Recycling in Pipelined Query Evaluation

Fabian Nagel
School of Informatics
University of Edinburgh
United Kingdom
F.O.Nagel@sms.ed.ac.uk

Peter Boncz
CWI
Amsterdam
The Netherlands
boncz@cwi.nl

Stratis D. Viglas
School of Informatics
University of Edinburgh
United Kingdom
sviglas@inf.ed.ac.uk

***Abstract*—Database systems typically execute queries in isolation. Sharing recurring intermediate and final results between successive query invocations is ignored or only exploited by caching final query results. The DBA is kept in the loop to make explicit sharing decisions by identifying and/or defining materialized views. Thus decisions are made only after a long time and sharing opportunities may be missed.**

***Recycling* intermediate results has been proposed as a method to make database query engines profit from opportunities to reuse fine-grained partial query results, that is fully autonomous and is able to continuously adapt to changes in the workload. The technique was recently revisited in the context of MonetDB, a system that by default materializes all intermediate results. Materializing intermediate results can consume significant system resources, therefore most other database systems avoid this where possible, following a pipelined query architecture instead.**

The novelty of this paper is to show how recycling can successfully be applied in pipelined query executors, by tracking the *benefit* of materializing possible intermediate results and then choosing the ones making best use of a limited intermediate result cache. We present ways to maximize the potential of recycling by leveraging subsumption and proactive query rewriting. We have implemented our approach in the Vectorwise database engine and have experimentally evaluated its potential using both synthetic and real-world datasets. Our results show that intermediate result recycling significantly improves performance.

I. Introduction

Data analysis and decision support applications typically generate complex queries that access large volumes of data, contain heavy aggregations, and have small result sizes. These queries are often generated from only a few query patterns—more so when the queries are generated by an interactive application. Successive queries are often based on the previous result by refining some of its parameters (*e.g.,* an OLAP roll-up). This suggests that there is sharing potential that could be exploited to reduce both the execution time of the entire workload and individual query response time. Concurrent query execution may further increase this potential. We refer collectively to the line of work that exploits this sharing potential by caching results during query evaluation for possible reuse in future workloads as *recycling*. Under this definition, a system supports recycling if: (*a*) all actions are performed online while processing incoming queries, (*b*) all queries are processed as they arrive (no batching or prior knowledge of the workload needed), (*c*) no intervention of the DBA is required, and (*d*) the system automatically adapts to workload changes.

Though recycling has been around for a long time, albeit under different names (see *e.g.,* [3], [6], [11], [14], [17], [18], [19]), it remained relatively dormant until it was recently revisited in the context of MonetDB [10]. There, it was shown that workloads that exhibit common subexpressions can benefit from reusing intermediate and final results and drastically improve the response time and throughput of a workload. MonetDB [1], however, is a database system that significantly differs from commercial state-of-the-art database systems. Its design is based on an operator-at-a-time paradigm: intermediate results are always materialized as a by-product of query execution. Therefore, intermediate results are already available in main memory and the recycler only has to decide which ones to keep based on the result's execution cost and size; both known at the time. This assumption is shared with most prior work on recycling. This is in stark contrast to the more widely-used tuple-at-a-time [7], or vector-at-a-time [2] query execution paradigms; both avoid intermediate result materialization through pipelined evaluation. These two extremities of query execution models, complete materialization versus no intermediate result materialization at all, present the potential for a hybrid approach. Namely, informed intermediate result materialization and reuse in pipelined query execution. This is exactly what we tackle in this paper. To the best of our knowledge, this is the first recycling system that is specifically designed for integration in a pipelined DBMS.

Introducing recycling in pipelined query evaluation presents many challenges. The materialization of intermediate and final results is not a free by-product of the execution paradigm. Materializing a result slows down query execution and consumes considerably more memory and system resources, thereby increasing cost. It becomes imperative to carefully choose which intermediate results to cache, and for how long. Thus, the system must dynamically perform a cost-benefit analysis of each potential intermediate result before it is executed and, hence, before its exact cost and size are known.

Contributions and Outline. We present an architecture for recycling intermediate and final results that is specifically tailored to the needs of a pipelined database system and is implemented in the context of Vectorwise [2], [20][1]—a modern, pipelined database system. We first give a high-level overview of the recycling process and the changes we need

[1] http://www.actian.com/vectorwise

to make to the query engine to support recycling in pipelined query evaluation (Section II). We then present the recycling process in more detail (Section III). We use state-of-the-art techniques from prior recycling work, materialized view selection and multi query optimization, but adapt them to the needs of a pipelined database system. To compensate the high cost of materializing an (intermediate) result, we extend the *benefit metric*, which is used to compute the benefit of having a result materialized, to allow for more informed decisions. We achieve this by (*a*) incorporating into decision making information and statistics collected on *all* previously executed queries, (*b*) modeling how materialized results *influence* the benefit of each other and of yet to be materialized results, and (*c*) *aging* the influence of a query invocation to allow the system to adapt to changing workloads. We use a directed acyclic graph, the *recycler graph*, as a structural representation of the past query workload. It is not only used to index materialized results but also serves as input to the benefit metric, supplying it with a wide range of parameters from the *full query history*. It is dynamically populated and updated at run-time. In the absence of historical data, we use *run-time sampling* to estimate the benefit of a result while the result is being buffered. Given the basic framework for supporting recycling, we then look at ways to maximize the recycling potential of the system (Section IV) by using techniques familiar to recycling (*subsumption*) and ones that are new in the context of recycling (*proactive recycling*).

In Section V, we experimentally evaluate our approach using both real-world (SkyServer) and synthetic workloads (TPC-H throughput runs). Our results show the clear advantage of result recycling in pipelined query evaluation. After our experimental analysis, we present the related work in the area (Section VI) and our concluding remarks (Section VII).

II. Recycler Architecture

The main component of the recycling architecture is the *recycler*, which is composed of two data structures, as shown in Figure 1: (*a*) the *recycler graph*: a directed acyclic graph (DAG) of relational operators that is used to index already materialized and cached results and to represent the past workload, and (*b*) the *recycler cache*: a finite cache of intermediate results materialized during previous query executions. Given an incoming query, the recycler: *(a)* consults the recycler graph to check if any of the contents of the recycler cache can be used to answer any part of the query, *(b)* inserts the query tree into the recycler graph, *(c)* consults the recycler graph and run-time estimates to select intermediate and final results from the query for materialization, and *(d)* materializes results during execution and adds them to the recycler cache.

Note that if a database system can leverage materialized views during plan enumeration, step (*a*) above can be skipped. This is possible if the recycler cache is exposed as a pool of materialized views to the optimizer (shown as the *optional* edge in Figure 1). Vectorwise's optimizer does not support materialized views so we could not explore this direction. Our results show that there is substantial performance to be gained even without optimizer integration. However, we can safely posit that such an integration will only improve the possibilities for reuse of materialised results.

The Recycler Graph is a structural representation of previous query invocations. Its purpose is: (*a*) to match the current query tree with previous query trees to find materialized results that can be used to answer the current query, and (*b*) to find results from the current query that could be beneficial to materialize in order to answer future queries.

The recycler graph resembles the AND(-OR) DAG that was presented in the context of materialized view selection [9] and multi-query optimization [15]. It embodies the unification of query trees from previously query evaluations; each node representing a relational algebraic operator and its parameters. For instance, one such operator might be a selection and its parameter the selection predicate it evaluates. Identical (*exactly matching*) subtrees are merged and stored only once in the recycler graph, thereby reducing the cost of finding exactly matching subtrees in the recycler graph, as there is at most one. It also reduces the memory footprint of the recycler graph.

