to materialized views, a resource pool of recycled intermediate results adapts continuously to the workload without DBA intervention and incurs minimal start-up and maintenance costs.

This hypothesis is tested in the context of the operator-at-a-time database system MonetDB [17]. Its architecture differs in a fundamental way from state-of-the-art (commercial) systems. In addition to a different execution paradigm, it is based on a canonical implementation of a column store. This means that recycling can be focused on horizontal fragments of base columns or their derivations. This greatly simplifies fragmentation management and predicate subsumption analysis to find matching operations.

The MonetDB system has been shown to be highly efficient in memory resident settings [2]. Of course, in a pure main-memory setting, the resource management has to discard intermediates as soon as possible, which is handled by a specific optimizer module that injects garbage collection commands into the execution plans to drop temporary tables as soon as they are not needed for the remainder.

The realization of our idea requires a modification in the MonetDB query execution engine. Therefore, its abstract machine interpreter is hooked up with a *recycler* optimizer and runtime module. The optimizer marks operations of interest for harvesting. The runtime support uses this advice to manage a pool of partial results. It avoids re-computation of common sub-queries by extracting readily available results from the pool.

The key issue in the design of the recycler is to identify efficient and effective policies to use and maintain the resource pool. It encompasses decisions in three dimensions: instruction matching, investment cost versus savings, and pool administration maintenance. For each instruction to be executed the recycler performs a matching process, i.e., it searches for a possible reusable relational algebra operation in the recycle pool. For each operation executed the recycler decides if it is beneficial to keep the result. Finally, to prevent the pool of intermediates becoming a resource bottleneck itself, operations with low potential for reuse should be cleaned from the pool to reduce the memory usage and the search time.

Cleaning of low beneficial intermediates to accommodate new instructions gradually adapts the content of the recycle pool to workload changes. We propose and evaluate several cache policies selecting instructions for eviction. These include traditional approaches, such as LRU, and cost-based policies based on plan semantics. A distinguishing characteristic of all policies is that they respect and exploit the semantic relationships amongst the operations executed.

We consider the recycler architecture especially suitable for applications with prevailing read-only workload and relatively expensive processing, such as data analytics and decision support. Low data volatility means that invalidation of intermediates is not needed too often. Expensive processing due to computational complexity and/or large data volumes creates weighty intermediates, that are worth keeping and beneficial for reuse.

The recycler is evaluated extensively in two experimental settings. First, we conducted experiments with the SkyServer application[20]; a sizable and complex scientific database application, whose 200 page sized SQL schema includes views, procedure abstractions, and a well chosen set of indices. The experiments with a 100 GB database and samples of the workload observed show that a tenfold improvement is achieved by our approach by keeping only partial replicas over persistent tables. This is remarkable, because the database design of the SkyServer already underwent a significant DBA design exercise[21].

Next, to gain more insight in the benefits, we focused on the TPC-H decision support benchmark [23]. Despite the fact that this benchmark consists of rather orthogonal queries, it illustrates the internal mechanisms of the recycler and its performance efficiency in a more controlled manner. The benchmark is used to analyze the baseline performance and the impact of different design choices.

The results obtained in the context of MonetDB are, in principle, applicable in a tuple-at-a-time execution paradigm [9]. It calls for selection of operators in the execution plan that mirror the results to the resource pool as well as the next operator in the plan. Judicious use of the technique may complement the prevalent technique based on workload analysers and (partially) materialized views. However, experimental proofs of this hypothesis should come from their code owners.

The remainder of the paper is organized as follows. Section 2 provides a short overview to the MonetDB architecture and its abstract relational algebra engine. Section 3 describes the recycler architecture and discusses the policies for management of the resource pool. The experimental evaluations are presented in section 4 and section 5. Section 6 describes the related work and Section 7 summarises our findings.

2. BACKGROUND

In this section we give a summary of the MonetDB architecture[1] focusing on the processing model and illustrated by an SQL:2003 example.

2.1 Architecture

MonetDB is a modern fully functional column-store database system, designed in the late 90's with a proven track record in various fields [2, 26, 7]. To make this paper self-contained we re-iterate the system's basic building blocks, its architecture, and its execution model.

MonetDB stores data column-wise in binary structures called Binary Association Tables, or BATs, which represent a mapping from an OID to a base type value. The storage structure is equivalent to large, memory-mapped dense arrays. It is complemented with hash-structures for fast key look-up. Additional properties are used to steer selection of more efficient implementation, e.g., sorted columns lead to sort-merge join operations.

The software stack of MonetDB consists of three layers. The bottom layer is formed by a library that implements a binary-column storage engine, including a rich set of highly optimized relational operators. This engine is programmed using the MonetDB Assembly Language(MAL), which provides a convenient abstraction over the kernel libraries, and a concise programming model for plan generation and ex-

[1]The system can be downloaded from http://monetdb.cwi.nl

```
function user.s1_2(A0:date,A1:date,A2:int,A3:str):void;
  X5 := sql.bind("sys","lineitem","l_returnflag",0);
  X11 := algebra.uselect(X5,A3);
  X14 := algebra.markT(X11,0@0);
  X15 := bat.reverse(X14);
  X16 := sql.bindIdxbat("sys","lineitem","l_orderkey_fkey");
  X18 := algebra.join(X15,X16);
  X19 := sql.bind("sys","orders","o_orderdate",0);
  X25 := mtime.addmonths(A1,A2);
  X26 := algebra.select(X19,A0,X25,true,false);
  X30 := algebra.markT(X26,0@0);
  X31 := bat.reverse(X30);
  X32 := sql.bind("sys","orders","o_orderkey",0);
  X34 := bat.mirror(X32);
  X35 := algebra.join(X31,X34);
  X36 := bat.reverse(X35);
  X37 := algebra.join(X18,X36);
  X38 := bat.reverse(X37);
  X40 := algebra.markT(X38,0@0);
  X41 := bat.reverse(X40);
  X45 := algebra.join(X31,X32);
  X46 := algebra.join(X41,X45);
  X49 := algebra.selectNotNil(X46);
  X50 := bat.reverse(X49);
  X51 := algebra.kunique(X50);
  X52 := bat.reverse(X51);
  X53 := aggr.count(X52);
  sql.exportValue(1,"sys.orders","L1","wrd",32,0,6,X53);
end s1_2;
```

Table 1: MAL plan of example query

ecution. Powerful debugging tools create an environment where debugging database optimizers has become feasible.

The next layer is formed by a series of targeted query optimizers. They perform a program transformation, i.e., take a MAL program and transform it into an improved one. Two dozen optimizer modules are included in the distribution, ranging from a simple constant expression evaluator to a complex dynamic plan choice generator, such as a runtime driven memo-plan query optimizer.

The top layer consists of front-end compilers (SQL, XQuery), that translate high-level queries into MAL plans. The compilers also include optimizers to exploit language semantics and heuristic rewrite rules that do not depend on physical properties or algorithmic cost. MonetDB is an easy, accessible toolkit for embarking upon database kernel innovations as studied in this paper.

