# An Architecture for Recycling Intermediates in a Column-store

MILENA G. IVANOVA
MARTIN L. KERSTEN
NIELS J. NES
ROMULO A.P. GONÇALVES
Centrum Wiskunde & Informatica

Automatic recycling intermediate results to improve both query response time and throughput is a grand challenge for state-of-the-art databases. Tuples are loaded and streamed through a tuple-at-a-time processing pipeline, avoiding materialisation of intermediates as much as possible. This limits the opportunities for reuse of overlapping computations to DBA-defined materialised views and function/result cache tuning.

In contrast, the operator-at-a-time execution paradigm produces fully materialised 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-organising behaviour and its significant performance gains. The results indicate the potentials of recycling intermediates and charts 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

Additional Key Words and Phrases: 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 optimised in isolation using statistics

Authors' address: Centrum Wiskunde & Informatica (CWI), Science Park 123, 1098 XG Amsterdam, The Netherlands. Email:{milena,mk,niels,goncalve}@cwi.nl.
**Disclaimer**. This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2010 ACM 0362-5915/2010/0300-0001 $5.00

gathered, analytical models, and heuristic rewrite rules, and run 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: materialised views and reuse of (partial) results. Both depend and interact heavily with the underlying architecture, its execution paradigm and opportunities for optimisers 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 materialising intermediates [Graefe 1994]. 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 materialised views or function/query result set caches (Oracle, DB2, SQLServer). Materialised views have been extensively researched in recent years [Mistry et al. 2001; Goldstein and Larson 2001; Zhou et al. 2007a]. They represent common sub-queries, whose materialisation improves subsequent processing times. The view management component of an optimiser takes them into account while exploring the space of alternative execution plans. Typically, a database administrator supported by workload analysers determines which portions to materialise [Agrawal et al. 2000; Bruno and Chaudhuri 2007]. Reuse of partial results is also useful in applications with parametrised queries [Zhou et al. 2007a; Luo 2007; Phan and Li 2008].

The operator-at-a-time execution paradigm, where complete intermediates are a by-product 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-organising 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 capitalise on the algebra semantics. In contrast to materialised 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 [MonetDB 2010]. 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 realisation of our idea requires a modification in the MonetDB query ex-

ecution engine. Therefore, its abstract machine interpreter is hooked up with a *recycler* optimiser and run-time module. The optimiser marks operations of interest for harvesting. The run-time 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 manage 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 eviction 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.

We describe the design space to accommodate volatile environments in Section 6. It relies on delta-based relational update processing, an area well covered in the database research area.

The recycler is evaluated extensively in two experimental settings. First, we focus on the TPC-H decision support benchmark [Transaction Processing Performance Council 2008]. 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 relatively controlled manner. The benchmark is used to analyse the baseline performance and the impact of different design choices.

Next, we conduct experiments with the SkyServer application [SkyServer 2008]; 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 [Szalay et al. 2002]. In addition, we use the SkyServer application to prepare micro-benchmarks for evaluation of the subsumption algorithm.

The results obtained in the context of MonetDB are, in principle, applicable in a tuple-at-a-time execution paradigm [Graefe 1994]. 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) materialised views. However, experimental proofs of this hypothesis should come from their code owners.

The remainder of the paper is organised as follows. Section 2 provides an overview of the MonetDB architecture and its abstract relational algebra engine. Section 3 describes the overall recycler architecture and discusses different design alternatives. The policies for management of the resource pool are presented in Section 4. Subsumption of instructions is presented in Section 5. Section 6 describes the issues arising when recycling volatile databases and charts the landscape of solutions. The experimental evaluations are presented in Section 7 and Section 8. Section 9 describes the related work and Section 10 summarises our findings.

## 2. BACKGROUND

In this section we give a summary of the MonetDB architecture[1] focusing on the processing model for SQL.

### 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 [Boncz et al. 2008; Zukowski et al. 2006; Cornacchia et al. 2008]. 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 relational structures called Binary Association Tables (BATs). A BAT represents a mapping from an OID to a base type value ANY, i.e., it is a binary table with schema *BAT(head:*OID*, tail:*ANY*)*. This storage structure is equivalent to large, memory-mapped dense arrays. It is complemented with hash-structures for fast key look-up. Associated BAT properties are used to steer selection of more efficient implementations, 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 optimised 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 execution. Powerful tools create an environment where debugging database optimisers has become feasible.

The next layer is formed by a series of targeted query optimisers. They take a MAL program and transform it into an improved one. Two dozen optimiser 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 optimiser.

The top layer consists of front-end compilers (SQL, XQuery), that translate high-level queries into MAL plans. The compilers include optimisers 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

[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","li_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;
```

Fig. 1. MAL plan of the example query

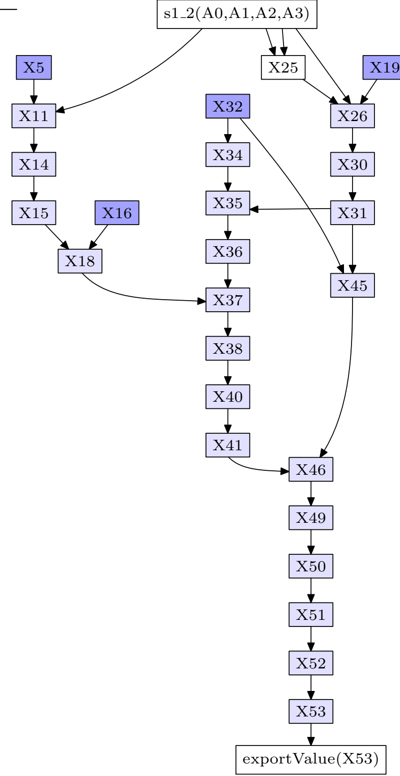



Fig. 2. Execution plan marked by the recycler optimiser

database kernel innovations and domain specific optimisations as studied in this paper.

## 2.2 Query Processing

In this work we focus on the SQL front-end. All SQL queries are translated into a parametrised 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. Template-based query optimisation in MonetDB aligns well with the idea of recycling, since it provides more possibilities for reuse among template instances with different parameters.

Plan generation exploits 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. Figure 1 illustrates the MAL query template produced for an example query over the TPC-H database:

```
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 such instructions. The function body is a linear representation of the query plan. It may appear complex at first sight, but this is mostly a consequence of the canonical representation of the binary relational algebra being supported. The plan is composed of three abstract parts: instructions for catalogue and persistent data access, binary relational algebra instructions, and construction of an SQL query result set.

The data access is implemented by the *bind* instruction and its variations. It localises the persistent BATs for ORDERS (X19,X32) and LINEITEM tables (X5) in the SQL catalogue. Since the query joins the two tables on the foreign key constraint from LINEITEM to ORDERS, a supporting join index is accessed (X16) instead of accessing the persistent BAT for the L_ORDERKEY column in the LINEITEM table.

The major part of the plan consists of binary relational algebra instructions. There are several versions of *select* and *join* that implement numerous algorithms for predicate-based filtering and joins.

In addition, there is a set of MAL-specific instructions with auxiliary functionality. For example, *markT* creates a BAT with the same head as its argument and a fresh dense sequence of unique OIDs in the tail. The typical usage is to create OIDs for the query or partial result tuples. The *reverse* instruction swaps the places of the head and the tail of a BAT. The *mirror* returns an image of a BAT where the tail column is a mirror of the head. These auxiliary instructions are zero cost operations without data copying. They only materialise a new viewpoint over the underlying data structures[2].

Finally, the *exportValue* ships the computed result set to the SQL front-end.

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

Figure 2 shows the MAL plan as a directed graph with instructions at the nodes, and edges representing instruction dependencies. 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. On multi-core systems the threads are executed in parallel.

The final MAL program is interpreted in a linear fashion by the MonetDB kernel. The overhead of the interpreter is kept low, well below one microsecond per

[2]More detail on the MAL instructions can be found on http://monetdb.cwi.nl

instruction. The default interpreter is fully equipped with run-time debugging and performance monitoring. However, if performance measurements are not needed, a fast-path interpreter is called.

### 2.3 Materialisation

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

Do not be misled to overestimate the resource cost related to materialisation, because the MonetDB kernel extensively uses data structure sharing to minimise the need for taking a complete copy. Many instructions are primarily aimed at administration of the properties or viewpoints. As explained above, instructions such as *mirror*, *markT*, and *reverse*, are zero cost operations that only materialise a new viewpoint over the underlying structures. Even a range select operation may become a cheap operation when the underlying BAT happens to be ordered. Then a BAT view is returned, which only keeps a reference to the underlying BAT and the range of qualifying tuples.

The materialisation of intermediates is usually considered an overhead that is avoided in the pipelined execution paradigm. However, full materialisation benefits from 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 optimiser and run-time 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 optimiser 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. In this way the optimised query templates are independent of the intermediates currently available and are readily reusable.

### 3.1 Marking 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 optimiser*. 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 over-

head 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 should be handled with caution. Updates are not candidates for recycling, but they affect the content of the pool of intermediates. Every time a BAT is updated, any copy, or derivation of it retained in the pool should be invalidated or synchronised. Therefore, the update volatility puts a boundary on the effectiveness of recycling. We consider this problem in more detail in Section 6.

Since processing an SQL query starts with binding variables to persistent columns using catalogue names, the net effect is that the recycler optimiser 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 2 shows the execution plan of the example query with instruction dependencies. The majority of these instructions are marked by the optimiser for monitoring, depicted as shaded nodes in the graph. The dark coloured nodes are independent from the query template parameters and reused upon next invocation of the same template with different parameters. The light coloured part depends on the parameters and is not reused unless the parameter values match or allow for subsumption.

The position of the recycler optimiser in the chain of MonetDB optimisers requires some care. Optimisers are modules glued together in a pipeline that transform the MAL programs. Components at the start of the pipeline typically view the MAL program as a linear representation of a declarative query plan, which allows for ease of re-ordering. Modules along the pipeline will gradually inject execution specific properties, e.g., parallel execution guidelines, which frame the execution threads, and, thereby, limit the opportunities for re-arrangement without clear knowledge of plan run-time properties.

The SQL compiler comes with a default chain of optimisers. 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. Optimisers for in-lining of SQL (scalar) functions, evaluation of constant expressions, symbolic evaluation to remove empty partial results, and dead code elimination should be called first.