The recycler graph is populated and navigated *dynamically* during query evaluation and has to handle a huge number of (partially concurrent) query invocations. To meet this demand, we chose to strip off OR-edges, resulting in an AND DAG where each query is represented by a single tree: the one that has been chosen by the optimizer. Only considering optimised query plans (no OR-edges) leads to a reduced complexity of matching and recycler graph maintenance (as the OR-edges may result in exponentially more alternatives to consider). This reduced complexity comes at the price of missing sharing opportunities that can only be revealed by comparing alternative query plans. This was a design choice: doing so allows us to handle more queries in the recycler graph. This in turn means that we have a more detailed record of the past workload. Hence, we can make more informed decisions on which results to materialize. This is particularly important in pipelined query processing where materializations are expensive and should only be performed if there is enough evidence that the result is frequent. This is in line with previous work on recycling, which has shown that the execution time of some workloads can be significantly improved by only considering sharing possibilities in optimized plans [10]. We will cover methods for discovering additional sharing opportunities in Section IV.

Each node in the recycler graph is annotated with statistics (*e.g.,* number of references, execution time) that have been collected during previous query runs. These statistics are used to compute the benefit of the result produced by the (partial) execution plan corresponding to that node and its subtree, as we will present in more detail in Section III-C. On a production deployment, the recycler graph has to be truncated periodically to prevent it from consuming too much main memory and to keep the matching cost manageable. The graph can, *e.g.,* be truncated by periodically removing subtrees that have not been accessed for some time. In our testing, there was no need for truncation. When comparing subgraphs of the recycler graph with a query tree, we will refer to both as

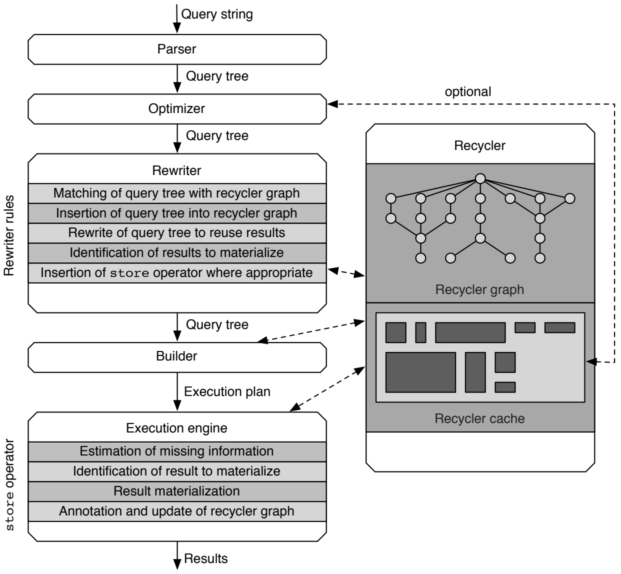

Query string
Parser
Query tree
Optimizer
optional
Query tree
Recycler
Rewriter
Rewriter rules
Matching of query tree with recycler graph
Insertion of query tree into recycler graph
Rewrite of query tree to reuse results
Identification of results to materialize
Insertion of store operator where appropriate
Recycler graph
Query tree
Builder
Execution plan
Recycler cache
Execution engine
store operator
Estimation of missing information
Identification of result to materialize
Result materialization
Annotation and update of recycler graph
Results

Fig. 1. The high-level architecture of the recycler

trees to make the discussion easier to follow.

The Recycler Cache is a finite in-memory cache of materialized intermediate and final results from previous queries that have been deemed beneficial for caching. The content of the recycler cache is managed by admission and replacement policies; both policies utilize a benefit metric. The admission policy selects intermediate results from currently executing queries for materialization, and adds them to the cache once they have been produced. If the cache is full, the replacement policy uses the benefit metric to evict results and make room for new ones, but only if the latter are deemed more beneficial.

When queries *update* the database, certain cached results may become invalid. An approach to handling updates, also proposed in [10], is to evict all cached results dependent on modified tables when a transaction commits. A better approach would build on view maintenance techniques [8] to update cached intermediates, rather than evict them. Benefit metrics should again be used to decide whether to evict or update a cached result. In this work, we focus on materialization strategies and leave maintenance out of scope.

Changes in Query Evaluation. The recycler interacts with query evaluation through *query rewriting rules* in the rewriter, and `store` operators during execution. The rewriting rules are used to match incoming query trees with the recycler graph and identify (*a*) materialized results that can be used to answer the query and (*b*) results worth materializing. The `store` operators are specialized operators that can either buffer, materialize, or plainly pass along the tuples produced by their input operators without interrupting the tuple flow.

The rewriting rules work as follows. First, a bottom-up rewriting rule traverses the optimized query tree and identifies subtrees that have an exact match in the recycler graph; subtrees that do not have an exact match are inserted into the recycler graph. A second top-down rewriting rule checks if the first rewriting rule has identified sharing opportunities. If there are matching subtrees whose result is cached, they are substituted in the query tree with an operator that uses the result from the cache. A final rule injects `store` operators on top of subtrees that have already been executed by previous query invocations (match in the recycler graph) and deemed beneficial enough for materialization. The final rule also places `store` operators on top of designated subtrees that have not been executed before. This is because the rule does not have enough information to decide on materializing the results; thus, the decision has to be postponed until execution.

Algorithm 1: matchTree(n, G)

```
input : Query tree rooted at node $n$, recycler graph $G$
output: An annotated query tree
if $n.type = scan$ then
    foreach $s \in leaves(G, n.hashkey, n.signature)$ do
        if $matches_e(n, s, \emptyset)$ then
            $(n.exact, n.mapping_e) = (s, \langle n.cols, s.cols \rangle)$;
            return;
else
    matchTree$(n.child, G)$;
    $(x, M) = (n.child.exact, n.child.mapping_e)$;
    foreach $p \in parents(x, n.hashkey, n.signature)$ do
        if $matches_e(n, p, M)$ then
            $n.exact = p$;
            $n.mapping_e = M \cup \langle n.cols, p.cols \rangle$;
            return;
```

The query builder then turns the query tree into an executable plan of operators and we move on to the run-time operation of recycling. Once the query starts being executed, the `store` operators that have already been selected for materialization materialize their input. The `store` operators that have not been chosen for materialization yet (*i.e.,* the last case outlined above) buffer their input tuple flow and use dynamic estimates to decide whether materializing the result is beneficial or not. If the result is not deemed beneficial, the store operator cancels buffering and reverts to passing its input tuples along to its parent. After the query has been executed, each operator annotates its equivalent node in the recycler graph with measured run-time parameters. The entire process is outlined in Figure 1; we will now examine it in more detail.

III. The Recycling Process

A. Matching

Matching is used to identify common subtrees between the current query tree and the recycler graph. This is necessary for both managing the recycler graph by ensuring that common subtrees are shared, but also to answer the current query. The matching algorithm that we present in this section is similar to that of AND(-OR) DAGs [9], [15]. Here, however, matching is performed *dynamically* during query processing against *every* intermediate result from the past query workload. Hence, efficient and scalable matching/insertion as well as concurrent modifications of the recycler graph are crucial challenges.