2.2 Query Processing

In this work we use the SQL front-end. All SQL queries are translated into a parametrized representation, called a query template, by factoring out all literal constants. This means that a query execution plan in MonetDB is not optimal in terms of a cost model, because range selectivities do not have a strong influence on the plan structure. Plans do, however, exploit both well-known heuristic rewrite rules, e.g., selection push-down, and foreign key properties, i.e., join indices. The query templates are kept in a query cache. Table 1 illustrates the MAL query template produced for an example query over the TPC-H database[2]:

```
select count(distinct o_orderkey)
from orders, lineitem
where l_orderkey = o_orderkey
   and o_orderdate >= date '1996-07-01'
   and o_orderdate < date '1996-07-01'
                   + interval '3' month
   and l_returnflag = 'R';
```

This query has been translated by the SQL compiler into a MAL function under the assumption that the table is accessed in read only mode. For the general case, where concurrent transactions may register updates to the underlying tables, the MAL plan grows to several hundreds of instructions. The function body is a linear representation of the query plan. It may appear complex at first sight, but this is a consequence of the canonical representation of the binary relational algebra being supported.

Using the bind operation the query plan localizes in the SQL catalogue the persistent BATs for ORDERS and LINEITEM tables. The major part of the plan consists of binary relational algebra instructions. We can distinguish several threads of execution, each starting with binding a persistent column, reducing it using a filter expression or joining it with another column, until the tuples are transformed into result attributes. The last instruction constructs the query result table.

The query template is processed by a chain of optimizers before taking it into execution. The default optimizers range from simple constant expression evaluation, preparation for multi-core parallel processing, and garbage collection to reduce the memory footprint. The design of MAL simplifies instruction pattern analysis and exploitation of data flow relations.

The MAL program is interpreted in a linear fashion. The overhead of the interpreter is kept low, well below the one usec per instruction. The default interpreter is fully equipped with runtime debugging and performance monitoring. However, if performance measurements are not needed, a fast-path interpreter is called.

2.3 Materialization

A discriminating factor of MonetDB is its reliance on full materialization of all intermediate results. That is, every relational operator takes one or more columns and produces a new set of columns. All but a few of the kernel libraries exhibit this functional behavior. For instance, in the example in Table 1 the result of the selection operation over the L_RETURNFLAG attribute is materialized in a BAT assigned to the variable X11.

Don't be misled to overestimate the resource cost related to materialization, because the MonetDB kernel extensively uses structure sharing to minimize the need of making complete copies. Many instructions are primarily aimed at administration of the properties or viewpoints. For example, the *mirror()* operator switches the two columns in a BAT and is a zero cost operation without data copying. Likewise, *markT()* and *reverse()* are zero cost operations that only materialize a new viewpoint over the underlying data structures. Even a range select operations may become a cheap operation when the underlying BAT happens to be ordered. Then a view is returned, which only keeps a reference to the underlying BAT and the range of qualifying tuples.

The materialization of intermediates is usually considered an overhead that is avoided in the pipelined execution paradigm. However, full materialization benefits from

[2]Details on the MAL plans and optimizers can be found on http://monetdb.cwi.nl

fast, cache-conscious algorithms and the price of RAM. It also makes the intermediates readily available for reuse by queries with overlapping expressions, as we will show in the remainder of this paper.

3. RECYCLER ARCHITECTURE

In this section we describe in detail the recycler optimizer and runtime module for the MonetDB system. The recycler is designed with several boundary conditions in mind. First, and foremost, it is targeted to SQL queries over a predominantly read-only database. The query plans are produced in isolation, i.e., without knowledge of the workload itself, using common relational query optimizer techniques. Intermediate results from individual MAL operations laying around from previous queries are not taken into account at optimization time. Instead, matching of instructions, eventually followed by a decision to reuse an intermediate, is performed at *run time*. This just-in-time approach appears to be more flexible. The optimized query templates are in this way independent of the intermediates currently available and are readily reusable.

3.1 Designating Instructions for Recycling

The first design issue is to identify instructions of interest to the recycler. This is performed during query optimization by the *recycler optimizer*. It inspects the MAL plan and marks instructions and variables eligible for control by the recycler. An instruction becomes subject to recycler monitoring if all arguments are either constants or variables that are already designated as recycling candidates.

Many MAL instructions are of no interest to the recycler. For example, cheap operations, such as simple arithmetic expressions, should not be recycled. The overhead of their administration outweighs the expected gain. In addition, a symbolic expression evaluator already removes side-effect free expressions involving constant scalar arguments.

All instructions with side effects are to be handled with caution. Updates are not candidates for recycling, but they affect the content of the recycle pool. Every time a BAT is updated, any copy, or derivation of it retained in the pool, should be invalidated. Therefore, the update volatility puts a boundary on the effectiveness of recycling.

Since processing of an SQL query starts with binding variables to persistent columns using catalogue names, the net effect is that the recycler optimizer marks operator threads starting with access to these columns and propagates the property through the query plan as far as possible. Typically, the threads involve selections, joins and other primary relational operations.

Figure 1 shows the execution plan of the example query with instruction dependencies. The majority of these instructions are marked by the optimizer for monitoring, depicted as shaded nodes in the graph. The dark colored nodes are independent from the query template parameters and reused upon next invocation of the same template with different parameters. The light colored part depends on the parameters and is not reused unless the parameter values match or allow for subsumption.

In MonetDB optimizers are independent modules that transform the MAL programs. The SQL compiler comes with a default chain of optimizers. Therefore, the position of the recycler optimizer in this chain requires some care. Evidently, recycling should be performed before we inject garbage collection statements to free up resources. However, it should not be applied too early in the chain either. Optimizers for inlining of SQL (scalar) functions, evaluation of constant expressions, symbolic evaluation to remove empty partial results, and dead code elimination should be called first.

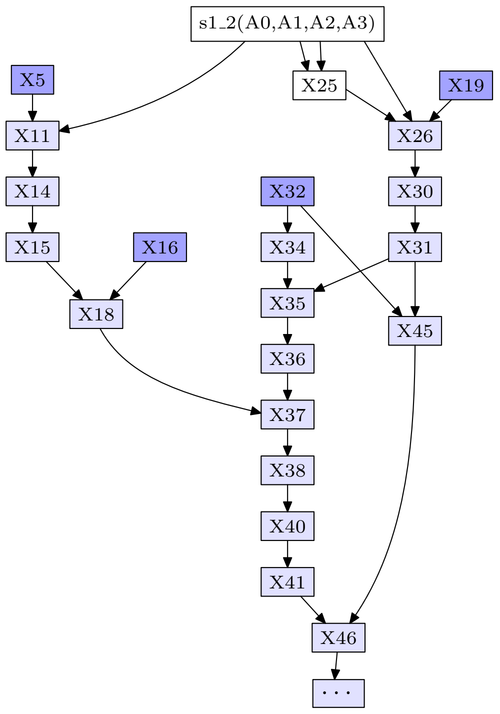


Figure 1: Execution plan marked by the recycler optimizer

3.2 The Recycle Pool

We refer to the system buffer for storing intermediates as *recycle pool*(RP). It is internally represented as a MAL program block, which simplifies its management, inspection and debugging. The recycle pool is filled with the instructions captured, and their arguments and results are stored as constants in the block's symbol table. The instructions are accompanied by execution and reuse statistics such as the CPU time to compute, the sizes of the operands and result, and the number and the type of reuses. In the example RP shown in Table 2 we use the following name conventions for the entries in the symbol table: the parameters of the query template have names starting with A, the variable names start with X, and the constants names begin with TMP.