### 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 as well, 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 I we use the following naming 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 constant names begin with TMP.

| 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 I. Recycle Pool

The content of the recycle pool is managed through a combination of policies. The *admission policy* determines which of the monitored instructions should be added to the pool. The *eviction 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 may also be evicted from the pool when update statements invalidate intermediates derived from the modified persistent columns. The details of the admission and eviction policies are given in Section 4.

A separate policy is to decide on the memory hierarchy impact. For large main memories and relatively small intermediates, a portion of memory for recycling can be shared with main stream processing. Alternatively, the recycle pool can consume all slack resources on the hard disks (SSD) provided write/read cost are significantly less than reconstruction of the intermediate. In the remainder of this paper we limit the size of the recycle pool to a fraction of main memory and let the underlying operating system manage it through its memory mapped file functionality.

### 3.3 The Recycler Run-time Support

The run-time 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: $recycleEntry()$ and $recycleExit()$, shown in Algorithm 1.

The purpose of the $recycleEntry()$ operation (lines 9-17) is to search the recycle pool for a matching instruction and reuse it, if possible, instead of computing the instruction result. Since all arguments are known at run time, matching boils down to comparing instruction types and argument values. It consists of two phases:

**Algorithm 1** Recycler Run-time Support

```
1:  Input: MAL instruction I
2:  Global variable to recycle pool RP
3:  if marked(I) then                                  ▷ I is marked for recycling
4:      if ¬recycleEntry(I) then
5:          execute I
6:          RECYCLEEXIT(I)
7:  else                                   ▷ Regular execution without recycling
8:      execute I
9:  function RECYCLEENTRY(I)
10:     M ← match(I)
11:     if M then                                 ▷ Found matching intermediate
12:         retrieve M to the execution stack
13:         update statistics for M
14:         return true
15:     else
16:         return false
17: end function
18: procedure RECYCLEEXIT(I)
19:     if admission(I) then                ▷ Admission policy decides to keep I
20:         if resSize(I) > freeSpace(RP) then
21:             CLEANCACHE(resSize(I))
22:         add result of I to RP
23: end procedure
```

looking for an exact match and looking for super-set instructions whose results contain the result of the planned one. 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, the recycler looks for subsuming instructions. Once found, it modifies the instruction for a more efficient execution. We describe the details of the subsumed execution in Section 5.

If neither an exact match instruction is found, nor subsumption is possible, the instruction is executed normally. Then the *recycleExit*() procedure (lines 18-23 of Algorithm 1) arranges for storing the result of the instruction into the recycle pool.

### 3.4 Design Alternatives of Run-time Matching

The recycler is charged with the task to both capture and reuse intermediates at run time. Therefore, in its design we aimed for a very light-weight implementation with minimal overhead since it is potentially performed for each interpreted MAL instruction. Several matching strategies can be considered.

Alternative 1: *bottom-up sequence matching*. Instructions are matched syntactically one-at-a-time, which through interpretation leads to bottom-up matching of evaluation plans. Matching of an instruction pair comprises comparing of their type, followed by comparing the number and value of its arguments. Note that

value comparison is possible at run time. This approach requires the least intrusive modification of the execution engine with a negligible overhead ($< 1$ microsecond).

However, it also limits the intermediates cache management to properly respect instruction dependencies. In particular, all prefix-instructions producing intermediates that are arguments of a given instruction A have to be kept in the pool and matched in order to enable successful matching of A. In fact, the lineage of an instruction A is kept by keeping the prefix-instructions. This requirement has associated storage costs and may affect the recycle pool usage and efficiency.

The storage overhead of prefix-instructions should not be overestimated. Many instruction types, e.g., reverse and mark, materialise only alternative viewpoints (i.e., metadata) over the data. They lead to minor adjustments and auxiliary information in the BAT run-time administration.

Alternative 2: *top-down sequence matching*. This alternative uses a more sophisticated lineage mechanism that allows matching of instructions independently of whether their prefix-instructions are kept in the recycle pool. The intermediates are annotated with lineage records with meta-information about the source columns and the operations performed over them. This facilitates a more restrictive admission scheme to the recycle pool. Only worthy instructions are kept. Such a restrictive admission scheme may reduce the storage needs of the recycle pool, especially for prefix-instructions with simple computations producing large intermediates.

The top-down matching strategy requires a more substantial change in the execution engine, namely the addition of a top-down evaluation of the query plan. The increased complexity of the matching mechanism can have a negative effect on its run-time performance.

Alternative 3: *optimistic sequence matching*. Similarly to the bottom-up sequence matching, all prefix-instructions in a path are kept, but the result data sets are stored selectively. The matching proceeds bottom-up with a special treatment of the 'phony' intermediates. If an instruction $A$ is matched against a phony intermediate, the matching is optimistically considered successful. When a dependent instruction $B$ is reached, there are two possibilities: either $B$ is successfully matched and an intermediate is reused, in which case the execution of $A$ will be skipped, or $B$ cannot be reused, in which case the interpreter backtracks and executes both $A$ and $B$.

Similarly to top-down matching, the optimistic matching requires some changes in the execution engine: the interpreter needs to be extended with backtrack functionality, and instruction matching has to take phony arguments into account. The last two approaches need cache admission algorithms to select intermediates for caching.

The precise trade-offs of all approaches in terms of complexity, performance, missed opportunities, and storage needs require much more in-depth investigation. Preliminary studies hint at a strong trade-off dependency on the workload characteristics, that is on factors such as the degree of partial overlaps (e.g., prefix and subsumption reuse), the ratio of aggregation operators processing large arguments and producing small intermediates, etc.

## 4. RECYCLE POOL MAINTENANCE

The goal for minimal run time overhead determined our preference to alternative 1: bottom-up sequence matching. Keeping all intermediates in a thread of execution allows for a prefix of an existing path to be reused by another query. This design choice impacts both the admission and eviction policies for recycle pool maintenance.

### 4.1 Recycle Pool: a Cache with Lineage

Caching is a technique used since the dawn of operating systems to manage their files, and of database systems to manage their page buffer pool. These techniques look at individual objects and mostly ignore their content. The recycler technique introduced here, however, is strongly focused on the semantics and history of all objects being managed. Matching instructions in the recycler pool calls for comparison of all arguments for validity. Since some arguments are results of earlier instructions, the bottom-up sequence matching depends on instruction dependencies and the way the admission and eviction policies treat them.

To illustrate 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 value 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 recycle pool even further.

Therefore, preserving instruction lineage by keeping all prefix instructions is crucial for successful bottom-up sequence matching and effective recycling. This means that both the admission and eviction policies have to respect instruction dependencies and keep whole threads of execution intact.

### 4.2 Admission Policies

The $recycleExit()$ 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 implementation supports the following admission policies:

—the KEEPALL policy is a baseline policy that keeps all instruction instances advised for recycling by the optimiser. 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 utilisation. 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.

Credits can be returned in two ways. 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 later, 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. Of course, when the underlying temporary results are dropped from the pool, the instruction gets a chance to re-enter it with new materialised results.

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 all its credits. Hence, the CREDIT admission provides almost full recycling opportunities, but with more economic resource use.

Preliminary experiments with admission policies based on filtering individual instructions on their properties, 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 Figure 2). 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.

### 4.3 Eviction Policies

Keeping a large number of intermediates around and checking for their usefulness at query run time at some point becomes a performance issue in itself. The main sources of overhead are the time taken for instruction matching and the space for storage of intermediates. To keep this overhead under control, the recycler routine *cleanCache* is called when needed to release resources (Algorithm 1, line 21). It uses the *eviction* policy to determine which intermediates to evict to make room for the ones more useful for the current load. This process is supported by the execution and reuse statistics of the instructions.

Despite similarities with the traditional cache replacement policies, there are several important differences when replacement algorithms are applied to caches of database query results. More precisely, maximising the cache hit ratio is not sufficient by itself since the items in the cache have in general different sizes and different costs in terms of resources consumed for their computation. Replacement algorithms that take these factors into account have been proposed in [Scheuermann et al. 1996] and other work. An important assumption in this case is that the retrieved sets are independent of each other.

As explained in Sec. 4.1, the design choice of bottom-up sequence matching requires the eviction policy to respect instruction dependencies. Therefore, the eviction policies first find the set $L$ of all instructions at the end of the execution

threads, called for convenience *leaf* instructions. In principle, the instructions from the query under execution, including the last instructions in each execution thread, are protected from eviction. They are predecessors of the current instruction and should be kept due to instruction dependencies[3].

The eviction policy picks one or more leaf instructions, such that they have the least expected loss for the system at large. We consider three factors to capture evidence for reuse from recycling: freshness, contribution to performance, and lifetime. The first factor is the time when an instruction has been used. The second is the benefit that the system has already gained from recycling the instruction. Finally, the lifetime reflects when an intermediate of an instruction has been created and allows for ageing of the benefit. Based on these factors, we propose the following three eviction policies.

—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 considers 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:

$$B(I) = Cost(I) * Weight(I). \tag{1}$$

The cost reflects the resources spent to compute the intermediate. The major resource we consider is CPU time, $Cost(I) = t_{CPU}(I)$. This choice is based on the encompassing optimization goal to minimise the total query and workload response time. The weight function reflects the number of reuses and their type.

$$Weight(I) = \begin{cases} k-1, & k > 1 \wedge global\ reuse(I) \\ 0.1, & k = 1 \vee local\ reuse(I) \end{cases} \tag{2}$$

where $k$ is the total number of references to the instruction $I$. Since intermediates that have been reused have already demonstrated a return-of-investment, they have a bigger weight than ones that have not. We also note that if an instruction has been reused only locally, there is no incentive to keep it in the pool beyond the query scope. Hence, we give a minimal weight of 0.1 to those instructions. In this way, an intermediate with relatively small cost, but a high reuse count, might be kept, and one with a high potential benefit(cost) that never 'materialises' in a reuse might be evicted.