Exactly matching nodes are identified through a notion of *bisimilarity*. Two nodes v and w exactly match if: (*i*) v and w represent the same type of operation (*e.g.,* both are selections); (*ii*) the parameters of the two nodes are equal (*e.g.,* they evaluate the same selection predicate); (*iii*) they have the same number of children v^c and w^c and $\forall v_i \in v^c$ there is an exactly matching $w_i \in w^c$. The matching procedure shown in Algorithm 1 is used during a bottom-up pass through the query tree, performed by the rewriter. It is used to find matches for the query or any of its subtrees in the recycler graph. To avoid cluttering the description, we only show the algorithm for unary nodes and for exact matching. Generalizing to nodes with multiple children is straightforward; we treat subsumption and non-exact matching in Section IV-A.

Each node of the query tree has a reference to its corresponding node in the recycler graph (n*.exact*) and a name mapping (n*.mapping*). Both are assigned during the matching process. Furthermore, every node in the query tree and the recycler graph has a hash-key (n*.hash*) and a signature (n*.signature*) to speed up matching, both derived from operator characteristics. The hash-key is derived from characteristics that should exactly match (*e.g.,* the operator type) and is used in small hash-indexes attached to each node of the recycler graph to quickly identify interesting matching candidates (parent nodes). There is also a global hash table for efficiently matching table scans (recycler graph leaf nodes). An integer mask (n*.signature*), in which each column switches on one bit, is subsequently used to quickly eliminate candidates that do not provide all needed columns.

For each leaf node in the query tree (*i.e.,* a table scan operator—lines 1 to 5), the rule requests a list of candidate leaf nodes from the recycler graph using the hash/signature mechanism. The hash-key of a leaf node uses the table name and the type of the node's parent (*e.g.,* selection). For each column in the scan, one bit is set to 1 in the mask (determined by a hash function on column name). The leaf node is matched with each candidate s through a call to $\mathsf{matches}_e()$, which compares the type and parameters of the operations and returns true if nodes n and s match exactly; the third parameter is a list of mappings, which, for leaves, is empty. Since exactly matching nodes are unified in the recycler graph, there can be at most one exactly matching candidate. Once it is found, the remaining candidates do not need to be checked (line 5). If an exact match is found the rule creates an initial name mapping between the column names used by the query leaf node and the ones used by the exactly matching candidate. This mapping is required because the query could assign different column identifiers than the ones used in the recycler graph. The query leaf node is annotated with a reference to the candidate and the name mapping between these nodes.

For non-leaf query nodes (lines 8 to 13), a node n can only have an exact match in the recycler graph if all its children have an exact match. Thus, candidates for matching are all parents of the node in the recycler graph that has successfully been matched with the child of n. Since common subtrees are shared in the recycler graph, each node can have several parents that are stored in a hash index at the node. As with leaves, we probe the hash index of candidate matches based on the node's hash-key and prune the resulting candidates using the node's signature (lines 8). To test whether the node n from the query tree and a candidate p are exact matches, we call $\mathsf{matches}_e()$ with the name mapping that was created when matching the child of n as third parameter. During matching, $\mathsf{matches}_e()$ updates the name mapping when new names are assigned, or columns are no longer used. If an exact match is found, the remaining parents do not have to be checked (line 13) and node n is annotated with a reference to the candidate and its name mappings (lines 11 and 12).

B. Building the Graph

Nodes with no exact match are inserted into the recycler graph. The insertion process copies the node and its parameters and initializes the state of that node. The recycler graph uses unique names for column and dataflow identifiers that are assigned by the query (*e.g.,* the name of an output column of an aggregation). We avoid duplicate name assignments in the recycler graph by appending a query-specific identifier. A mapping is maintained to keep track of the different names used in the query tree and the recycler graph. It is created or extended whenever nodes are added to the recycler graph. If the inserted node is a leaf node, a reference is added to the hash table in the recycler that is used to find all candidate leaf nodes during matching. If a non-leaf node is copied to the recycler graph, it is either added as a parent of the node that has been inserted by its child or, if the child had an exact match, it is added as a parent of the exact match. In the latter case, the exact match of the child will have more than one parent; it can even have many, and thus we insert the node, its hash-key and its signature into the small parent hash-tables attached to each node. If the node assigns new names, the mapping that has been created while matching or inserting the child is updated. Inserting query trees at the granularity of a node rather than the entire query tree enables the recycler to detect intra-query sharing possibilities.

To support concurrently executed queries and simultaneous updates to the recycler graph, we employ optimistic concurrency control [12] at a node granularity. Queries proceed with matching as if they were being executed in isolation. Before they attempt to add a node to the recycler graph, the operation is checked for conflicts through backwards validation. The transaction checks if there have been any conflicting modifications to the recycler graph since it started matching the node that it is about to add to the recycler graph. If there are conflicts, insertion is aborted and matching of that node is restarted. Otherwise, the node is added to the recycler graph and the rewriting rule continues matching the next node in the query tree. Note that queries inserting the same subtree into the recycler graph can therefore overtake each other while successively inserting the nodes of the subtree. This is possible because the mapping used for matching and the one used for insertion are interchangeable.

An example of the matching and recycler graph building processes is shown in Figure 2. We have a simplified illustration of the recycler graph on the left and an incoming query tree on the right. Matching for this query proceeds as follows: (*1*) First a match is found in the recycler graph for the scan on relation R. The scan node in the query tree is annotated with a reference to the matching node in the recycler graph

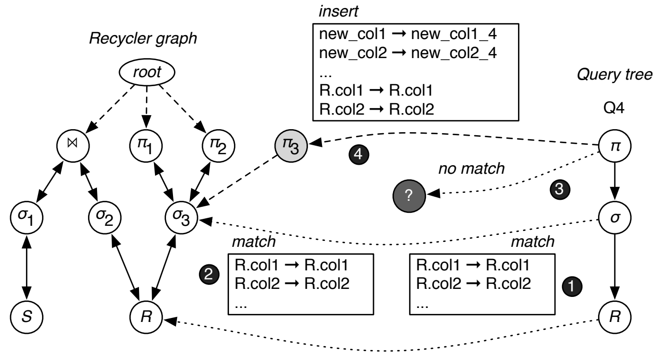


Fig. 2. An example of the matching procedure

and a name mapping is created between the names used in the query tree and the recycler graph. (*2*) Next, the selection σ from the query tree is matched with the parents σ_2, σ_3 of the scan in the recycler graph that matched with the scan in the query tree. Selection σ_3 matches and the node in the query tree (σ) is annotated with a reference to σ_3. The mapping is not updated because the selection did not assign any new column names. (*3*) Then, the projection π from the query tree is matched with the parents π_1, π_2 of the node in the recycler graph that matched with π's child (σ_3). This time, there is no matching projection in the recycler graph. (*4*) The insertion process creates a copy of π (π_3) and adds it as parent of the node in the recycler graph that has matched with π's child. Node π is then annotated with a reference to the new node. As π assigns new column names, these names are renamed in the recycler graph by appending the (unique) query identifier (_4) and the existing mapping is extended with all new names.