The content of the recycle pool is managed through a combination of policies. The *admission policy* determines which of the monitored instructions should be kept in the pool. The *cache policy* decides which entries to evict in order to make room for new instructions. It is used to meet the resource limitations, such as the total memory used, and to adapt the content of the pool to the recent workload. Entries are also evicted from the pool when update statements invalidate intermediates derived from the modified persistent columns. The details of the admission and cache policies are given in the subsequent sections.

Symbol Table			
Name	Value	Data type	#Tuples
...	...	...	
X19	642	:bat[:oid,:date]	
TMP1	"sys"	:str	
TMP2	"orders"	:str	
TMP3	"o_orderdate"	:str	
TMP4	0	:int	
X26	1222	:bat[:oid,:date]	57768
A0	1996-07-01	:date	
X25	1996-10-01	:date	
TMP5	1	:bit	
...	...	...	
X142	1527	:bat[:oid,:date]	228626
A5	1996-01-01	:date	
X134	1997-01-01	:date	
...	...	...	

```
...
X19 := sql.bind("sys","orders","o_orderdate",0)
X26 := algebra.select(X19,A0,X25,true,false)
...
X142 := algebra.select(X19,A5,X134,true,false)
...
```

Table 2: Recycle Pool

3.3 The Recycler Run-time Support

The runtime support of the recycler extends the interpreter of MAL query plans. If an instruction is marked for recycling, it is wrapped with two recycler operations: *entry*() and *exit*(). The purpose of the *entry*() operation is to search the recycle pool for a matching instruction and reuse it, if possible, instead of computing the instruction. The *exit*() operation arranges for storing the result of the computed instruction into the recycle pool.

The *entry*() operation performs matching between an instruction to be interpreted and candidates in the recycle pool. Since all arguments are known at run time, matching boils down to comparing instruction types and argument values. It consists of two phases: looking for an exact match and looking for super-set instructions whose results contain the result of the planned one. The mechanism of exact matching is general, while subsumption of instructions is specific for each instruction type.

The reuse of an exact match is straightforward. The result is already available in the pool. It is brought to the execution stack and the wrapped instruction is skipped without execution.

If no exact match exists, but several subsuming instructions are present, the recycler chooses the instruction with the smallest result super-set. The smallest intermediate result replaces the original column operand of the instruction for the time of its execution.

For example, let us assume that a new query comes with a selection predicate on order date that is compiled into the following MAL instruction:

```
X369 := algebra.select(X19,1996-08-01,
                              1996-09-01,true,false)
```

The instruction overlaps with two previously executed selections on order date, whose intermediates are kept in the RP as variables named X26 and X142, respectively. The new instruction is matched to the two super-set instructions, and the intermediate with the smallest number of tuples, X26, is chosen. The recycler runtime support modifies the original instruction for the time of interpretation into the following:

```
X369 := algebra.select(X26,1996-08-01,
                              1996-09-01,true,false)
```

After execution the instruction is restored and its result is eventually added to the RP.

3.4 Recycle Pool: a Cache with Lineage

Instruction matching calls for comparison of all arguments for validity. Since some arguments are results of earlier instructions, matching depends on instruction dependencies and the way the admission and cache policies treat them. Let us consider a sequence of two instructions, $(A; B)$, where the intermediate result of A is an argument of B. If only the result of B is kept in the pool, while the result of A is discarded, we would miss an opportunity for reuse. For, if the sequence $(A; B)$ is computed again, the occurrence of instruction A will be recomputed producing an intermediate object, possibly different from the one used as an argument of the kept copy of instruction B. A comparison of the new object against all objects kept in the pool would be prohibitively expensive. It thus leads to unsuccessfully matching instruction B and the inability to use the result that was kept in the pool. Instruction B would also be re-evaluated and, thereby, pollute the recycler pool even further.

Therefore, dependency analysis is crucial for successful matching and effective recycling and requires instruction dependencies to be respected. This means that both the admission and cache policies have to keep whole threads of execution intact.

3.5 Admission Policies

The *exit*() operation of the recycler is called only if the instruction marked for recycling has indeed been executed. It uses the admission policy to decide about storing the result in the pool. In order to keep the result, a copy of the instruction together with its arguments, results and execution statistics are stored in the recycle pool and thus made available for reuse by subsequent queries. In the presence of limited resources, a recycler routine is called to make room for new instructions.

The recycler supports the following admission policies:

- the KEEPALL is a baseline policy that keeps all instruction instances advised for recycling by the optimizer. It allows for entire execution threads to be stored in the pool and reused later on without disturbing the matching process.
- the CREDIT policy applies an economical principle to resource utilization. Initially every instruction marked for recycling is supplied with a number of credits. Every time an instruction invocation is stored in the recycle pool, the source instruction 'pays' with one credit. The instruction may receive its credits back only upon a reuse of some of its invocations in the pool.

In particular, return of credits distinguishes two types of reuse: In the case of local reuse during the same query invocation, the credit is returned immediately. If a global reuse occurs, i.e. outside the source query invocation, only the reuse statistics are updated. If such a globally reused instance is evicted by the cache policy, the source instruction in the query template receives its credit back. In this way an instruction that has already shown to be useful, has the opportunity to be admitted again to the pool in the future.

If instruction instances are not reused, for example, due to different parameter values, the credits are exhausted after a few invocations. In this case new instruction instances are not admitted to the pool anymore and, thus, cannot claim more resources.

The CREDIT policy respects the instruction dependencies. In contrast to the KEEPALL policy, a thread of execution might be cut off earlier at an instruction that is not reused and has spent its credits. Hence, the CREDIT admission provides almost full recycling opportunities, but with more economic resource use.

Our preliminary experiments with admission policies based on filtering individual instructions irrespective of the dependencies did not prove to be useful. If, for example, the policy filters instructions purely based on their individual CPU cost, it would discard some cheap instructions, such as reverse, but would also cut the opportunity to recycle some expensive dependent instructions, such as joins (see for example ($X36$; $X37$) in Fig. 1). The scope of such negative effects is hard to estimate at run time when complete statistics about dependent instructions is not available before their actual execution. Future work will explore other ideas for admission policies.

3.6 Recycle Pool Maintenance

Keeping around a large number of intermediates and checking for their usefulness at query run time at some point becomes a performance issue. The main sources of overhead are the time taken for instruction matching and the storage to keep the intermediates. To keep this overhead under control the recycler routine *cleanCache* is called when needed to release resources. It uses the *cache* policy to determine which intermediates to evict to make space for the ones more useful for the current load. This process is supported by the execution and reuse statistics of the instructions.