—HISTORY POLICY (HP). The history policy is a modification of the benefit policy that takes both the total benefit and the lifetime of the instructions into account.

$$B(I) = \frac{Cost(I) * Weight(I)}{t_{cur} - t_{adm}}, \tag{3}$$

where $t_{cur}$ is the current time and $t_{adm}$ denotes the time of admission of the instruction to the pool. This allows for eviction of the oldest one among instructions with comparable total benefits. The policy is an adaptation of the cache replacement algorithm based on the *profit* performance metric proposed in

[3]An exception of this rule is made when intermediates of a single query fill the entire RP.

[Scheuermann et al. 1996]. It considers the $K$-latest references and uses a different cost metric (the number of buffer block reads). Since the recycler applies it per instruction, rather than per entire retrieved set as in Watchman, and to minimise the statistics administration, we chose to keep only the first reference moment instead of the $K$-latest.

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 and HISTORY policies corresponding to the resource limitation that triggers the eviction. If a single entry needs to be freed, the $BP_{ent}$ and $HP_{ent}$ policies pick the entry with the smallest benefit $B = min_{I \in L} B(I)$, where $L$ is the set of all leaf instructions, and the benefit is defined according to the policy specification.

To address a memory limitation, the $BP_{mem}$ and $HP_{mem}$ policies have to solve an optimization problem to find a set of the least beneficial instructions that would also release enough memory. The policies use the same algorithm differing only in the definition of the benefit. Let $M(I)$ be the memory taken to store the result of an instruction $I$, and $M_{req}$ be 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 minimises the total benefit $\sum_{I \in E} B(I)$. In practice, we solve the complementary problem, which is a version of the binary 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, usually called the *greedy* algorithm [Martello and Toth 1990]. It considers the items in a decreasing order of the profit per unit weight, where the profit is $B(I)$ and the weight is the size $M(I)$ of the intermediate result. Each item is put into the knapsack if it fits. To improve the worst-case performance, the solution is compared with an alternative given by the item of maximum profit. This puts an upper bound of the worst-case to be at most twice worse than the optimal solution.

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

## 5. INSTRUCTION SUBSUMPTION

Expression subsumption analysis is an effective way to improve query execution. During query optimization it is used to find common sub-expressions to avoid their repetitive evaluation. It is also used in finding alternatives in a collection of materialised views to obtain cheaper plans [Goldstein and Larson 2001]. Subsumption analysis can also be applied at *run time* in the recycler on the *instruction* level. For this we need to find instructions whose result set is a superset of what we intend to compute. To determine whether a result set is a super-set of another one requires knowledge of instruction semantics. Therefore, subsumption implementation in the recycler considers each instruction type individually. In the following we describe two types of subsumption implemented in the recycler: from a single instruction and from a collection of instructions.

### 5.1 Singleton Subsumption

During the search for a reusable result in the recycle pool we determine if a subsumption relationship holds with the target instruction. If several candidate intermediates are present, the recycler chooses one based on a cost model, and replaces the corresponding column operand of the target instruction with a reference to the result kept. The performance characteristics of MonetDB allow for a simple cost model based on the size of the operands.

The primary target for the recycler is to establish the subsumption relationship for select operations, because most queries start by reducing one or more base tables. Furthermore, an efficient relationship test calls for a minimal set of properties to be considered. This leads to focusing on range selections over ordered domains. More formally, consider the instructions $A$ and $B$ over an attribute with an ordered domain:

```
A := algebra.select(X,lb1,ub1);
B := algebra.select(Y,lb2,ub2);
```

The instruction $B$ can be subsumed from the intermediate result of instruction $A$ iff:

(1) $X \equiv Y$
(2) $[lb2, ub2] \subset [lb1, ub1]$

If there are no auxiliary indices then the cost of the selection is determined by the size of the operand, which means that we can safely compute $B$ by substituting the intermediate result of $A$ for the column operand $Y$.

```
B := algebra.select(A,lb2,ub2);
```

To illustrate, suppose the content of the recycle pool is as in Table I and a query comes with a selection predicate on ORDERDATE that is compiled into the following MAL instruction:

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

This instruction overlaps with two previously executed selections on ORDERDATE, whose intermediates are kept in the recycle pool 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 run-time support modifies the original instruction for the duration of interpretation into the following:

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

Upon completion of this modified instruction, the result is admitted to the recycle pool according to the prevalent admission policy and the target instruction is restored to permit re-evaluation of the query template.

A special case of select subsumption is also implemented for the SQL 'like' operator, i.e., for string matching with wild card patterns. It assumes exact match of the column argument of the operator. Typically, this occurs for operators over the same base table columns or those with exact match predecessors.

The second most important operation in the SQL query plans produced by MonetDB are the semijoins, because they implement the relational projection of the

candidate result set OIDS with all target columns of the result set. The semijoin is a form of subset selection and can be subsumed in an analogous way. It extracts all tuples of the first operand whose keys are in the domain of keys of the second operand. Assume

```
A := algebra.semijoin(X,V);
B := algebra.semijoin(Y,W);
```

be two semijoin instructions. The instruction $B$ can be subsumed from the intermediate result of instruction $A$ iff:

(1) $X \equiv Y$
(2) $W \subset V$

The instruction $B$ is computed by substituting the intermediate of $A$ for the first operand $Y$.

```
B := algebra.semijoin(A,W);
```

The condition stated can be proven using the set semantics of the relational operators. In particular, the semijoin operator $A = \{t | t \in X \wedge key(t) \in K(V)\}$ where $K(V)$ is the set of all key values of $V$. Since $W \subset V$ we can represent the set of tuples $V$ as a union of $W$ and $W' = V - W$. Hence, $A = \{t | t \in X \wedge key(t) \in K(W) \cup K(W')\}$ and clearly contains all result tuples of $B$.

To illustrate, consider the following content of the recycle pool:

```
X26 := algebra.select(X19,1996-07-01,1996-10-01,true,false);
X340 := algebra.semijoin(X2, X26);
X369 := algebra.select(X19,1996-08-01,1996-09-01,true,false);
```

The variable X369 is a subset of X26, computed in this case by selection subsumption. Then the following semijoin operator:

```
X370 := algebra.semijoin(X2, X369);
```

can be subsumed from the intermediate result X340 of the matching semijoin operator. In particular, the variable X2 can safely be replaced by X340 in the modified instruction.

### 5.2 Combined Subsumption

Re-using a single instruction can be implemented efficiently, but does not exploit all possibilities of a cache of intermediates. Instead, a more complex analysis over sets of instructions is implemented in the recycler.

For example let the recycle pool contain three selections over a column $A$.

```
X1 := algebra.select(A,3,7)
X2 := algebra.select(A,5,15)
X3 := algebra.select(A,6,40)
```

Furthermore, assume that a new selection should be computed over the range of values $[4, 8]$. Besides the regular computation over the column $A$, it is possible to compute it by subsuming the union of intermediate results $(X1, X2)$ covering the range $[3, 15]$, or the combination of intermediates $(X1, X3)$ covering the range $[3, 40]$. In general, the subsumption can be performed over any collection of intermediates that combined cover the target instruction range.

**Algorithm 2** Combined Subsumption

```
1: Input: MAL instruction I
2: Global variable to recycle pool RP
3: Sol ← I
4: R ← ∅
5: P1 ← ∅
6: for all X ∈ RP do
7:     if overlap(range(I), range(X)) then
8:         R ← R ∪ {X}
9:         P1 ← P1 ∪ {{X}}
10: for N = 1 to | R | −1 do
11:     P2 ← ∅
12:     for all S ∈ P1 do
13:         for all X ∈ R do
14:             if X ∉ S ∧ overlap(range(S), range(X)) then
15:                 U ← S ∪ {X}
16:                 if cost(U) < cost(Sol) then
17:                     if range(I) ⊂ range(U) then        ▷ A solution is found
18:                         Sol ← U
19:                     else                               ▷ A partial solution is found
20:                         P2 ← P2 ∪ {U}
21:     P1 ← P2
22: return Sol
```

The decision whether to apply combined subsumption and which combination to use is based on a cost model. Suppose $A$ is a column, $X_i$ are intermediates derived from $A$, and $C$ denotes the cost of an operator. The subsumption from a combination $S = \cup_{i=1}^{n} X_i$ of $n$ intermediates is more efficient than regular computation iff $C(A) > C(S)$. The cost of the combined solution is composed from the costs of piece-wise executions and an overhead, $C(S) = \sum_{i=1}^{n} C(X_i) + ov$. An example of overhead is the time to run the subsumption algorithm itself. When subsumption is possible from several combinations, the same cost model is applied to choose the most efficient solution.

Our subsumption analysis algorithm starts with finding the set $R$ of all instructions in the recycle pool whose result set overlaps with the target instruction $I$. Then, it finds the subsets $S \subset R$ of instructions that are sufficient to compute $I$. Although each of the subsets $S$ can be used for computation of $I$, the recycler picks the combination with the minimal estimated cost according to the cost model.

Formally, the problem boils down to constructing all possible subsets of the set $R$, checking whether they are possible solutions and finding the one with minimal cost. The problem has exponential complexity $O(2^k)$ on the number $k$ of instructions in the set, $k = |R|$.

We implemented a combined subsumption algorithm based on dynamic programming, illustrated in Algorithm 2. The main idea is to sequentially construct combinations of $2, 3, \ldots, k$ intermediates and cut the less promising partial solutions based on their estimated cost. Each iteration of the loop on the number of com-

ponents $N$ (lines 10-21) processes the set $P1$ of partial solutions of size $N$ and constructs a set $P2$ of partial solutions of size $N+1$. The solution with the lowest cost found so far is stored in the variable $Sol$. The algorithm complexity is reduced by early cutting of partial solutions with estimated cost higher than the cost of $Sol$ (line 16). In the case of a selection operator, we use a simple cost model considering only the size of the operand $C(X_i) = Sz(X_i)$.

In general this type of subsumption is expected to bring substantial performance benefits for queries with relatively small selectivities over large databases. Therefore, we evaluate the combined subsumption algorithm using the SkyServer application. Our micro-benchmark experiments show good scalability of the algorithm with overhead under 0.5ms for small $k, k < 10$. Detailed results are presented in Section 8.3.