C. Benefit-Based Result Selection

Recycling systems traditionally rely on some kind of benefit metric to decide which results to materialize [17], [16], [10]. In pipelined query evaluation, the choice of what result to materialize is particularly important because materializing a result comes with a high cost. Materializing a result is beneficial only if materializing and then reusing it will improve the execution time of the entire workload. This depends on the initial materialization cost, the number of times a materialized result will be reused while cached, and the execution time saved with each reuse. In pipelined query evaluation, the recycler has to decide whether to materialize a result before computing it and at that time, none of these factors are known. Instead, the recycler uses reference statistics and run-time measurements that have been collected during previous computations of a result and are stored in the recycler tree; or dynamic run-time estimates in the absence of such information. The recycler uses a benefit metric to assess each result; it is used to select (*a*) results from the current query for materialization, and (*b*) cached results for eviction from the cache. The benefit $\mathcal{B}(R)$ of result R is defined as:

$$\mathcal{B}(R) = \frac{\mathsf{cost}(R) \cdot h_R}{\mathsf{size}(R)} \tag{1}$$

where (*a*) $\mathsf{cost}(R)$ is the *true cost* to compute R: it is defined as the optimal CPU time required to compute R using base tables and/or other results in the recycler cache; (*b*) h_R is the *importance* of a result: it uses previous references (previous occurrences of the same partial plan) as an estimate of the likelihood of future references [10], [14], [16]; and (*c*) $\mathsf{size}(R)$ is the *memory footprint* of R in the recycler cache.

The $\mathsf{size}(R)$ of a result R is the space it occupies in the recycler cache. When a result is materialized, its actual size is stored in its corresponding node of the recycler graph. If the result has not been materialized before, the size needs to be estimated; we use the measured cardinality and the tuple width of the result to do so. For variable-width columns, we sample incoming tuples at run-time to estimate the width.

In what follows we assume that, when computing the benefit of a result, the result has already been computed by a previous query invocation and the recycler graph contains information about previous executions that can be utilized to compute the benefit of the result. We will deal with speculatively materializing results that have not been seen before in Section III-D. A node m is subsequently called a *direct materialized descendant* (DMD) of another node v, if: (*i*) m is a descendant of v; (*ii*) the result of m is materialized; and (*iii*) there is no node that is an ancestor of m and a descendant of v whose result is also materialized. A *potential* DMD is a node that would have been a DMD had its result been materialized.

True Cost and Base Cost. The *true cost* is defined as the optimal CPU time required to compute a result using base tables and/or other materialized results in the recycler cache. Although it is used to compute the benefit of a result, it is not stored explicitly in the recycler graph. Instead, the *base cost* is stored at the nodes of the recycler graph and the true cost is computed from the base cost as needed. Recomputation is cheap and avoids having to navigate upwards the recycler graph when results are added or evicted, hence, scaling better with increasing recycler cache size. The base cost is the cost to compute the result from base tables only. It is measured during the execution of each operator while producing the result and its final value is stored at the corresponding node in the recycler graph. In contrast to the base cost, the true cost also includes the interaction with other materialized results. If a node has one or more DMDs, they are used to compute the result instead of reproducing the result from base tables. This decreases the true cost of the result and therefore also its benefit. The true cost $\mathsf{cost}(R)$ is computed from the base cost $\mathsf{bcost}(R)$ as follows:

$$\mathsf{cost}(R) = \mathsf{bcost}(R) - \sum_{j \in \mathrm{DMDs}(R)} \mathsf{bcost}(R_j) \tag{2}$$

The value of $\mathsf{bcost}(R)$ stored in the recycler graph is updated with the current measurement each time the result is recomputed to reflect the most up-to-date system load.

A recycler graph where every node is annotated with its total number of references, its base cost, and its result size is shown in Figure 3. If, for example, the results of σ_1 and σ_2 are materialized in the cache, the true cost to produce the result of π_1, π_2 or ⋈ is reduced by the base cost of σ_1 and σ_2 because the recycler would use their results.

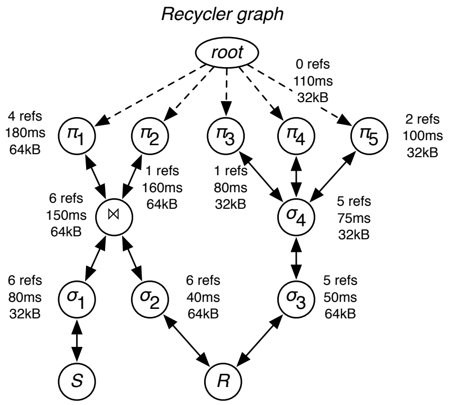


Fig. 3. Example annotations in a recycler graph

Importance Factor. The importance factor h_R of a result R is defined as the maximum number of queries in the recycler graph that would have been answered using R, assuming that all materialized results that are currently in the cache had been materialized the first time they were computed. Intuitively, it is a metric of the frequency of use of the cached result. For each potential result R, h_R is stored in its corresponding node of the recycler graph, regardless of whether the result is actually materialized or not. In what follows, and for conciseness, let n denote a node in the recycler graph and $n.R$ denote that node's result; thus $h_{n.R}$ is the importance factor of node n. The value of $h_{n.R}$ is set to zero when n is first inserted into the recycler graph. When a query Q has finished the matching/insertion process for all of its nodes, the $h_{x.R}$ value for all nodes x in the recycler graph whose result could have been used to answer Q are incremented by one. The result of a node x in the recycler graph G could be used to answer Q if: (a) G and the query tree T_Q share a common subtree S that has not been inserted by Q, and (b) if there is no ancestor of x in S whose result is already materialized; if there was such an ancestor, the result of the ancestor would be used to answer Q instead. This means that, just like the total cost, the $h_{n.R}$ value of a node n is affected by other materialized results in the cache: the result of a node is only used to answer a query if none of its ancestors in the query tree are materialized.

When adding the result of a node ν to the recycler cache, all its descendants are no longer used by queries that can use the newly added result. Therefore, the value $h_{d.R}$ of each descendant d has to be reduced by the number of queries in the recycler graph that would have used $d.R$ before, but no longer are. The value $h_{\overline{d}.R}$ of descendants $\overline{d}$ whose result would not have been used to answer these queries before (as they are below a DMD of ν) should not be modified. For all other descendants d (DMDs and potential DMDs) of ν, we do:

$$h_{d.R} = h_{d.R} - h_{\nu.R} \tag{3}$$

When a new result is added to the recycler cache, updateHR of Algorithm 2 is called on the child of the corresponding node n in the recycler graph to update all h_R values of nodes that are affected by the new result; updateHR is invoked recursively and stops when it finds a materialized result.

Algorithm 2: updateHR$(m, h_{n.R})$

```
input: Child $m$ of node $n$ in the recycler graph whose result has
       been added to the cache; $h_{n.R}$ value
$h_{m.R} = h_{m.R} - h_{n.R}$;
if $m$ is materialized then return;
foreach $c \in m.children$ do updateHR$(c, h_{n.R})$;
```

When evicting the result of a node ϵ from the cache, all queries from the recycler graph that would have been answered using that result ($\epsilon.R$) are no longer able to use it. Instead, these queries would have been answered using the result of the node's DMDs; let d be such a DMD *or* a potential DMD. The value of each $h_{d.R}$, therefore, has to be increased by the number of queries in the recycler graph which would not have used the result before, but can use it now. For all DMDs and potential DMDs d of the evicted node ϵ, $h_{d.R}$ is modified:

$$h_{d.R} = h_{d.R} + h_{\epsilon.R} \tag{4}$$

The number of references shown in Figure 3 are the nodes' h_R values without any result materialized. After σ_4 is materialized, h_{σ_3} has to be reduced by h_{σ_4} because all queries that contain σ_4 would use the materialized result instead of recomputing its subtree: h_{σ_3} = h_{σ_3} - h_{σ_4} = 5 - 5 = 0. After π_5 is materialized, h_{σ_4} has to be reduced by h_{π_5} because it is no longer used by queries that contain π_5: h_{σ_4} = h_{σ_4} - h_{π_5} = 5 - 2 = 3. It is, however, still used by queries that contain π_3 or π_4. The value of h_{σ_3} is unaffected as its result would not have been used by any query that has to compute π_5: these queries would have used the result of σ_4 instead.