As explained earlier, the recycler cache policy has to respect instruction dependencies. Therefore, the cache policies first select the instructions at the end of the execution threads, i.e., focus on instructions without dependents. The last instruction from the current thread of execution is an exception. It is a probable predecessor of the current instruction and should therefore be protected from eviction. From this set of 'leaf' instructions the cache policy picks one, or several, with the smallest expected utility for the system.

We propose two policies to capture the strongest evidence for utility from recycling: freshness and contribution to performance. The first factor is the time when an instruction has been computed or reused. The second is the benefit that the system has already gained from recycling the instruction.

- LEAST RECENTLY USED (LRU). The traditional LRU policy takes into account the time when an instruction has been computed or most recently reused. It picks the oldest entries for eviction.

- BENEFIT POLICY (BP). The benefit policy takes into account the intermediate's contribution to performance so far and picks entries with the smallest one. The contribution is computed from the cost of the intermediate and a weight factor, $B = Cost * Weight$. The cost presents the resources the system has spent to compute the intermediate, $Cost = f(CPU, IO)$. The reuse weight reflects the number and type of the reuses, $Weight = f(cnt, type)$[3].

Since reuses are an evidence for a return of resource investment, reused intermediates have a bigger weight than non-reused ones. We also observe that if an instruction has been reused only locally, there is no incentive to keep it in the pool after the scope of the query. Hence, the inter-query reused intermediates are weighted more than ones with only internal query reuse. In this way an intermediate with relatively small cost but a large number of reuses might be kept, and one with a high potential benefit(cost) that never 'materializes' in a reuse might be evicted.

The *cleanCache* routine is triggered when a resource limit is reached. Resource limits can be put on the size of the recycle pool memory, the number of entries in the recycle pool, or both. Since the resource pool is located in memory the recycler always watches the hard limit of the physical memory size.

We provide two versions of the BENEFIT policy corresponding to the resource limitation that triggers the eviction. If a single entry needs to be freed, the BP_{ent} policy picks the entry with the smallest benefit $B = min_{I \in L} B(I)$, where L is the set of all leaf instructions.

To address a memory limitation, the BP_{mem} policy has to solve an optimization problem to find a set of the least beneficial instructions that would also release enough memory. Let $M(I)$ be the memory taken to store the result of an instruction I, and M_{req} is the memory required for a new intermediate. The algorithm needs to find the set of instructions E to evict, such that $E \subseteq L$, $\sum_{I \in E} M(I) > M_{req}$, and that minimizes the total benefit $\sum_{I \in E} B(I)$. In practice, we solve the complementary problem, which is a version of the knapsack problem. We find the instruction subset $L - E$ that fits in the knapsack volume $\sum_{I \in L} M(I) - M_{req}$ and has maximum total benefit. To achieve run time performance we use an approximate solution with an upper bound of two times the optimal solution.

In the case of memory limitation, it is possible that the leaf instructions do not release enough memory. Then the cache policies evict all leaf instructions and start another iteration of the algorithm.

4. SKYSERVER EVALUATION

In this section we demonstrate the potential of the recycler in the context of the SkyServer project [20]. SkyServer is a sizable 4 TB scientific database with 91 tables, 51 views, and 203 persistent module functions. The test database deployed in the experiments is a 100 GB subset of SkyServer Data Release 4 (DR4).

All experiments were run on a computer with 4 Dual Core AMD Opteron 2GHz processors with 8 GB RAM and 1 TB of disk space. All reported times were measured with a hot cache. A subset of the query batch was executed first so that the queried columns were read from disk into memory.

4.1 Workload Characteristics

We prepared two query batches of 100 and 500 queries against DR4 randomly picked from the real life query log

[3]Details are available in the code base of MonetDB.

Instruction type	# Cache lines	Memory (MB)	Avg. time (ms)	# Reused Cache lines	# Reuses	Avg. time saved (ms)
Select	29	148	126	22	317	166
Join	78	1221	118	37	1585	249
Bind	44	0	1	34	1836	1
Mark	33	0	1	24	439	1
...	...	...	...	...	...	...
Total	258	1500		170	5711	

Table 3: Characteristics of recycle pool after DR4 query batch

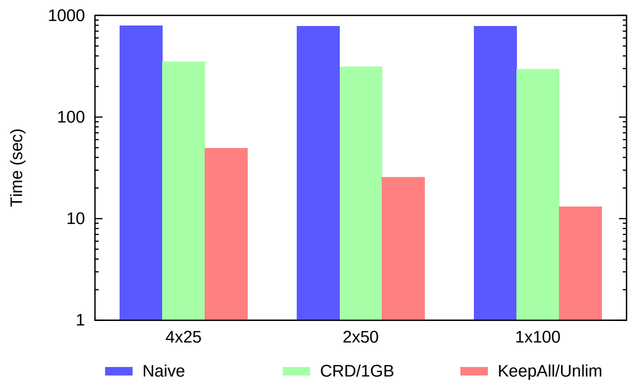


Figure 2: Recycler effect on SkyServer query batch

from January 2008. A manual inspection of the random set confirmed our previous observations [11] of a high percentage of (partially) overlapping queries. The batches contain a small number of patterns over a limited part of the database schema. This is typical for web-based applications where a few tens of query patterns are used with different parameters.

To make the discussion concrete, we illustrate with the most common query pattern in this batch:

```
SELECT p.objID, p.run, p.rerun, p.camcol,
   p.field, p.obj, p.type, ...
FROM fGetNearbyObjEq(195,2.5,0.5) n,
   PhotoPrimary p
WHERE n.objID = p.objID
LIMIT 10;
```

The query accesses the catalogue table with photometric properties of sky objects through the view PhotoPrimary. The object selection is based on their sky location. They are extracted through the table-valued spatial function fGetNearbyObjEq with parameters the location equatorial coordinates and size. A set of 19 popular properties of the objects are projected.

The execution plan computes the spatial function and the view, joins them and performs projection joins to extract the properties. When executed with recycler the majority of the intermediates of those operators is reused.

When the 100-query batch ran with the keepall admission and unlimited storage, the recycler monitored the execution of 5969 instructions, constituting approximately 15% of total instructions executed. 5711, or 95.6% of the monitored instructions, were successfully reused. Table 3 breaks down the content of the recycle pool at the end of the batch execution.

The total storage overhead is 1.5 GB which is approximately 50% of the 2.9 GB taken by the columns queried, or less than 2% of the total database size. Except several intermediates of size smaller than 1 KB each, all the memory taken by intermediates is reused. The join intermediates are the major consumers of memory, but also contribute most substantially to the time savings. They have both a high number of reuses and significant saved time per reuse. We also observe a larger percentage of reused selections than reused joins, since selections are typically predecessors of joins in the execution threads.

4.2 Workload Performance

Figure 2 illustrates the total time for the 100 query batch with and without recycler intervention. The naive strategy denotes regular execution without recycling. We ran two recycler versions: one with keepall admission and unlimited storage and one in a resource limited mode, i.e., with crd/lru policies and memory limited to 1 GB, constituting 65% of the memory taken by the unlimited version.

To simulate the effect of updates invalidating the intermediates cache we split the 100 query batch into shorter sequences of 25 and 50 queries and ran them with cleaning the RP in between.