## 6. RECYCLING WITH UPDATES

The MonetDB recycler has been designed for read-only database applications. This limitation was justified by the characteristics of the workload of scientific data warehouses with prevailing analytical queries, a well-defined small set of web-inspired SQL query templates, and rare periodical bulk updates. With a little care, however, the technique can also be applied to a volatile database. The base line is to *monitor* the database state, to recognise when and where an update takes place, and to *react* accordingly by synchronising the affected part of the recycle pool. This calls for monitoring the DDL statements, e.g., creation or dropping of database objects, the DML statements, e.g., inserts, deletes and updates, and the transaction boundaries to cope with transaction commit and abort.

In the remainder of this section, we illustrate the solutions to be effectuated in the context of MonetDB. Other systems may require different technical solutions.

### 6.1 Design Space for Update Recycling

The synchronisation of the recycle pool with the updated data state has three important design dimensions: what, when and how. The choices made along these axes are not independent, they may affect the possible alternatives in other dimensions. However, the choices are similar to those encountered in e.g., view and index maintenance, as we will elaborate in the following section.

First we consider *when* the synchronisation can take place. An immediate synchronisation approach keeps the system in a consistent state. However, it also adds a potentially high overhead to the performance of the DML statements and it has, in general, a small guarantee for a positive return of investment since the synchronised intermediates in the recycle pool may be evicted before they have been reused.

The transaction commit point also provides a natural moment to synchronise the recycle pool. The MonetDB/SQL transaction scheme is based on building delta tables for insert/deletes. At transaction commit they are checked for conflicts using an optimistic concurrency control scheme and, subsequently, used to update the base table. The disadvantage of this scheme aligns with the immediate synchronisation approach. It delays transaction response time and the work invested might be in vain.

Therefore, we consider approaches to postpone the hard work using a decoupling from the transaction commit. The recycler only marks the intermediates as being invalid as soon as an underlying persistent table is updated. In this way a relatively small overhead is added to the DML statements. A separate recycler synchronisation thread can take care of the synchronisation steps using the deltas. The last opportunity to reconcile the recycle pool is to postpone the work until an attempt is made to re-use stale information. This approach avoids any unnecessary work on intermediates that are not reused and, hence does not waste resources.

The second dimension of the synchronisation design is *granularity* or *what part* of the recycle pool content should be affected. Possible solutions are the entire pool, complete relational tables, individual columns, or selective operations. The granularity of recycle pool synchronisation has influence on its effectiveness and cost. A naive solution is to clean the entire recycle pool upon a DML statement. Obviously, such an approach can lead to a potentially large loss of benefits from intermediates being thrown out that were not actually affected by the update. The main advantage is its simplicity and the low overhead. Its performance would not be worse than MonetDB without the recycler being active. A refinement is to decrease the grain size, e.g. using a transaction time stamp kept with each relational table and making sure that all its snapshots in the recycle pool carry the same stamp. The drawback, however, is that a small local update may invalidate all the work already done which was potentially valuable for reuse.

In the column-store MonetDB it is straightforward to detect which columns and their derivatives are affected by an update, and to synchronise on column level. Finally, on the other extreme is to consider individual MAL operations only. For example, a select operation over a base table is a good starting point. Given the insert/delete deltas for each column, we can simply check for non-overlap with the operation. If none of the newly added tuples qualify and the intermediate does not contain any of the deleted ones, then the intermediate can be retained. If the insertion overlaps with the fragment in the pool, we can add the missing tuples and remove the remainder of the plan since propagation of the insertion may require doing complex operations.

The third important design question is *how* exactly to synchronise the recycle pool. We envision two basic mechanisms for this: *invalidation* and *propagation*. Evicting the invalid intermediates is simple, fast and releases memory in the pool. Propagation of updates is in general a more complex and costly mechanism, which preserves, at least partially, the opportunities for reuse. The invalidation of intermediates is generic, while the propagation mechanism is specific for the operator type and the type of the update.

Consider an expensive intermediate computed over a large column to which a few tuples have been appended. Propagation can be performed by executing the original operator over the newly appended tuples only and then appending the result to the intermediate retained. Such propagation can be much cheaper than re-computing over the original large attribute.

The choice of synchronisation mechanism requires careful consideration to be made separately. It also strongly depends on the workload mix, which makes general guidelines hard to quantify.

### 6.2 Related Work on Update Propagation

The problem of keeping the content of the recycle pool synchronised upon updates of base tables is closely related to the problem of materialised view maintenance, which has been extensively studied in the database community. The invalidation approach corresponds to re-computing a view from scratch, while propagation algorithms correspond to incremental view maintenance using differential algorithms [Blakeley et al. 1986; Griffin and Libkin 1995; Ross et al. 1996; Luo and Yu 2008].

There are a few aspects in which recycle pool maintenance differs from view maintenance. First, and foremost, materialised views are part of the database schema and managed by the DBA. Conversely, the recycled intermediates are automatically collected and reused at run time. Hence, their maintenance needs to be integrated with the run-time support of the recycler and they do not require manual intervention. Second, the recycle pool in MonetDB is kept on the server where the base tables reside. This restriction does not apply to materialised views, which can be spread over a complete distributed system. Finally, the finer granularity of recycling per relational algebra operator allows for instance partial propagation of the updates to operators for which it is cost efficient, and invalidation for the remainder of a cached plan.

### 6.3 Outlook to a Solution

The analysis of the design space shows that the best synchronisation mechanism for a given situation heavily depends on the trade-off between the re-computation and propagation costs. In turn, these costs depend on the update type, the operation involved, and the resource claims of the argument columns. In general, the solutions can not be separated from a clear vision of the workload characteristics.

The synchronisation of DDL statements is straightforward. Creation of new objects does not affect the content of the recycle pool. Deletion of objects (drop table, drop index) requires invalidation of all dependent intermediates. It does not necessarily require a scan through the recycle pool, because the eviction policy will in due course drop them. The explicit scan through the recycle pool ensures that we free up resources quickly.

The MonetDB DML statements require more complex synchronisation. We consider here only inserts and deletes. Updates are implemented as a combination of both. In general the propagation is a form of incremental computation well studied in the database literature. To propagate a delta $\delta$ through an operator $P$ first the operator is executed over $\delta$. The result $P(\delta)$ is used for two purposes: (1) to update the intermediate result of $P$, and (2) as input delta that has to be propagated through all the operators dependent of $P$ (that use $P$'s result as input).

If the result $P(\delta)$ is empty the partial result of $P$ can be retained without change. The propagation in this case is just a marking of the intermediate as valid again.

If the result $P(\delta)$ is not empty it is used to modify the stale intermediate, where the precise modification depends on the type of update and the operator type. In the following we describe the propagation mechanisms for different situations.

—*Selection*. The propagation first applies the select operator to the insert delta $\delta^+$, delete delta $\delta^-$, or the combination of both. The computed delta $P(\delta^+)$ is appended to the intermediate result, the delta $P(\delta^-)$ is deleted. The propagation

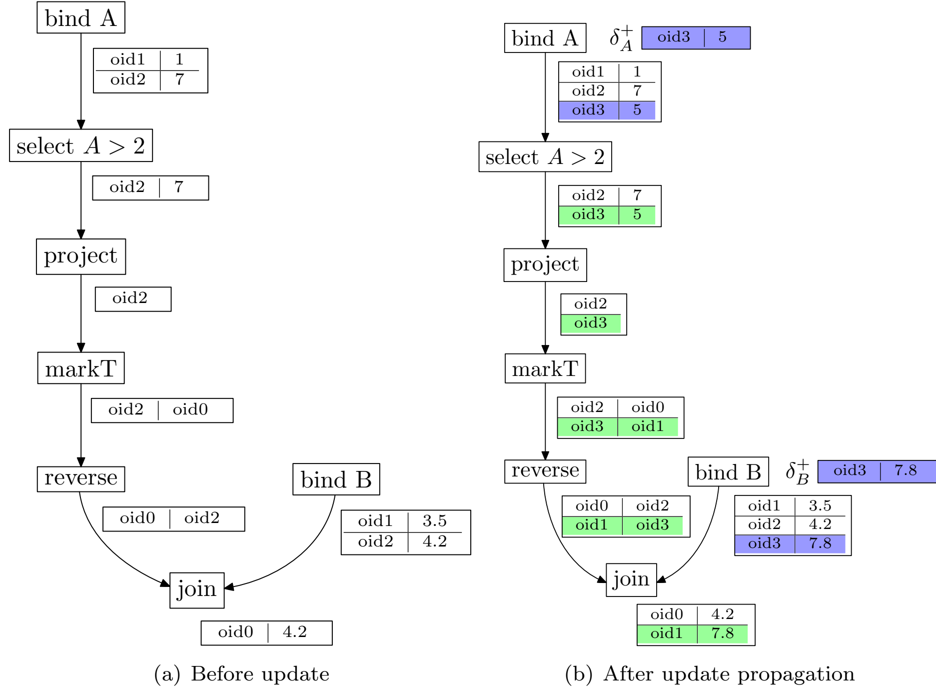



Fig. 3. Excerpt of a cached MAL plan.

of updates through selection is performed by deleting the delta $P(\delta^-)$, followed by appending $P(\delta^+)$.
After the stale selection intermediate is refreshed, the recycler either invalidates the remainder of the execution thread or continues with delta-wise propagation through dependent operators using the freshly computed $P(\delta^+)$ and $P(\delta^-)$.

—*Projection.* A projection operation extracts the head (row identifier) of a column. The propagation step is to simply execute the projection over the delta and append, resp. delete, the result from the intermediate.

—*Reverse.* A reverse operation exchanges the places of the head and tail of a binary table (BAT). The propagation step is to simply execute the reverse over the delta and append, resp. delete, the result from the intermediate.

—*MarkT.* The markT operation generates a new dense sequence of row identifiers. The propagation of $\delta^+$ requires modified execution of the MarkT operation over $\delta^+$ where the sequence starts with the next row identifier after the last one in the stale intermediate. The propagation of $\delta^-$ is more complex since deletions create holes in the sequence of row IDs. The result delta can be computed as a compensatory semijoin operator of the MarkT intermediate result and $\delta^-$. However, the holes created by $\delta^-$ propagation in the dense sequence of identifiers complicate further propagation and we expect it to be inefficient.