Aging. The decreasing predictive power of old node re-uses is modeled by *aging* the $h_{n.R}$ of all nodes, for every query:

$$h^t_{n.R} = h^{t-1}_{n.R} \cdot \alpha \tag{5}$$

where $h^t_{n.R}$ denotes the value of $h_{n.R}$ at time t and $\alpha < 1$ models the weight of aging (*i.e.,* how quickly we want queries to age). To prevent having to update every node in the recycler graph at each query invocation, all aging is performed at once whenever a node is referenced (*i.e., lazy aging*). Aging enables the benefit metric to adapt to changing workloads.

D. Speculation

In the previous section, we described the benefit metrics for results whose (partial) query plans have already occured in the workload. However, it can also be beneficial to *speculatively* materialize results that have not been executed before. In particular, computationally expensive results with a small result size are good candidates for speculative materialization. In this scenario, for a node n, the importance factor $h_{n.R}$ of its result is zero and we have not recorded the computational cost and the size of $n.R$. Therefore, the computation of $\mathcal{B}(n.R)$ has to be postponed until execution. Recall from Section II that the recycler employs this mechanism by injecting `store` operators after designated operators in the query tree that have not been executed before. These `store` operators are inserted after all operators that are expected to be computationally expensive and are likely to have a small result size (*e.g.,* the final result of a query, or the result of an aggregation). When executing the query, each of these `store` operators monitors and temporarily buffers the tuple flow and estimates $\mathsf{cost}(n.R)$ and $\mathsf{size}(n.R)$ for its corresponding input operator n in the recycler graph. It does so by extrapolating run-time measurements using the current progress of the operator which produces the result. We use a variant of progress meters [13] to obtain the progress of each operator in the execution plan. The progress of an operator that has processed n tuples out of a total of m tuples is n/m. Scans and blocking operators know of their current progress because they know the total number of tuples that need to be processed. All other operators in the execution pipeline have the same progress as the closest scan or blocking left-deep descendant. The recycler uses cost and size estimates and a small constant for $h_{n.R}$ (*e.g.,* 0.001) into Equation 1 to decide which result to materialize. If materializing the result is not deemed beneficial, the `store` operator stops buffering; if it is, it fully materializes the result.

E. Cache Policies

Managing the recycler cache is essentially a knapsack problem (maximum benefit with a cache size restriction), which we address along the lines of the basic greedy algorithm by Danzig [4]. The results in the recycler cache are classified in a limited number of groups according to the logarithm of their size; results in the same group are arranged in increasing benefit order. Whenever the benefit of a result changes (by other results being added or evicted, or the result being reused), its benefit is recomputed and, if necessary, the result is moved to a different position in the group.

Admission Policy. While there is still enough space in the recycler cache, we materialize the result with the highest benefit of every subtree that has been seen before in the workload; and/or every result that was chosen by speculation. It is added to the appropriate group size-wise and to the appropriate position within that group in terms of benefit.

Replacement Policy. If the recycler cache is full, a new result will be materialized and added to the cache if there is a set of cached results that (*a*) has a lower average benefit, and, (*b*) if evicted, creates enough space to materialize the new result. The replacement policy only checks in the appropriate group for this set of results. It scans the results in increasing benefit order keeping track of the sum of sizes and the average benefit until either (*a*) the average benefit exceeds that of the result, so we decide not to materialize the result after all; or (*b*) the results in this set together are large enough, such that they can be replaced by the new result once fully materialized.

IV. Maximizing Recycling Potential

We now explore ways of maximizing the potential for recycling by (*a*) extending exact matching to also account for partial matching by leveraging *subsumption*, and (*b*) *proactively* executing queries and caching their results.

A. Subsumption

Even if a node from the query tree has not been exactly matched with a node in the recycler graph, its result can sometimes be derived from the recycler cache. This happens if there is a result in the recycler cache that subsumes [15], [10] the result of that node. A node n subsumes a node m if the result $m.R$ can be derived from the result $n.R$. We consider two kinds of subsumption. *Column subsumption* requires that the columns of $m.R$ are a subset of the columns of $n.R$ and, hence, $m.R$ can be derived from $n.R$ by projecting out columns. For example, if $n.R$ is the result of the aggregation ($\mathcal{F}$) query $n.R =_{\text{age}} \mathcal{F}_{\texttt{sum}(\text{slry}),\texttt{min}(\text{slry})}(S)$ over some input S and $m.R =_{\text{age}} \mathcal{F}_{\texttt{sum}(\text{slry})}(S)$ then $m.R$ can be derived by using the columns of $n.R$ and projecting on $\texttt{sum}(\text{slry})$. *Tuple subsumption* requires additional relational operations to derive $m.R$ from $n.R$. For instance, if $n.R =_{\text{age,dno}} \mathcal{F}_{\texttt{sum}(\text{slry})}(S)$ and $m.R =_{\text{age}} \mathcal{F}_{\texttt{sum}(\text{slry})}(S)$, then $m.R$ can be derived from $n.R$ by $m.R =_{\text{age}} \mathcal{F}_{\texttt{sum}(\text{slry})}(n.R)$, *i.e.,* by applying m's operation with $n.R$ as the input. A subsumption variant we leave to future work is *dimension subsumption*. For example, if $n.R$ is the result of the aggregation $n.R =_{\text{nation}} \mathcal{F}_{\texttt{sum}(\text{slry})}(S)$, $m.R =_{\text{region}} \mathcal{F}_{\texttt{sum}(\text{slry})}(S)$ and *region* is higher in a dimension hierarchy than *nation*, then $m.R$ can be derived from $n.R$ by $m.R =_{\text{region}} \mathcal{F}_{\texttt{sum}(\text{slry})}(n.R)$. This requires the database system to be aware of dimension hierarchies.