The effect of the recycler keepall/unlimited on the response time is significant: it dropped from 785 sec to 14 sec for the 1x100 batch. Since the percentage of reused memory is very high for this workload, any shortage substantially affects the performance. Still, the total time of crd/lru/1gb is 296 sec, or approximately 38% of the naive strategy time.

The batches of 4x25 and 2x50 queries show similar performance with a small overhead due to the loss of intermediates from the previous batch that have to be computed again. To verify the results we scaled the experiment to 500 randomly selected queries. The times measured confirmed the observations from the shorter batches: the naive strategy ran for 4057sec, the keepall/unlimited achieved 17sec, and the crd/lru/1gb strategy took 1433sec, i.e., approximately 35% of the naive strategy time.

Analysis of the recycled instructions showed that the recycler had detected and effectively materialized the queried projection over the PhotoPrimary view without human intervention. The original database schema on the SQL Server implementation might have benefited from materializing this view or using some index structure. Whether a workload analyser would have detected it remains to be seen.

Query	Instructions			Time (s)			
	#	Intra %	Inter %	Total	Savings		
					Pot.	Local	Glob.
Q1	36	2.8	0	5.72	3.54	0.30	0
Q2	106	0.9	2.8	0.22	0.22	0	0.07
Q3	39	0	5.1	2.61	2.40	0	0
Q4	36	0	41.7	1.72	1.65	0	1.44
Q5	74	0	2.7	1.16	1.15	0	0
Q6	11	0	0	0.53	0.52	0	0
Q7	106	3.8	3.8	1.61	1.11	0.36	0.56
Q8	61	0	6.6	0.60	0.56	0	0.16
Q9	59	0	3.4	1.38	1.25	0	0
Q10	54	0	3.7	1.37	1.34	0	0.20
Q11	36	33.3	2.8	0.16	0.16	0.03	0
Q12	6	0	33.3	1.17	0.55	0	0
Q13	17	0	11.8	2.88	1.27	0	0
Q14	13	0	0	0.27	0.27	0	0
Q15	12	0	0	0.23	0.19	0	0
Q16	14	0	42.9	0.88	0.27	0	0.01
Q17	29	0	3.4	0.96	0.95	0	0
Q18	12	0	75.0	1.83	1.70	0	1.68
Q19	39	15.4	7.7	3.72	1.69	0.99	0.49
Q20	25	0	12.0	0.95	0.82	0	0.01
Q21	154	9.1	12.3	5.80	5.38	0.72	2.94
Q22	4	0	75.0	0.65	0.15	0	0.15

Table 4: Characteristics of TCP-H queries

5. TPC-H EVALUATION

To gain an understanding of the recycler mechanisms and its effect on the query performance we conducted experiments with the TPC-H Decision Support benchmark [23]. The experiments ran against a database of scale factor 1 (SF1), i.e., of size approximately 1 GB. The times reported were again measured in a hot cache.

First, we analysed the queries with respect to commonalities that can potentially bring benefits from reuse of intermediates. We distinguish two types of commonalities: intra- and inter-query. The intra-query (or local) type describes the cases when common sub-expressions exist within a single query plan, typically among sub-queries or between a sub-query and the main query. The inter-query (or global) type refers to different query invocations sharing common sub-queries in the TPC-H workload. These can be different queries or different instances of the same query pattern.

Table 4 shows the commonality characteristics of the TPC-H queries. The *Instructions/#* column contains the total number of instructions marked by the recycler optimizer for monitoring. We will call those the *potential hits* of the recycler. The number does not include instructions that bind columns to variables. Although those instructions are monitored and reused, they do not constitute common computations. The *Intra* and *Inter* columns show the percentage of marked instructions that are locally, respectively globally, reused. To estimate the inter-query commonalities we assume that the same query is executed with different parameters, where the parameter generation follows the TPC-H specification. In this analysis we do not include inter-query commonalities among different queries. These depend strongly on the application and are hard to estimate in general.

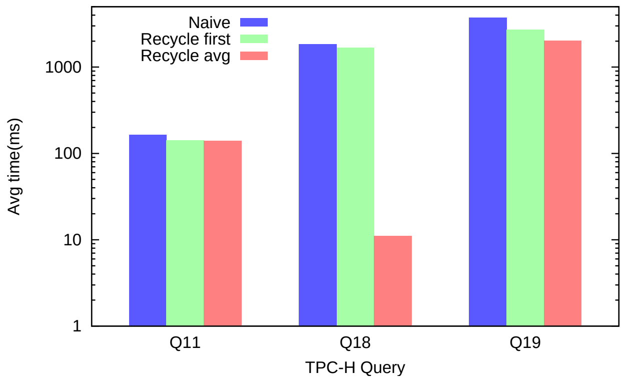


Figure 4: Recycler effect on performance

Given the algebraic framework, this table highlights the opportunities for reuse independent of the technique deployed. Equally, it indicates the limited opportunities for reuse when a pipelined architecture is considered.

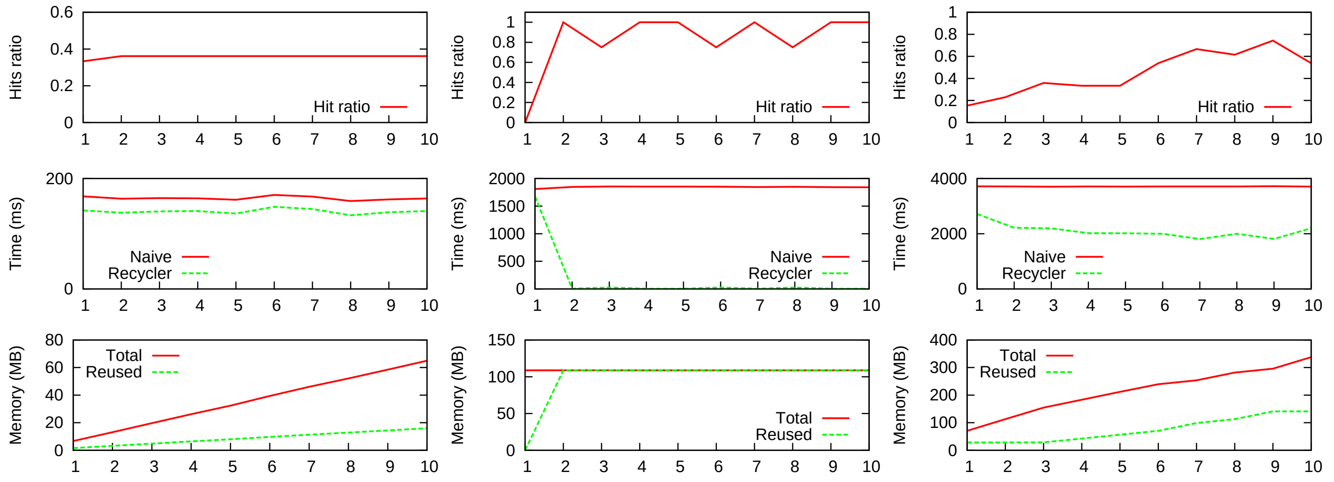
The right side of table 4 shows the total execution times of the queries against MonetDB together with time savings from the recycler. We summarize the potential savings, i.e. the total time spent in monitored instructions, the realized intra-query savings and those from a single inter-query reuse. A manual inspection of the execution traces of queries demonstrating high time savings, such as Q4, Q18, and Q21, confirmed that the reused instructions have high cost. For other queries, such as Q16, time savings are minimal despite the high percentage of instructions reused. The recycled instructions are too cheap in this case.