—*Join.* The propagation of join takes into account the possible updates in both operands. Suppose join has two operands $R$ and $S$. The propagation of inserts

computes $\delta_R^+ \bowtie S$, $R \bowtie \delta_S^+$, and $\delta_R^+ \bowtie \delta_S^+$, and appends the results to the intermediate. Similarly, the propagation of deletes computes $\delta_R^- \bowtie S$, $R \bowtie \delta_S^-$, and $\delta_R^- \bowtie \delta_S^-$, and deletes the results from the intermediate. For the case of mixed inserts and deletes we follow the rules for differential join re-evaluation described in Blakeley et al. [1986].

Figure 3a shows an example of MAL plan instructions and their respective intermediates cached in the recycle pool. Figure 3b illustrates the state of the intermediates after update propagation triggered by the addition of a tuple to the base table. At the end of the insert transaction the $\delta^+$ is merged with the base table, updating the intermediates of the two bind operators. The remaining instructions in the cache plan are marked as invalid. The first attempt to reuse the select operator triggers the insert propagation through the remainder of the cached plan.

### 6.4 Implementation Status

The MonetDB implementation used during the experiments poses some limitations on the solution space. Since the recycler works at the MAL level, it should be able to recognise all DDL and DML statements and the transaction boundaries[4].

The DDL and DML statements are monitored by the recycler optimizer. When an update is detected, the optimizer injects calls to the synchronisation routine *RecycleReset*. The synchronisation is based on immediate invalidation of all intermediates that have been affected by the change. The invalidation is performed column-wise. Insertion and deletion of rows affect all cached columns of the changed table, but updates invalidate only the columns directly affected. The advantages of this choice are its low-cost and immediate freeing of memory resources. As a future work we intend to investigate the trade-offs between this approach and the update propagation, which is expected to have advantages for small changes over expensive intermediates.

## 7. TPC-H EVALUATION

To gain a deep understanding of the recycler mechanisms and its effect on the query performance, we conduct extensive experiments with the TPC-H Decision Support benchmark [Transaction Processing Performance Council 2008]. The experiments are run against a database of scale factor 1 (SF1), i.e., of size approximately 1 GB.

All experiments are run on dual Quad Core AMD Opteron 2GHz processors with 8 GB RAM and 1 TB of disk space. All times reported are measured in an experimental environment prepared in the following way: first, we execute a subset of the query batch with an instance of each query template. This ensures that all persistent BATs retrieved in the batch are touched and fill memory with hot data. Next, we empty the recycle pool, which triggers queries to fill it before they can benefit from reuse of intermediates. The purpose of this preparation step is to factor out the IO costs and better illustrate the pure effect of the recycler.

First, we analyse 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

[4]The experiments were run against the Feb 2010 release. Transaction boundaries were not yet visible at MAL level, which called for a focus on the immediate synchronisation.

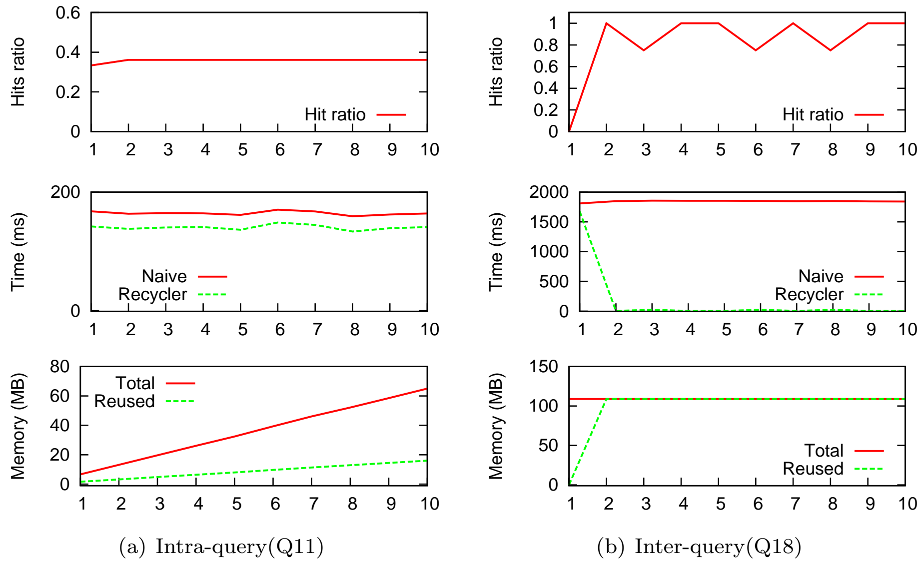
| Query | Instructions | | | Time (s) | | | |
|---|---|---|---|---|---|---|---|
| | # | Intra % | Inter % | Total | Savings | | |
| | | | | | Pot. | Local | Glob. |
| Q1 | 36 | 2.8 | 0 | 5.72 | 3.54 | 0.30 | 0 |
| Q2 | 106 | 0.9 | 2.8 | 0.22 | 0.22 | 0 | 0.07 |
| Q3 | 39 | 0 | 5.1 | 2.61 | 2.40 | 0 | 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 | 18 | 0 | 0 | 0.21 | 0.21 | 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 II. Characteristics of TCP-H queries

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 II shows the commonality characteristics of the TPC-H queries. The *Instructions/#* column contains the total number of instructions marked by the recycler optimiser 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. Given the algebraic framework, this table highlights the opportunities for reuse independent of the technique deployed.

The right side of Table II shows the total execution times of the queries against MonetDB together with time savings from the recycler. We summarise the potential savings, i.e., the total time spent in monitored instructions, the realised intra-query savings and those from a single inter-query reuse. Manual inspection of the query




(a) Intra-query(Q11)

(b) Inter-query(Q18)

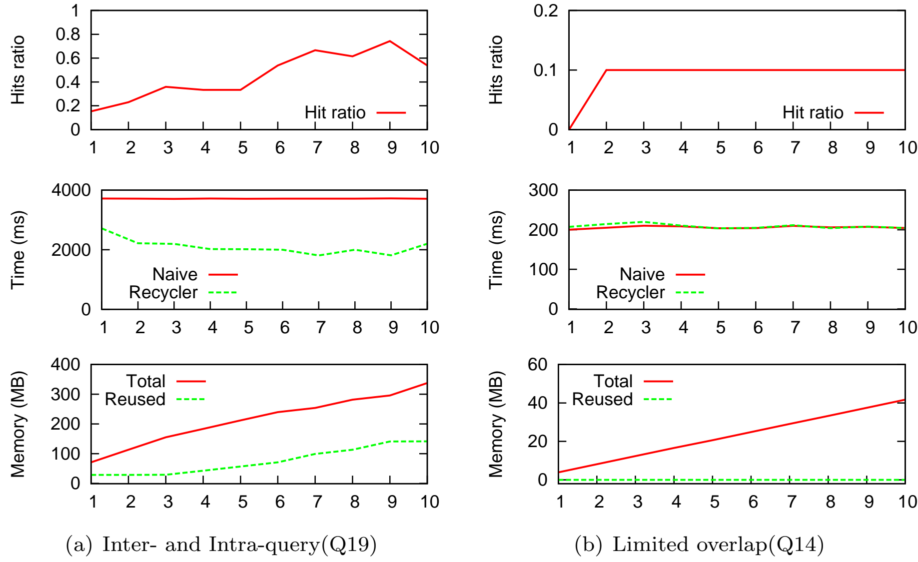
Fig. 4. Recycler effect with different type of query commonalities

execution traces demonstrates significant time savings, such as for Q4, Q18, and Q21, which confirms that the instructions reused carry a high cost. For other queries, such as Q16, time savings are minimal despite the high percentage of instructions reused. These instructions are just too cheap.

## 7.1 Micro- Benchmarks

Next, we examine the recycler performance and resource utilisation over four query workloads: with prevailing local, prevailing global, or mixed commonalities, as well as queries that do not exploit overlaps. We choose queries typical for each of the aforementioned groups. In this micro-benchmark we execute 10 instances of each query generated with the TPC-H query generator. To better illustrate the recycler mechanics, the admission policy is KEEPALL and there are no resource limitations; no eviction policy interferes with the results.

Query Q11 contains substantial intra-query commonality where a large part of the sub-query is shared with the outer query block. 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 Figure 4a. 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.




(a) Inter- and Intra-query(Q19)

(b) Limited overlap(Q14)

Fig. 5. Recycler effect with different type of query commonalities

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 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 Figure 4b 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 a 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 materialised view storing the groups of the LINEITEM table rows together with the computed aggregates. We experimented with a version of query Q18 using a materialised view and observed performance comparable with the recycler, namely 2ms per query instance using the same experimental settings. The recycler brings both performance and 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.

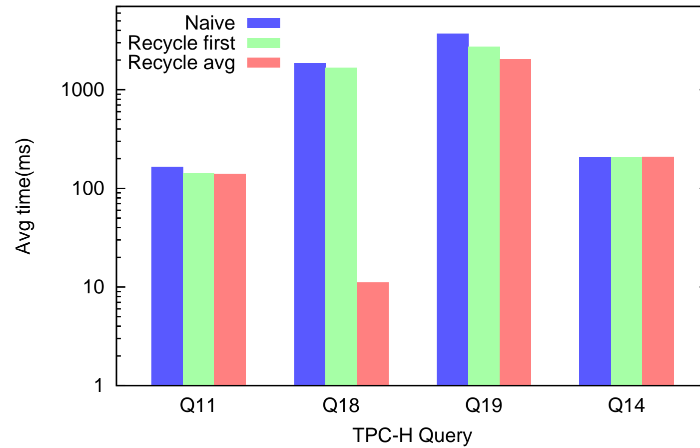



Fig. 6. Recycler effect on performance

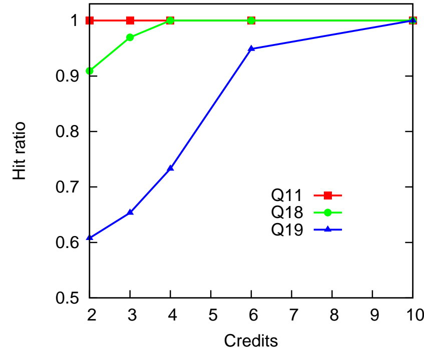
Query Q19 has a mixture of intra- and inter-query overlaps. It contains three sub-queries with a number of predicates overlapping both inside a query and among different instances. In the query profile in Figure 5a we observe some hits in recycle pool and time improvement in the first instance due to 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.