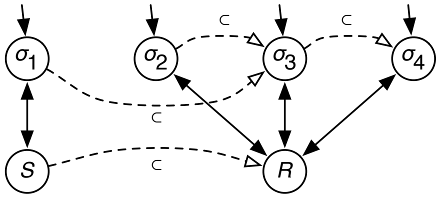
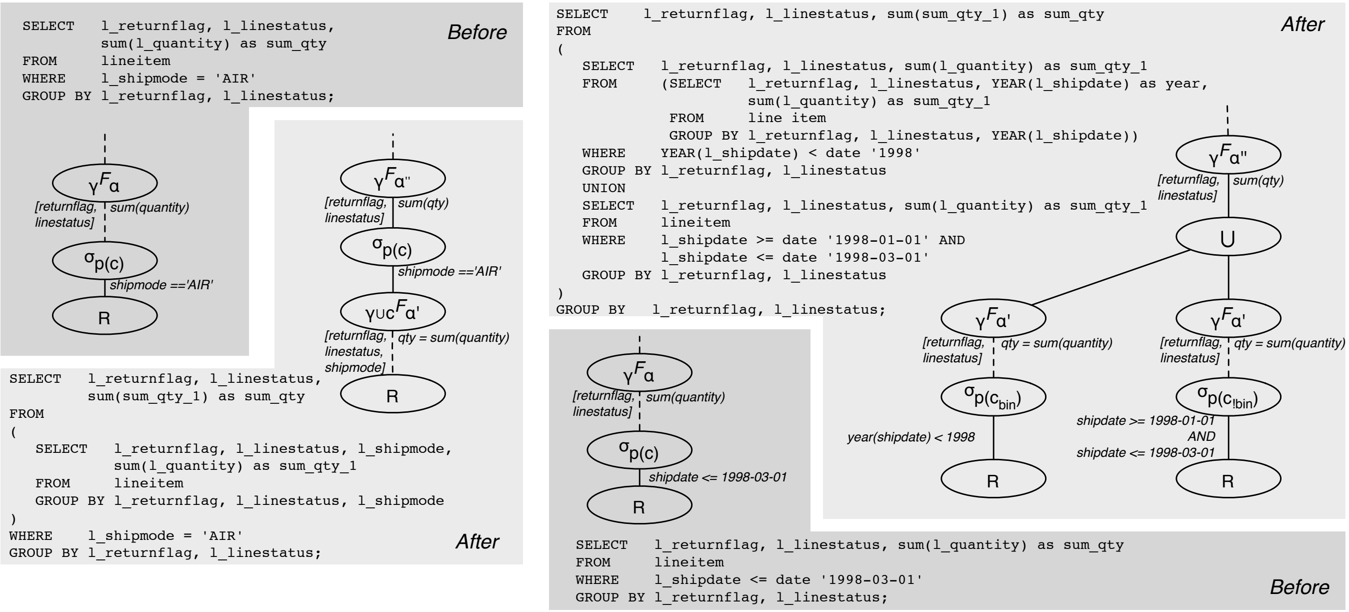


Fig. 4. Subsumption edges in the recycler graph

The requirements for our subsumption system were: (*a*) the complexity of finding an exact match should remain unaffected by the subsumption implementation and extra effort should only be spent on searching for subsuming results if there is no exact match in the recycler graph, and (*b*) subsumption references of each node (*i.e.,* how often the result would have been used by subsuming queries) should be accessible through the graph. Based on these requirements, we add subsumption to the recycler graph as specialized OR-edges, called *subsumption edges*. They are specialized in the sense that matching considers subsumption edges only after all regular edges have been checked and no exact match has been found. Additionally, transitive subsumption relationships are not explicitly annotated in the recycler graph structure but result from the structure itself. A part of a recycler graph with subsumption edges is shown in Figure 4, where $\subset$ denotes subsumption. Though σ_4 subsumes σ_1, σ_2 and σ_3, the recycler only creates a subsumption edge to σ_3 because σ_3 subsumes σ_1 and σ_2. All subsumption relationships of a node can be found by following subsumption edges.

B. Proactive Strategies

Proactive recycling strategies choose to execute a more expensive query to compute an intermediate result, which can then be used to compute the desired result. Such a policy only makes sense if the larger intermediate result has recycling potential. Similar techniques (*e.g.,* cube caching) have been extensively studied in view materialization [5]. We extend



```
SELECT   l_returnflag, l_linestatus,
         sum(l_quantity) as sum_qty
FROM     lineitem
WHERE    l_shipmode = 'AIR'
GROUP BY l_returnflag, l_linestatus;
```

```
SELECT   l_returnflag, l_linestatus,
         sum(sum_qty_1) as sum_qty
FROM
(
   SELECT   l_returnflag, l_linestatus, l_shipmode,
            sum(l_quantity) as sum_qty_1
   FROM     lineitem
   GROUP BY l_returnflag, l_linestatus, l_shipmode
)
WHERE    l_shipmode = 'AIR'
GROUP BY l_returnflag, l_linestatus;
```

```
SELECT    l_returnflag, l_linestatus, sum(sum_qty_1) as sum_qty
FROM
(
   SELECT   l_returnflag, l_linestatus, sum(l_quantity) as sum_qty_1
   FROM     (SELECT   l_returnflag, l_linestatus, YEAR(l_shipdate) as year,
                      sum(l_quantity) as sum_qty_1
             FROM     line item
             GROUP BY l_returnflag, l_linestatus, YEAR(l_shipdate))
   WHERE    YEAR(l_shipdate) < date '1998'
   GROUP BY l_returnflag, l_linestatus
   UNION
   SELECT   l_returnflag, l_linestatus, sum(l_quantity) as sum_qty_1
   FROM     lineitem
   WHERE    l_shipdate >= date '1998-01-01' AND
            l_shipdate <= date '1998-03-01'
   GROUP BY l_returnflag, l_linestatus
)
GROUP BY   l_returnflag, l_linestatus;
```

```
SELECT   l_returnflag, l_linestatus, sum(l_quantity) as sum_qty
FROM     lineitem
WHERE    l_shipdate <= date '1998-03-01'
GROUP BY l_returnflag, l_linestatus;
```

Fig. 5. Proactive Strategies: cube caching with selections (left), with binning (right)

them and adapt them for recycling intermediate query results.

Top-N Caching. The motivating example for the introduction of proactive strategies are top-N queries; or even windowing queries like LIMIT N SKIP K, typically in combination with ORDER BY. The SQL standard supports various syntactical incarnations, and such queries are sometimes used to page through results. Vectorwise natively supports such top-N queries through the topN operator. It does not sort its input to produce the top-N results; rather, it maintains a heap of size N to maintain the top N results (for input of size M) at just $O(M \log N)$ complexity. The proactive *limit* strategy exploits the fact that $L = \mathtt{topN}(Q, 10{,}000)$ for any query Q is practically as cheap as $L' = \mathtt{topN}(Q, N)$ for any $N \in (1, 10{,}000]$ as long as the heap fits the CPU cache. Thus, one can proactively execute L instead of L', materialize its result $L.R$, and then employ subsumption to recycle $L.R$ by rewriting L' to use $L.R$. Hence, by proactively generating a larger result we create opportunities for subsumption recycling.

Cube Caching with Selections. The benefit metric of Equation 1 grows as results become expensive and small. Typical examples of queries exhibiting such behavior involve aggregations, which reduce a large (thus expensive) input to a small result. In this strategy we target queries involving aggregation in conjunction with selection. Algebraically, these queries are expressed as $Q =_{\gamma} \mathcal{F}_{\alpha}(\mathcal{P}((\sigma_{p(c)}(R)))$ where γ is the set of GROUP BY columns; α are the aggregates (*e.g.,* sum, avg, *etc.*); $\mathcal{P}$ is an arbitrary plan containing some selection with a predicate $p(c)$ over a column c; and R is the selection input, which may be an entire query itself. Assume now that the recycler observes a workload with similar queries that only differ slightly in the selection predicate $p(c)$. However, due to this difference, the result of the aggregation cannot be reused.