5.1 Micro- Benchmarks

Next, we examined the recycler effect on the performance and resource utilization over four groups of queries: with prevailing local, prevailing global, or mix commonalities, as well as queries that do not exploit overlaps. We chose queries that are typical representatives of each of the above groups. In this micro-benchmark we executed 10 instances of each query generated with the TPC-H query generator. To illustrate the recycler mechanics better, the admission policy is KEEPALL and there are no resource limitations, thus no cache policy interferes with the results.

Query Q11 contains substantial intra-query commonality where a large part of the sub-query is shared with the main query. The common part includes a selection, a 2-way join, and an arithmetic computation over projected attributes. The profile of the query is shown in Fig. 3a. The top diagram shows the hit ratio of individual queries. It is the ratio of the hits in the recycle pool (successfully recycled instructions) and the potential hits. Due to the intra-query commonalities, we observe recycle pool hits and time improvements (in the middle diagram) from the very first query instance. Since the inter-query overlap is negligible, the time improvement and the hit ratio are stable for all instances. The bottom diagram shows the RP memory, i.e., the cumulative memory consumption for intermediates, after each query instance. It grows with a stable rate: each query adds its own intermediates different from the previous ones.

Query Q18 illustrates the inter-query type of commonalities among different instances. Its sub-query groups the rows of the LINEITEM table on the foreign key and selects



(a) Intra-query(Q11) (b) Inter-query(Q18) (c) Inter- and Intra-query(Q19)

Figure 3: Recycler effect with different type of query commonalities

the groups with an aggregate function value above a certain level. Grouping of rows and computing the aggregate function is the overlapping computation between instances of Q18 that differ only in the value of the selected quantity level. Thus, when the intermediates of grouping and aggregate functions are kept in the recycle pool after the first execution, every subsequent instance of Q18 reuses them and computes only the remainder of the plan that depends on the query parameter. In the case of Q18 the grouping and aggregation are also the main ingredient of the processing time. The query profile shown in Fig. 3b reveals how the inter-query commonalities are used by the recycler. The first query instance has a very low hit ratio and time improvement, but high memory consumption for the intermediates kept in the pool. The subsequent queries achieve very high hit ratio and time savings. The time goes from 1.8s for the first execution (SF1) to 24ms for all subsequent executions with 75% hit ratio, and to 1ms for executions with 100% hit ratio. The memory diagram shows that all intermediates are reused and no sizable new intermediates are added to the pool.

Traditionally, such a query can be sped up by a materialized view storing the groups of the LINEITEM table rows together with the computed aggregates. We experimented with a version of query Q18 using a materialized view and observed performance comparable with the recycler, namely 2ms per query instance using the same experimental settings. What the recycler brings in addition to performance is flexibility. If grouping attributes or the aggregate functions change slightly, the recycler will keep the modified intermediates and automatically adapt to the workload change without human guidance.

Query Q19 has a mixture of intra- and inter-query overlaps. It contains three sub-queries with a number of predicates overlapping both inside a query and among different instances. In the query profile in Fig. 3c we observe some hits in recycle pool and time improvement in the first instance due to the intra-query commonalities, followed by larger improvements and a higher hit ratio for subsequent instances due to the combined effect of intra- and inter-query commonalities. The average performance improvements for the 10-instance benchmarks of the above queries are illustrated in Fig. 4.

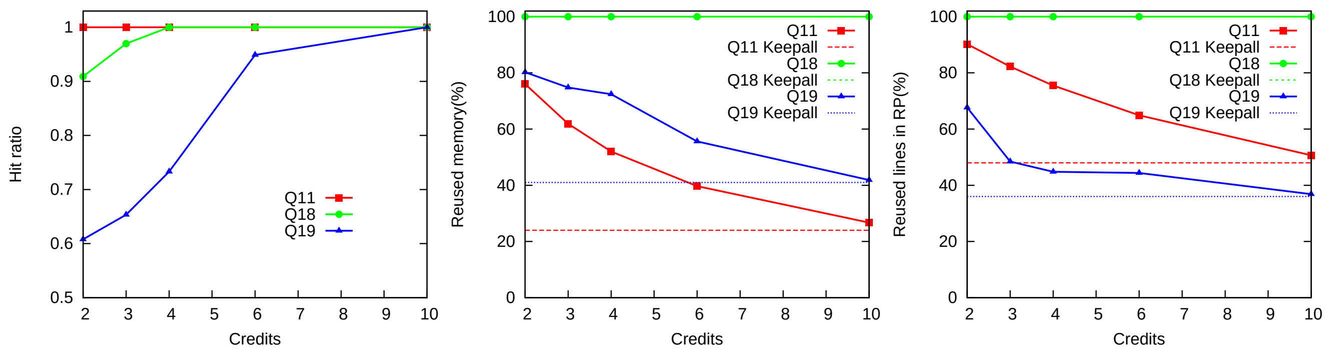
As a counter example of a query for which the recycler is not efficient we consider Q14. Although a number of instructions are monitored by the recycler, all the invocations have different parameters and are in practice not reusable. Hence, the query demonstrates rather the potential overhead of the recycler for storing and matching instructions. Each invocation adds 13 instructions to the recycle pool and allocates 5.5 MB for the intermediates without amortizing this resource investment in the form of performance improvements. The memory profile of Q11 and Q19 shows that queries with overlaps may also accumulate non reused intermediates. Having observed this danger, we turn our attention to the admission policies that can prevent waste of resources by early filtering of intermediates at the admission to the RP.

5.2 Evaluation of Admission Policies

In this section we evaluate the CREDIT admission policy in terms of resource utilization and the number of RP hits achieved. The base line for comparison is the KEEPALL policy that stores all the instructions designated for recycling. Figure 5 shows the hit ratio to the base line(a), the percentage of reused memory(b), and reused RP entries(c), as the number of credits increases. The experiments were run in unlimited resource settings.