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 invocations have different parameters and the overlap is limited to two cheap auxiliary instructions. Hence, the query demonstrates rather the potential overhead of the recycler for storing and matching intermediates. In the query profile in Figure 5b we observe a small hit ratio caused by the auxiliary instructions. Each invocation adds 18 instructions to the recycle pool and allocates 4 MB for the intermediates without amortising this resource investment in the form of performance improvements. We observe an average overhead of 3ms per query invocation due to the recycler extension. The average performance improvements for the 10-instance benchmarks of the above queries are illustrated in Figure 6.

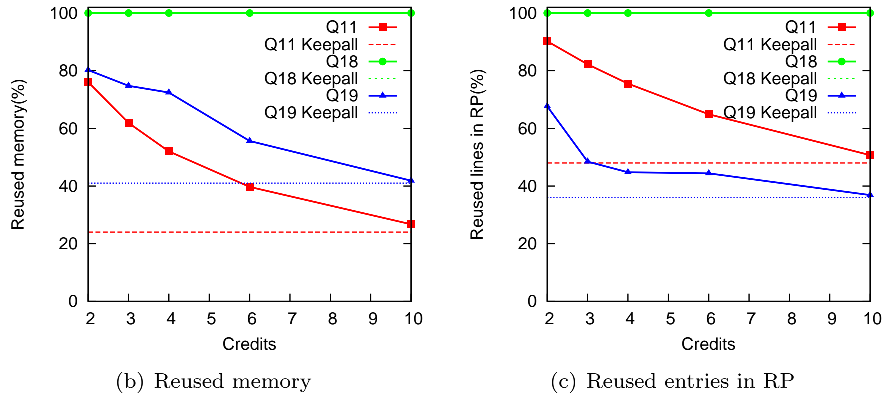
The memory profiles show that both queries with partial overlap and without overlap may accumulate intermediates that are not reused. Having observed this danger, we turn our attention to the admission policies that can prevent waste of resources by early filtering of intermediates at the admission to the recycle pool.

## 7.2 Evaluation of Admission Policies

In this section we evaluate the CREDIT admission policy (CRD) in terms of resource utilisation 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 7 shows the hit ratio to the base line(a), the percentage of reused memory(b), and reused recycle pool entries(c), as the number of credits increases. The experiments were run in unlimited resource settings.




(a) Hit ratio to Keepall policy




(b) Reused memory

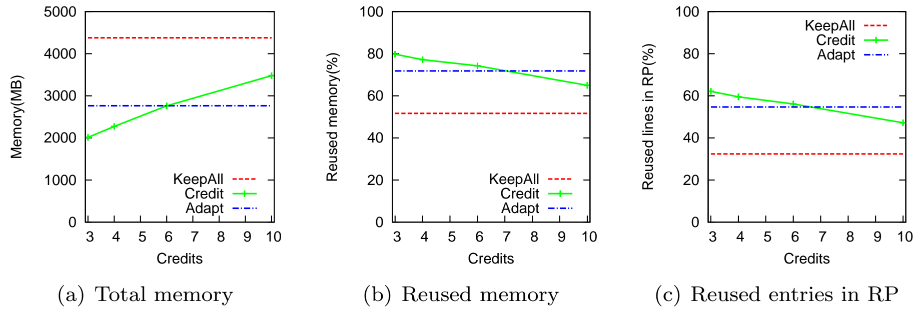
(c) Reused entries in RP

Fig. 7. Effect of credit parameter to resource utilisation and RP hits

Credits do not affect the hit ratio for intra-query commonalities (Q11), since local reuses return the credit immediately to the source instruction. However, having a small number of credits successfully limits the admission of instructions that are not reused and substantially improves memory and recycle pool entry utilisation.

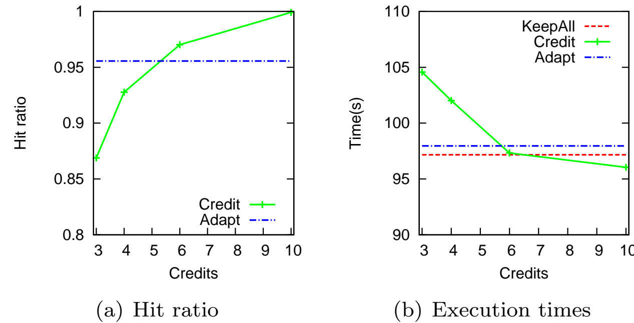
The number of credits affects the hit ratio for inter-query commonalities (Q18 and Q19). Having a small number of credits improves the resource utilisation (Q19), but also prevents keeping and reusing some of the overlapping intermediates. As the number of credits increases, the hit ratio improves, simultaneously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

The analysis of advantages and drawbacks of the originally proposed admission policies inspired us to look for a solution that achieves resource efficiency without losing performance. We developed an adaptive version of the credit admission policy that adjusts the number of credits given to an instruction based on statistics collected. The *adaptive credit policy* (ADAPT) starts in a similar way giving a number of credits $k$ to all instructions marked for recycling. However, after $k$ query template invocations, the instructions that have been reused at least once




(a) Total memory (b) Reused memory (c) Reused entries in RP

Fig. 8. Effect of admission policies on resource utilisation




(a) Hit ratio (b) Execution times

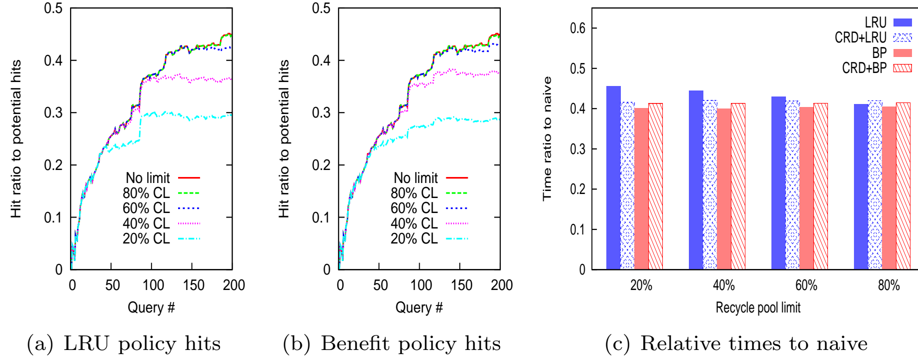
Fig. 9. Effect of admission policies on performance

get unlimited number of credits, while all others exhaust their credits and are not admitted to the recycle pool anymore.

To evaluate the overall effect of the admission policies we create a mixed workload with both local and global overlaps. We select 10 TPC-H queries (4,7,8,11,12, 16,18,19,21, and 22) with relatively large overlaps to highlight how well the admission policies recognise instruction categories in terms of reuse. We create a batch of 200 queries by mixing 20 instances of each.

Figure 8 shows that the adaptive credit admission ($k = 3$) substantially improves resource utilisation in comparison to the KEEPALL policy. The recycle pool needs 35% less memory (Fig. 8a), while the percentage of the reused memory increases from 51 to to 71 (Fig. 8b).

Figure 9 illustrates the effect of the admission policies on performance. The ADAPT policy achieves a high hit ratio of 95% in comparison to the hits of the KEEPALL policy, and an absolute execution time very close to the best one achieved by the 10-credit admission, followed closely by the KEEPALL admission. The ADAPT policy avoids the performance degradation of the CREDIT policy with a small number of credits, and is more resource economical than the CREDIT policy with a high number of credits.




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

Fig. 10. Eviction policies in limited recycle pool

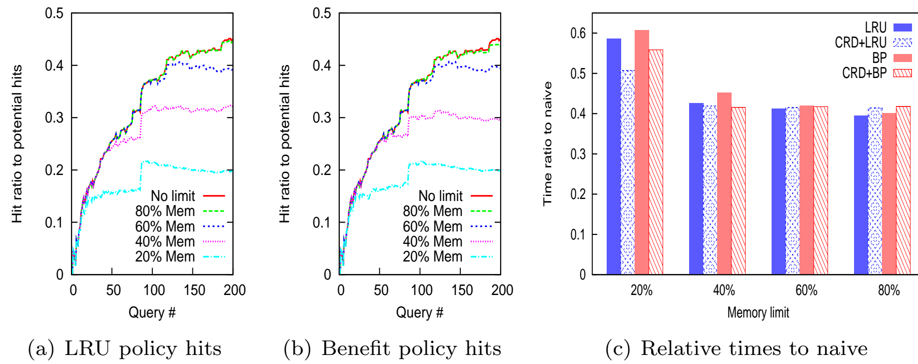
### 7.3 Evaluation of Eviction Policies

The eviction policy has a more diverse influence and strongly depends on 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 and HISTORY policies are 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 eviction policy. If the policy cannot distinguish instructions with potential reuse and throws them out, this directly leads to a higher number of recycle pool misses and reduced improvement of performance.

We evaluate the eviction policies using the mixed workload of 200 queries described in Section 7.2. Having relatively large overlaps among the queries increases the contention between instructions, i.e., a situation where a bad choice of the evicted instruction becomes more noticeable. 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 eviction policies with resources limited to a percentage of the total resources. We consider two major resources: memory taken by the intermediates, and number of recycle pool entries which affects the instruction matching time.

Figure 10 shows the effect of the LRU and BENEFIT eviction 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 (Figure 10c). Although the BENEFIT policy (BP) shows lower hit ratio than LRU in some cases, it succeeds better in distinguishing and keeping weighty intermediates in the pool. For all




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

Fig. 11. Eviction policies in limited memory

limits BP achieves the best performance running for 40% of the total time of the NAIVE strategy.

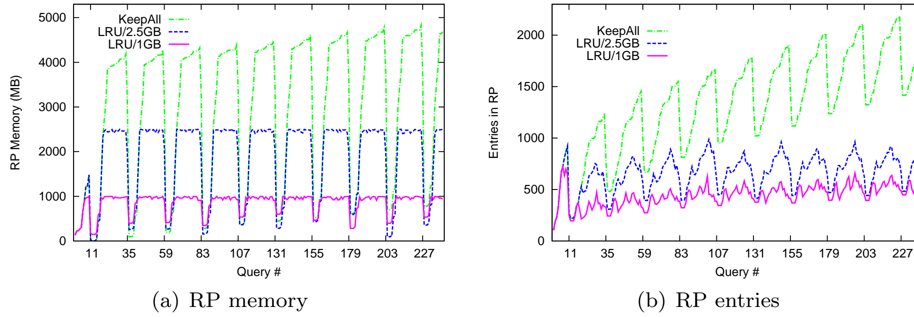
The effect of credit admission policy is two-fold. When combined with BP it leads to a small loss of performance due to some hit misses. The combination of CRD and LRU improves the LRU, especially when a severe limitation is imposed. The CRD admission achieves early filtering of instructions that are not reused. It manages to keep old reused instructions longer in the recycle pool in situations of high resource contention.