A proactive strategy can rewrite Q above into a query $Q' =_{\gamma} \mathcal{F}_{\alpha''}(\sigma_{p(c)}({}_{\gamma\cup c}\mathcal{F}_{\alpha'}(\mathcal{P}(R))))$ where γ is extended with c, the selection is pulled above the aggregation, and the result is computed by using the original groupings γ and a reaggregation α'' of α' using standard aggregate calculation decomposition rules (*e.g.,* if α'' contains $\mathtt{avg}(x)$, then α' must contain $\mathtt{count}(x)$ and $\mathtt{sum}(x)$, *etc.*). Doing that, we now have the subquery ${}_{\gamma\cup c}\mathcal{F}_{\alpha'}(\mathcal{P}(R))$ without the limiting selection condition, which will be cached in the recycler thus taking away most query evaluation cost and presenting opportunities for further reuse. An example of cube caching with selections is shown on the left of Figure 5. This strategy, though frequently applicable, is not always a good idea. Recall that the target is to cache small but expensive results. A heuristic to enforce this (other than estimating the result size) is to apply the proactive rule only if the number of distinct values of the column(s) c added to the GROUP BY list γ is smaller than a threshold. The reason is that the result size of the aggregation ${}_{\gamma\cup c}\mathcal{F}_{\alpha'}(\mathcal{P}(R))$ will increase by a factor equal to the number of unique values in c.

As the proactive strategy creates a more expensive plan in the hope of increasing reuse potential, its recycling *benefit* must steer the decision whether or not to apply it. The first step is to match and insert the proactive plan in the recycler graph, but not execute it. We can handle the presence of both the original query and the proactive variant in the recycler graph through bidirectional subsumption edges or through OR-edges. To obtain cardinality and cost information without executing the plan, the recycler must invoke the query optimizer. If a recycled result for the aggregate was found during matching, or a non-speculative store decision was made for it, we execute the proactive plan. Else, the original plan is executed. Each time this strategy is triggered and matches the proactive query variant in the graph, the common parts of the proactive plan obtain a higher benefit score. At some point a store decision may be made, causing valuable re-use of the aggregate result.

Cube Caching with Binning. We noted that the previous strategy is more sensible if the cardinality of the column(s) of the selection predicate is small. This requirement may be too restrictive, particularly for numeric columns. Hence, cube caching would often be impossible despite its great potential. Cube caching with binning tries to solve this problem. We need to take two steps: (*a*) reduce the cardinality of the selection columns through binning, and (*b*) extend the recycler's subsumption primitives to account for binning.

Binning reduces the number of distinct values in a column by combining them in bins based on some commonality. For numerical columns it is possible to cheaply compute a bin number from the column domain. For instance, in a numerical column such as price, with values in $[0, 10000]$, using $\frac{\text{value}}{100}$ will bin the column into 101 bins. Another example is date columns where the optimizer can bin on year or year-month, thus reducing the number of unique values of the column. If we restrict ourselves to range predicates, we can divide the bins into two classes: contained and non-contained by the bins. Hence, an aggregation result can be rewritten as the re-aggregation of the union of contained relevant binned results, combined with the full recomputation only for the non-contained areas. Given a query $Q =_\gamma \mathcal{F}_\alpha(\mathcal{P}(\sigma_{p(c)}(R)))$ and a decomposition of $p(c)$ into the union of two selections: $p_l(c_l)$ in terms of binnable c_l (binned c columns with a low number of unique values) and $p_r(c)$ in terms of the original columns c, we can rewrite it as $Q =_\gamma \mathcal{F}_{\alpha''}(Q_l \cup Q_r)$ with $Q_l =_\gamma \mathcal{F}_{\alpha'}(\mathcal{P}(\sigma_{p_l(c_l)}(R)))$ and $Q_r =_\gamma \mathcal{F}_{\alpha'}(\mathcal{P}(\sigma_{p_r(c)}(R)))$. The previous cube caching technique will now trigger on Q_l since it satisfies the requirement of few unique values.

For example, consider a column of type date, named *shipdate*, and a selection predicate *shipdate* $\leq$ `1998-03-01`. When binning on year(*shipdate*), the predicate can be decomposed as year(*shipdate*) $<$ `1998` and (*shipdate* $\geq$ `1998-01-01` $\wedge$ *shipdate* $\leq$ `1998-03-01`), so the entire query can be rewritten as the union of its two constituents (see also the right part of Figure 5). The benefit is that Q_l will be rewritten by the previously described "cube caching with selections" strategy, extending its grouping (*returnflag*, *linestatus*) with the selection attribute year(*shipdate*). This aggregate will reoccur in similar queries that also trigger this rule. It typically produces a small result, yet takes most of the execution effort, and thus stands a high chance of being cached by the recycler.

Cube caching with binning is closely related to dimension subsumption. The multiple grouping dimensions and their multiple dimension levels (*e.g.,* year, month, day) form a lattice of proactive recycling candidates [5]. Dimension subsumption requires pre-knowledge by the DBMS of hierarchical dimensions. Since Vectorwise does not support this yet, we did not experiment with dimension subsumption.

V. Evaluation

We now demonstrate the advantages of our recycling techniques. We have used two workloads; the SDSS SkyServer to see the impact of recycling on real-world applications; and TPC-H throughput runs to show the benefits in a controlled environment. We used a system with two quad-core Intel Xeon 2.80GHz processors with 48GB RAM and 16 disk RAID-0 SSD storage. Where not stated differently, we split our main memory budget into 14GB for query processing, 10GB for the buffer pool and 18GB for the recycler cache. A bufferpool of 10GB was sufficient for keeping the working set in main memory at all times. To exclude I/O effects, all results are warm runs, where the working set was already in the bufferpool.

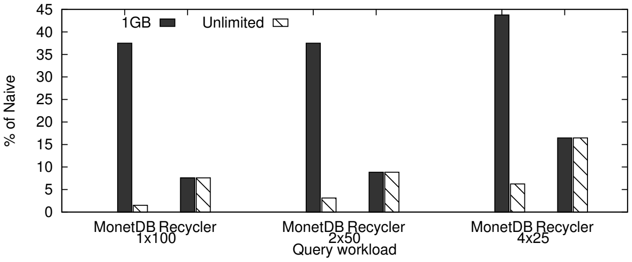


Fig. 6. Impact of recycling on SkyServer queries

Skyserver is a 18TB scientific database with 98 tables, 51 views and 144 persistent functions. We used a 100GB subset of the SkyServer Data Release 7 (DR7), the one used in the MonetDB recycling work [10]. We also used the same query workload as in [10] to allow a direct comparison. The workload consists of 100 queries that were randomly picked from a real-life query log and has a high recycling potential as it benefits from reusing intermediate and final results [10]. The most frequent query pattern in the workload is:

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

The workload queries are either identical to the one above, or share the computation of `fGetNearbyObjEq(195, 2.5, 0.5)`. The recycler materializes the results of the function call and the final results of these queries. As these results consist of a few tuples, the recycler cache only needs a few hundred KB to fit them all. This contrasts [10] where a cache size of 1.5GB was required to keep all results. This is because the MonetDB recycler does not use a recycler graph; rather, it matches directly on cached results. Hence, it needs to keep all intermediates that lead to a result.

The comparison to the naive approach (*i.e.,* no recycling) of the two systems is shown in Figure 6. To simulate the effect of updates invalidating cached results, we split the batch of 100 queries into shorter batches of 25 and 50 queries and ran them with flushing all cached results in between. For each run, we either gave a total of 1GB to the recycler cache, or kept the cache size unlimited. For both approaches the workload benefits greatly from recycling. In the comparison, MonetDB benefits the most with an unlimited recycler cache size. This is due to (*a*) their approach not having to explicitly instruct result materialization because of the execution paradigm of MonetDB (operator-at-a-time), and (*b*) their matching technique being more light-weight. In the limited recycler cache scenario, our approach shows a greater improvement. Our technique benefits from being able to selectively decide which result to materialize instead of having to keep all intermediates in the result's subtree. Both systems react similarly, when having their recycler cache flushed. Even with a refresh every 25 queries, recycling is still quite beneficial.