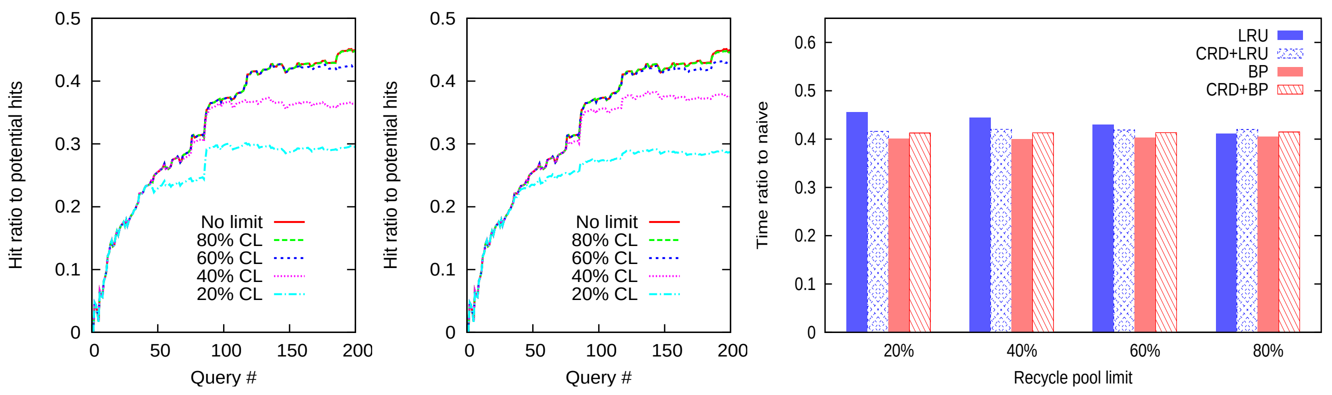
The number of credits affects the hit ratio for inter-query commonalities (Q18 and Q19). Having a small number of credits prevents keeping and reusing some of the overlapping intermediates, but also improves the resource utilization (Q19). As the number of credits increases, the hit ratio improves, simultaneously with degradation in resource utilization in terms of larger sizes and lower percentage of reuses. In the case of Q18 both admission policies use 100% of memory and RP entries and the resource utilization is independent of the credit parameter.

Credits do not affect the hit ratio for intra-query commonalities (Q11), since local reuses return the credit imme-



(a) Hit ratio to Keepall policy (b) Reused memory (c) Reused entries in RP

Figure 5: Effect of credit parameter to resource utilization and RP hits



(a) LRU policy hits (b) Benefit policy hits (c) Relative times to naive

Figure 6: Cache policies in limited recycle pool

diately to the source instruction. However, having a small number of credits successfully limits the admission of non-reused instructions and substantially improves memory and recycle pool entry utilization. To balance the resource use vs. hit ratio we set the number of credits to four when we use CREDIT admission in the rest of the experiments.

5.3 Evaluation of Cache Policies

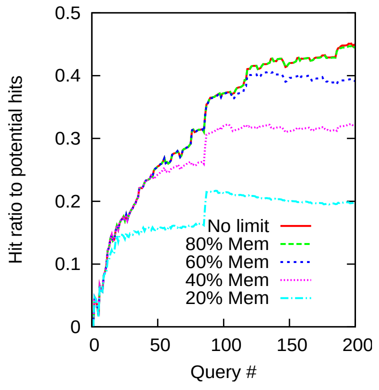
The cache policy has a different influence over different types of query overlaps. The intra-query commonalities are, with a few exceptions, not affected. The instructions are always put in the pool and reused in the course of the current query execution, since they are protected from eviction. The LRU policy does not touch them as they are most recent in the pool. The benefit policy is designed to exclude them from the list of eviction candidates. The exceptions arise for complex queries that cannot fit in the pool, thus incurring eviction of instructions earlier in the plan.

The reuse of inter-query commonalities is strongly influenced by the cache policy. If the policy cannot distinguish instructions with potential reuse and throws them out, this directly leads to a higher number of RP misses and reduced improvement of performance.

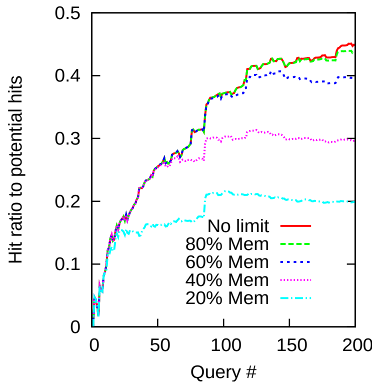
To evaluate the cache policies we selected 10 TPC-H queries (4,7,8,11,12,16,18,19,21, and 22) with relatively large overlaps to create a bigger contention between reused instructions, a situation where the punishment for a bad choice of the evicted instruction is more noticeable. We created a batch of 200 queries by mixing 20 instances of each. First, the batch was run with the KEEPALL/UNLIMITED strategy to measure the total resources needed (4 GB memory and 5219 RP entries), as well as the percentage of the reused resources (42.7% reused memory and 28% reused entries). Then we ran the batch using each of the cache policies with resources limited to a percentage of the total resources. We consider two major resources: memory taken by the intermediates, and number of RP entries which affects the instruction matching time.

Figure 6 shows the effect of the cache policies when limiting the number of recycle pool entries(also called cache lines, CL). The cumulative hits from the batch execution are shown with respect to the cumulative potential hits. For limits that fit the reused entries(>40%) the hit ratio is almost not affected. For the 20% limit the hit ratio drops to 0.3 of the potential hits. Still, both policies run for less than 45% of the time of the NAIVE strategy (Fig. 6c). Although the BENEFIT(BP) policy shows lower hit ratio than LRU in some cases, it succeeds better in distinguishing and keeping in the pool weighty intermediates. For all limits BP achieves the best performance running for 40% of the total time of the NAIVE strategy.

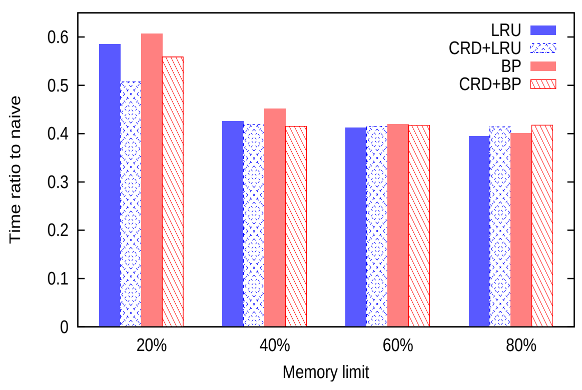
The effect of CRD admission policy is two-fold. When combined with BP it leads to a small loss of performance due to some hit misses. The combination of CRD and LRU improves



(a) LRU policy hits



(b) Benefit policy hits



(c) Relative times to naive

Figure 7: Cache policies in limited memory

the LRU, especially when a large limitation is imposed. The early filtering of non reused instructions achieved by the CRD admission manages to keep old reused instructions longer in the RP in situations of high resource contention.

Figure 7 shows the policies behaviour in limited memory. This limitation affects both the hit ratio and the processing time more substantially than the RP entries limit. The reason is that some of the beneficial intermediates occupy a lot of memory and need to be evicted to fit the resource limitation. In this case the simpler LRU policy or its version in combination with CRD admission show to be more efficient.

6. RELATED WORK

Recycling exploits the generic idea of storing and reusing expensive computations. Unlike low level instruction caches, we use an optimizer to pre-select instructions to look after. Likewise, the eviction policies respect the semantic dependencies amongst operators in the query plan, thus maintaining an operator cache with lineage.

Recycling (partial) results is also the driver behind materialized views and query caching [1, 5, 8, 16, 22, 25]. Our approach differs from this large body of work in some or all of the following three aspects: self-organizing behavior without human intervention, integration with the DBMS software stack, and granularity of operation. The materialized views are often defined by the DBA with the help of workload analysers to improve system performance [1, 16]. The dynamic materialized views proposed in [25] materialize hot subsets that adapt to the current workload by means of additional control tables. The content of the control tables is updated manually by the DBA or automatically by a cache controller. The definition of the views and associated control tables is again a responsibility of the DBA.