Figure 11 shows the policies behaviour in limited memory. This limitation affects both the hit ratio and the processing time more substantially than the recycle pool 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 prove to be more efficient.

We also performed experiments with the HISTORY eviction policy. The results obtained showed a minor variation from the BENEFIT policy and are not presented here. We expect a more substantial difference between the two versions for changing workloads.

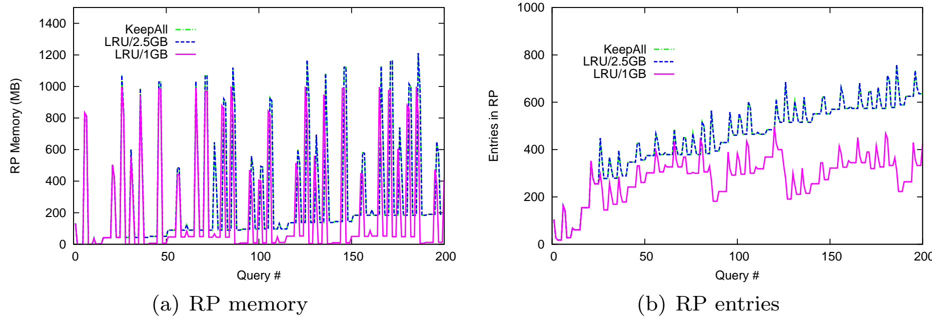
In general, we observed a relatively small impact of the kind of eviction policy used. We attribute this to the fact that the eviction policy is applied on instruction level, as opposed to application to the final result sets as in [Scheuermann et al. 1996]. Furthermore, due to the instruction dependencies, the policy only considers a relatively small number of leaf instructions. This specific application often leads to situations where a large new intermediate evicts a set of small leaf instructions, which is almost identical for different eviction policies.

We expect that the choice of eviction policy has a bigger influence with other design alternatives (Section 3.4), for instance if only selective intermediates are admitted to the pool.




(a) RP memory (b) RP entries

Fig. 12. Recycling in presence of updates, K=20




(a) RP memory (b) RP entries

Fig. 13. Recycling in presence of updates, K=1

### 7.4 Evaluation of Recycling with Updates

To study the behaviour of recycling in a volatile environment, we modify the query batch of Section 7.2 by injecting update statements with different frequencies. The updates are generated based on the specification of the TPC-H refresh functions. Each block of updates inserts a set of new customer orders, which effectively adds 7-8 rows into ORDERS and 25-56 rows into LINEITEM tables. Similarly, it deletes a set of old orders from both tables. One update block is put in the middle of each block of $K$ queries, for $K = 1, 10, 20$, and 50.

We study recycling with immediate invalidation of the intermediates affected by the updates. We evaluate both unlimited recycling, where the total size of the pool grows to almost 5 GB, and recycling with LRU eviction policy and a pool of size of 2.5 and 1 GB, respectively.

Figure 12 illustrates the change of the recycle pool memory(a) and number of entries(b) as the query batch for $K = 20$ (e.g., with 10 update blocks) executes. The labels on the X-axis denote the first statement in each update block. In all strategies we observe invalidation of a large part of the recycle pool during the update blocks. The invalidation affects the intermediates derived from ORDERS and LINEITEM tables. Note that since some queries, such as TPC-H 11 and 16, do not touch the updated tables, their intermediates are not affected by the invalida-

tion. The strategies with limited memory also evict some instructions during the query execution in order to fit the recycle pool size limitation. Hence the update invalidation affects a smaller part of the recycle pool content.

Figure 13 illustrates a situation with a highly volatile database where each query is followed by a block of four update statements (K=1). We observe continuous alternation of the recycle pool content where the intermediates added by a query are immediately thrown out by the subsequent update block.

Each time we clean the recycle pool, its contribution to improved performance in subsequent queries drops significantly. As such, the system falls back to the performance of vanilla MonetDB, i.e., without the recycler. The overhead of managing the recycle pool itself is negligible compared to the instructions being executed.

## 8. SKYSERVER EVALUATION

In this section we demonstrate the recycler in the context of the SkyServer project [SkyServer 2008]. SkyServer is a sizable 4 TB scientific database with 91 tables, 51 views, and 203 persistent module functions. The SkyServer database is publicly available, but its size and complexity is sufficiently complex that the MonetDB implementation is the sole known complete alternative implementation. For our experiments we extracted a smaller version from both the database and the query log. The test database deployed in the experiments is a 100 GB subset of SkyServer Data Release 4 (DR4).

The times reported are again measured in an experimental environment prepared with warming-up queries and emptied recycle pool.

### 8.1 Workload Characteristics

We prepare two query batches of 100 and 500 queries against DR4 randomly picked from the real life query log from January 2008. A manual inspection of the random set confirms the observations reported in [Ivanova et al. 2007] 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 (>60 %) in both batches:

```
SELECT p.objID, p.run, p.rerun, p.camcol, p.field, p.obj, ...
FROM fGetNearbyObjEq(195,2.5,0.5) n, PhotoPrimary p
WHERE n.objID = p.objID
LIMIT 1;
```

The query accesses the catalogue table with photometric properties of sky objects through the view PhotoPrimary. The object filtering is based on sky location. The table-valued spatial function fGetNearbyObjEq extracts the objects in a circular area specified by the equatorial coordinates and size parameters. 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. The instances of the query are almost identical: there are two different, but overlapping, sets of parameter values of the spatial search function. Furthermore, there is a difference in the

| 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 |
| MarkT | 33 | 0 | 1 | 24 | 439 | 1 |
| ... | ... | ... | ... | ... | ... | ... |
| Total | 258 | 1500 | | 170 | 5711 | |

Table III. Characteristics of recycle pool after DR4 query batch

projected columns, where a value is a function of the name of the server where the original query was executed. As a result, the recycler reuses the majority of the intermediates of this query template.

Approximately 36 % of the queries retrieve information from the self-descriptive documentation tables of the SkyServer web site. Those tables are relatively small and the queries over them are very fast. Finally, a small percentage (appr. 2%) of the batch are point queries retrieving all attributes of an object given its unique ID, e.g.:

```
SELECT * FROM ELRedshift WHERE specObjId=0x0559cf6177c00000;
```

When the 100-query batch runs with the KEEPALL admission and unlimited storage, the recycler monitors the execution of 5969 relational algebra instructions, constituting approximately 15% of all instructions being executed. 5711, or 95.6% of these monitored instructions are successfully reused.

Table III breaks down the content of the recycle pool at the end of the batched 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 for several intermediates of size smaller than 1 KB each, all 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.

### 8.2 Workload Performance

Figure 14 illustrates the total time for the 100 query batch with and without recycler intervention. The NAIVE strategy denotes regular execution without recycling. We run 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.

The SkyServer is a read only database. Nevertheless, to assess the impact of updates against it, we split the 100 query batch into shorter sequences of 25 and 50 queries and run 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

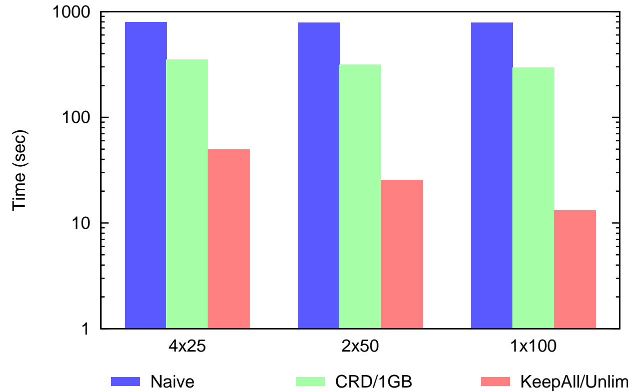



Fig. 14. Recycler effect on SkyServer query batch

performance. Still, the total time of CRD/LRU/1GB is 296 sec, or approximately 38% of the naive strategy time.

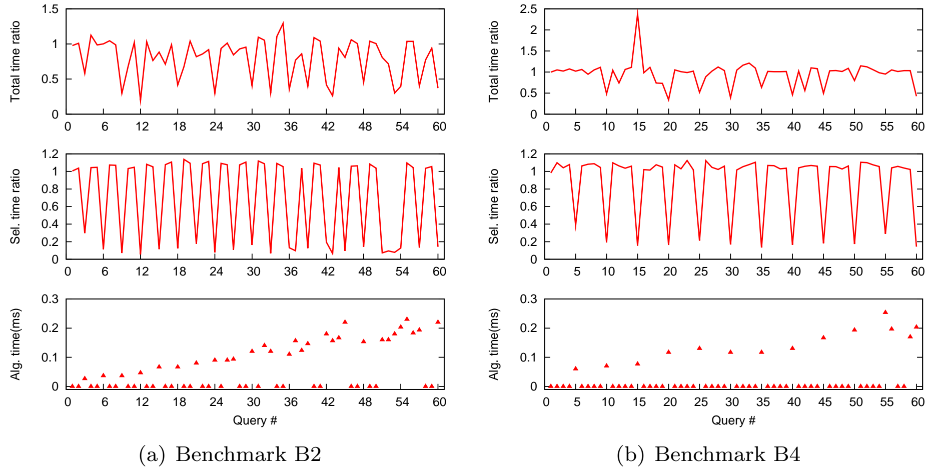
The batches of 4x25 and 2x50 queries show similar performance with a small overhead due to the loss of intermediates from the previous batch that have to be computed again. To verify the results we scale the experiment to 500 randomly selected queries. The times measured confirm the observations from the shorter batches: the NAIVE strategy runs for 4057sec, the KEEPALL/UNLIMITED achieves 17sec, and the CRD/LRU/1GB strategy takes 1433sec, i.e., approximately 35% of the NAIVE strategy time.

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

### 8.3 Evaluation of Combined Subsumption

We use the SkyServer application to prepare micro-benchmarks for evaluation of the combined subsumption feature of the recycler. The experiments run against a 75GB database of 10 million sky objects. We picked a common query pattern from the query log that performs spatial search of sky objects using their right ascension and declination coordinates, both attributes being floating point numbers. To generate the benchmarks we instantiate the query pattern with different parameter values that ensure combined subsumption. Starting with a *seed* query with selectivity factor $s$ over the right ascension attribute we generate a sequence of $k$ queries that together allow the seed query to be answered by a combined subsumption. The selectivity factor of the covering queries depends on the parameter $k$ and was set to $s(k) = 1.5 * s/(k-1)$ to provide overlap among the queries and full coverage of the seed query.

The micro-benchmark $B2$ of 60 queries is generated from 20 seed queries and parameter $k = 2$, i.e., each seed query can be answered by subsumption from at




(a) Benchmark B2 (b) Benchmark B4

Fig. 15. Performance of combined subsumption algorithm

least one combination of two queries. The micro-benchmark $B4$ is similarly created by setting $k = 4$ and using 12 seed queries. The selectivity factor $s$ in the seed queries was in both cases set to $s = 2\%$, or approximately 200K sky objects.

Figure 15 shows the performance figures of both micro-benchmarks. The uppermost diagram presents the ratio of total times of subsumed execution towards regular execution without recycling. The ratio is noticeably smaller than 1 for the majority of the seed queries (every 3rd one in $B2$ and every 5th one in $B4$) where the combined subsumption applies. Of course, when running the experiments in a complete system we also observe outliers when other factors influence the total execution time.

To isolate the effect of the execution environment we zoom into the execution times of the selection operator over the right ascension attribute. The middle diagram shows the ratio of the selection times with and without combined subsumption. We clearly can observe the time savings due to the combined subsumption in the seed queries. The average subsumed selection runs in about 20% of the time of the regular selection. It confirms a proper implementation of the algorithms in our framework.

The bottom diagram presents the absolute time in milliseconds spent in the combined subsumption algorithm. The time increases with the size of the cache and the number $k$ of the covering queries. We measured maximal time of 0.25ms per invocation for $k = 4$ and cache of 800 instructions and 0.4ms for $k = 9$, which indicates good scalability. For expensive operators of hundreds of milliseconds the algorithm overhead is negligible.

The experiments with batches with other selectivity $s$ and parameter $k$ showed similar results and are not presented here.

## 9. RELATED WORK

Recycling exploits the generic idea of storing and reusing the results of expensive computations. While traditional cache replacement policies focus on maximising the cache hit ratio, maintenance of caches of database query results have to also consider the items' different sizes and computational costs. Furthermore, recycling differs from file-caches in operating systems and page-caches in database system by exploitation of the operator semantics and their dependencies. We use an optimiser 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 materialised views and query caching [Agrawal et al. 2000; Chen and Roussopoulos 1994; Goldstein and Larson 2001; Mistry et al. 2001; Scheuermann et al. 1996; Tan et al. 2001; Zhou et al. 2007a]. Our approach differs from this large body of work in some or all of the following three aspects: self-organising behaviour without human intervention, integration with the DBMS software stack, and granularity of operation. The materialised views are often defined by the DBA with the help of workload analysers to improve system performance [Agrawal et al. 2000; Mistry et al. 2001]. The dynamic materialised views proposed in [Zhou et al. 2007a] materialise 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 query optimization [Chen and Roussopoulos 1994; Goldstein and Larson 2001; Tan et al. 2001]. 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 [Tan et al. 2001] recycling of intermediates is ensured by considering only the present. An overlap in an incoming query with the currently running queries triggers materialising common intermediates. This approach is beneficial in a multi-user scenario setting, but imposes temporal locality limitations on the overlapping queries.

Finally, recycling is a general technique that works at a finer level of granularity than materialised views and query caches. It keeps individual instruction results independently from the source of commonalities and the type of the entire query. DynaMat [Kotidis and Roussopoulos 2001] proposes dynamic management of a pool of materialised views in data warehouses. Similar 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 to 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 [Choi et al. 2003; Luo and Yu 2008].

In contrast, recycling considers intermediate results at the instruction level and it is a general technique that does not impose limitations on the query types.

[Scheuermann et al. 1996] proposed cache replacement and admission algorithms for retrieved query sets based on a profit metric that incorporates the reference rate, cost and size of the result set. The benefit policy in the recycler is similar to their profit metric with the following differences: we consider the total number of references, as opposed to the reference rate of the last-K references that incorporates ageing of sets. The algorithms are applied to independent query sets, while we need to take into account instruction dependencies, namely the eviction algorithm is applied only over leaf instructions.

Caching and reusing intermediates in the context of a pipeline execution engine is proposed in [Rao and Ross 1998]. The invariant parts of correlated subqueries are automatically recognised and the evaluation plan is restructured to keep and reuse the intermediate results. The application of this approach is limited to parts of correlated subqueries. Hence, the work does not tackle issues such as sharing and matching of intermediates among different queries and maintaining a common cache of intermediates.

Database caching [Bornhövd et al. 2004; Larson et al. 2004] 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 materialised views and thus is static with respect to the covered database sub-schema.

Our approach to share computations between queries also relates to multi-query optimization [Roy et al. 2000] and exploitation of similar sub-expressions [Zhou et al. 2007b]. Both techniques are applicable to queries that are known in advance and executed concurrently, such as query batches, sub-queries, and maintenance of materialised 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-priory knowledge about the workload.

The adaptive replication technique presented in [Ivanova et al. 2008] exploits the materialisation of selection intermediates to reorganise 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.

This manuscript extends the work presented in [Ivanova et al. 2009] in two major directions: enhanced predicate subsumption and recycling in volatile databases. The subsumption technique initially based only on a single intermediate result, has been enhanced to consider combinations of intermediates. The combined subsumption algorithm has been fully implemented in the code base and evaluated in the context of micro-benchmarks derived from the SkyServer application. Originally the recycling technique was developed, and showed to be efficient, for prevailing read-only workloads. In this work we discuss the problems of recycling in the presence of updates, describe the landscape of possible solutions, and evaluate an implementation based on immediate invalidation of cached intermediates affected by the updates. Finally, the detailed analysis of the initial admission policies led to

introduction of the adaptive credit admission policy that provides better utilisation of the recycle pool resources with a minimal loss of performance.

## 10. SUMMARY AND CONCLUSIONS

In this paper we have described a database architecture augmented with recycling intermediates from relational algebra programs, i.e., caching partial results. The approach is implemented as an extension to the MonetDB system, which differs from the main-stream database engines in its use of an operator-at-a-time execution paradigm. Our approach addresses the potential overhead incurred by this execution paradigm, which can lead to sizable intermediates being produced. These intermediates are used as a kind of dynamically materialised views described by relational algebra plan snippets, which turn them into a benefit without requiring (manual) DBA intervention or an a-priory workload analysis.

Unlike traditional file system caches, the recycler policies respect the inter-operator dependencies, which leads to a much more effective reuse of long instruction sequences in template based query sessions, e.g., web applications. The recycling policies studied cover both an extension to the basic LRU scheme for relational operations and ones driven by an economic cost principle. Extensive experimentation based on a full-fledged implementation shows that the MonetDB software architecture is well suited to be extended with such a targeted optimization goal.

The validity of the approach is demonstrated using the SkyServer real-life query log. Even in this well-designed application, recycling partial results can lead to significant gains. Primarily, because it can adapt more easily to the re-use of expensive query plan parts.

The results obtained indicate several areas for further exploration. Within the context of operator-at-a-time execution, e.g. MonetDB, it seems worth exploring subsumption relationships through join paths and opportunities offered by application specific query classes. Another direction of work is to investigate other design alternatives based on the top-down or optimistic instruction matching and develop admission and eviction policies suitable for them.

But first, and foremost, the technique seems amenable to pipelined architectures by tapping the stream at selected points in the query operator tree. To our knowledge, publicly available experimental proof is still lacking, but activities in the commercial setting are emerging [VectorWise 2010].

## 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.

## REFERENCES

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

Blakeley, J. A., Larson, P.-Å., and Tompa, F. W. 1986. Efficiently Updating Materialized Views. In *SIGMOD Conference*. 61–71.

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

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

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

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

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

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

Goldstein, J. and Larson, P.-Å. 2001. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *SIGMOD Conference*. 331–342.

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

Griffin, T. and Libkin, L. 1995. Incremental Maintenance of Views with Duplicates. In *SIGMOD Conference*. 328–339.

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

Ivanova, M., Kersten, M. L., Nes, N. J., and Gonçalves, R. 2009. An Architecture for Recycling Intermediates in a Column-store. In *SIGMOD Conference*. 309–320.

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

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

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

Luo, G. 2007. Partial Materialized Views. In *ICDE*. 756–765.

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

Martello, S. and Toth, P. 1990. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & sons, England.

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

MonetDB 2010. http://monetdb.cwi.nl/.

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

Rao, J. and Ross, K. A. 1998. Reusing Invariants: a New Strategy for Correlated Queries. *SIGMOD Conference*, 37–48.

Ross, K. A., Srivastava, D., and Sudarshan, S. 1996. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *SIGMOD Conference*. 447–458.

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

Scheuermann, P., Shim, J., and Vingralek, R. 1996. WATCHMAN : A Data Warehouse Intelligent Cache Manager. In *VLDB*. 51–62.

SkyServer 2008. Sloan Digital Sky Survey / SkyServer, http://cas.sdss.org/.

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

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

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

VectorWise 2010. http://www.vectorwise.com/.

ZHOU, J., LARSON, P.-Å., ET AL. 2007a. Dynamic Materialized Views. In *ICDE*. 526–535.

ZHOU, J., LARSON, P.-Å., ET AL. 2007b. Efficient Exploitation of Similar Subexpressions for Query Processing. In *SIGMOD Conference*. 533–544.

ZUKOWSKI, M., HÉMAN, S., NES, N., AND BONCZ, P. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proc. ICDE*. Atlanta, GA, USA.

Received October 2009; revised May 2010; accepted July 2010