Traditionally, view or intermediate matching is integrated with the query optimization [5, 8, 22]. Often this involves applying advanced algorithms over graph representations of query plans. Recycling does not modify query plans based on the intermediates available. Instead, it matches instructions one-at-a-time at run time. Hence, matching intermediates is interleaved with query execution. This is possible due to the abstract representation of query plans and MonetDB's execution paradigm. In the Cache-on-Demand framework [22] recycling of intermediates is ensured by considering only the present. An overlap in an incoming query with the currently running queries triggers materializing common intermediates. This approach is beneficial in a multi-user scenario setting, but imposes temporal locality limitations over the overlapping queries.

Finally, recycling is a general technique that works at a finer level of granularity than materialized views and query caches. It keeps individual instruction results independently from the source of commonalities and the type of the entire query. DynaMat [12] proposes dynamic management of a pool of materialized views in data warehouses. Similarly to the recycling policies, so called goodness metrics are employed to automatically decide which views to keep and which to evict from the pool. In this way the system adapts the pool content for maximal benefit of the current workload. Working in the context of data warehouse and decision support applications, DynaMat considers specific types of data cube queries, called multidimensional range queries, whose final results are put in the pool. A similar line of research is pursued in [6, 15]. In contrast, recycling considers intermediate results at the instruction level and is a general technique that does not impose limitations on the query types.

Database caching [3, 13] is typically used in distributed settings to augment the mid-tier application servers and to off-load the back-end database servers. The cache content is a DBA-defined collection of materialized views and thus is static with respect to the covered database sub-schema.

The adaptive replication technique presented in [10] exploits the materialization of selection intermediates to reorganize a persistent table column into a partial replica tree. Recycling is a general technique that manages the intermediates of different classes of relational operators which does not change the underlying column structures.

Our approach to share computations between queries also relates to multi-query optimization [19] and exploitation of similar sub-expressions [24]. Both techniques are applicable to queries that are known in advance and executed concurrently, such as query batches, sub-queries, and maintenance of materialized views. In contrast, the recycler alters the execution of individual queries to maximally benefit from intermediates currently available in the pool and, hence, is applicable to individual ad-hoc queries without a-priori knowledge about the workload.

7. SUMMARY AND CONCLUSIONS

In this paper we have described a database architecture augmented with recycling intermediates, i.e., caching partial results, in a relational algebra program. The architecture is implemented as an extension to the MonetDB system. Our approach addresses the overhead incurred by its operator-at-a-time execution paradigm and transforms it into a benefit without manual intervention.

The recycling policies respect the inter-operator dependencies, which leads to effective reuse of large threads in template based query sessions, e.g., web applications. The recycling policies studied cover both an LRU scheme and one driven by an economic principle. The extensive experimentation based on a full-fledged implementation shows that the MonetDB software architecture is well suited to be extended with a targeted optimization goal.

The experiments using the SkyServer real-life query set illustrate that even in a single system, recycling of partial results can lead to significant benefits. Primarily because the cost of the expensive body part of a query is reduced.

The results obtained indicate several areas for further exploration. Within the context of MonetDB, it seems worth exploring subsumption relationships through join paths and the opportunities offered by recognition of query classes. Another direction of work is refining and developing admission and cache policies that respect the semantic dependencies of instructions, as well as automatic accommodation of policies most appropriate for the current workload. But first, and foremost, the technique seems amendable to pipelined architectures by tapping the stream at selected points in the query operator tree. To our knowledge, publicly available experimental proof of this is still lacking.

8. ACKNOWLEDGMENTS

The authors would like to thank the members of the MonetDB database group, in particular Sjoerd Mullender, Stratos Idreos, and Stefan Manegold. We are also grateful to the SkyServer team for providing the data. This work was supported by the dutch Bsik-Bricks and MultimediaN research programs.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, pages 496–505, 2000.

[2] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *Commun. ACM*, 51(12), 2008.

[3] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive Database Caching with DBCache. *IEEE Data Eng. Bull.*, 27(2):11–18, 2004.

[4] N. Bruno and S. Chaudhuri. Physical Design Refinement: The 'Merge-Reduce' Approach. *ACM Trans. Database Syst.*, 32(4), 2007.

[5] C.-M. Chen and N. Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *EDBT*, pages 323–336, 1994.

[6] C.-H. Choi, J. X. Yu, and H. Lu. Dynamic Materialized View Management Based on Predicates. In *APWeb*, pages 583–594, 2003.

[7] R. Cornacchia, S. Héman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Flexible and Efficient IR Using Array Databases. *VLDB J.*, 17(1):151–168, 2008.

[8] J. Goldstein and P.-Å. Larson. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *SIGMOD Conference*, pages 331–342, 2001.

[9] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[10] M. Ivanova, M. L. Kersten, and N. Nes. Self-organizing Strategies for a Column-store Database. In *Proc. EDBT*, pages 157–168, 2008.

[11] M. Ivanova, N. Nes, R. Gonçalves, and M. L. Kersten. MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database. In *Proc. SSDBM*, Banff, Canada, July 2007.

[12] Y. Kotidis and N. Roussopoulos. A Case for Dynamic View Management. *ACM Trans. Database Syst.*, 26(4):388–423, 2001.

[13] P.-Å. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *ICDE*, pages 177–189, 2004.

[14] G. Luo. Partial Materialized Views. In *ICDE*, pages 756–765, 2007.

[15] G. Luo and P. S. Yu. Content-based Filtering for Efficient Online Materialized View Maintenance. In *CIKM*, pages 163–172, 2008.

[16] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *SIGMOD Conference*, pages 307–318, 2001.

[17] http://monetdb.cwi.nl/, 2008.

[18] T. Phan and W.-S. Li. Dynamic Materialization of Query Views for Data Warehouse Workloads. In *ICDE*, pages 436–445, 2008.

[19] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD Conference*, pages 249–260, 2000.

[20] Sloan Digital Sky Survey / SkyServer, 2008.

[21] A. S. Szalay, J. Gray, et al. The SDSS SkyServer: Public Access to the Sloan Digital Sky Server Data. In *SIGMOD*, pages 570–581, 2002.

[22] K.-L. Tan, S.-T. Goh, and B. C. Ooi. Cache-on-Demand: Recycling with Certainty. In *ICDE*, pages 633–640, 2001.

[23] Transaction Processing Performance Council. TPC Benchmark H, Revision 2.6.2, 2008.

[24] J. Zhou, P.-Å. Larson, J. C. Freytag, and W. Lehner. Efficient Exploitation of Similar Subexpressions for Query Processing. In *SIGMOD Conference*, pages 533–544, 2007.

[25] J. Zhou, P.-Å. Larson, J. Goldstein, and L. Ding. Dynamic Materialized Views. In *ICDE*, pages 526–535, 2007.

[26] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. ICDE*, Atlanta, GA, USA, 2006.