TPC-H. We tested the impact of recycling on the TPC-H throughput test with 4 to 256 query streams, as generated by the standard QGEN tool. Each stream consisted of the 22 TPC-H query patterns in different orders and with different randomly assigned parameters, according to the benchmark specification. All experiments were run against a 30GB database (scale factor SF = 30). The sharing potential of this workload is due to each query pattern only having a limited number of valid values for each parameter (*i.e.,* each selection predicate). With several streams being executed, it is likely that some of the queries in each stream have the same parameters assigned. If two or more queries of the same pattern have some parameter values in common, then intermediate results might be shared between them; if they have all parameter values in common, the final result can be shared. Furthermore, some query patterns have fixed subtrees that can be shared between queries of the same pattern. Finally, the potential for sharing grows as more streams are executed. Vectorwise was set up to execute 12 queries in parallel; the system queues any further concurrent queries. If multiple active queries share computation, the recycler stalls all but one until it has either finished materializing the result, or decides not to materialize.

We evaluate the recycler in three modes of execution. The *history mode (*HIST*)* only uses measurements from previous query invocations to decide whether to materialize a result. It is based on the assumption that results that occurred more than once in the past are likely to occur again. Thus, it only materializes results that have been seen before. All materialization decisions can be made in the rewriting phase and, hence, there is no need for buffering partial results in the execution phase. Since results are only considered for materialization the second time they are computed, a result has to appear at least three times in a workload for the recycler to benefit from reusing it. Thus, the history mode always misses one reuse possibility (the second time the result is computed) and performs badly for results that only occur twice in the workload. The *speculation mode (*SPEC*)* improves this by also speculatively materializing small results in the hope that they will occur again. If a result is indeed frequent and is materialized the first time it is executed, the recycler can already benefit from reusing it the second time it occurs. Speculation relies on dynamic estimates and therefore requires buffering the result until a decision can be made. The last mode is the *proactive mode (*PA*)*. Since proactive rules are not implemented in the recycler, we simulate their benefit by manually altering query plans according to the rules of Section IV-B; this affects TPC-H queries $Q1$, $Q16$, and $Q19$. Queries $Q16$ and $Q19$ use cube caching with selections, while query $Q1$ uses cube caching with binning.

Throughput Scaling. We use 4, 16, 64 and 256 streams to evaluate recycling in all three modes of execution. The average execution time per stream is shown in Figure 7. The execution time per stream is measured from the time its first query is issued to the time the result of its last query is received. For 4 streams we saw a 10% improvement over naive. For 16 and 64 streams there was an improvement of 24% and 55%, respectively; but the greatest improvement was 79% for 256 streams. Speculation gave better results than history; proactive combined with speculation scored the best results from 64 streams upwards. For 4 and 16 streams speculation performed slightly better than proactive, because there were not enough reuse possibilities in the workload to amortize the extra execution cost of the proactive query plans. With an increasing number of streams and hence a greater sharing potential, we saw improvement for the history mode. This is because at 256 streams almost every beneficial result in the workload occurs 4 or more times.

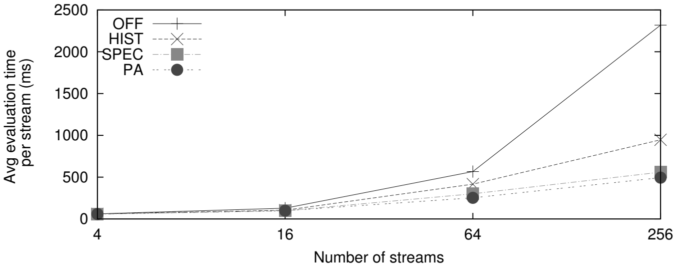


Fig. 7. Average time per TPC-H stream

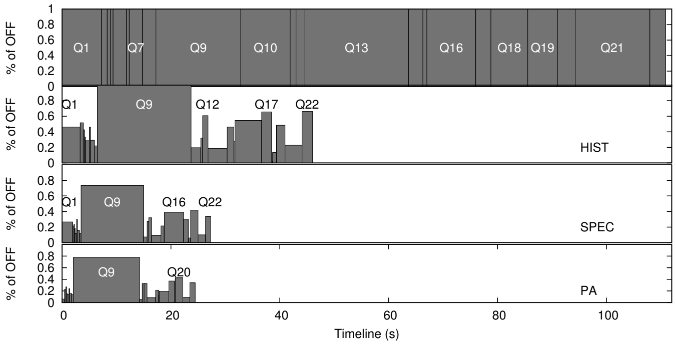


Fig. 8. 256-stream breakdown into average times

Per-Query Breakdown. We break down the results for 256 streams into single query patterns in Figure 8. The x-axis is the average execution time for each TPC-H query pattern $Q1$ to $Q22$. Unlike Figure 7, the displayed times do not include the time the query is waiting to start processing. The recycler effect is visible from top (naive) to bottom (proactive). The y-axis is the relative execution time in relation to naive. In history mode, all queries but $Q9$ benefit from recycling. The pattern for query $Q9$ has one parameter with nearly 100 different values. The results suggest that any of these values appeared at most two times in the workload: history could not benefit from it, but speculation could. Queries $Q8$, $Q10$, $Q13$, $Q18$ and $Q19$ showed the greatest improvement in history mode. Speculation improves the execution time of all query patterns, but by a smaller percentage than that achieved by history. For proactive, only queries $Q1$, $Q16$ and $Q19$ differ

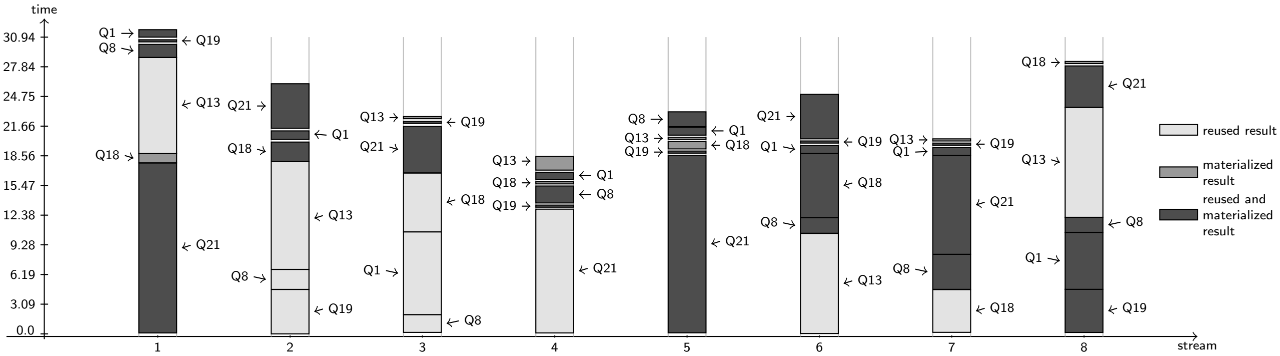


Fig. 9. Detailed timeline of concurrent stream execution of a subset of TPC-H queries

from the execution times in speculation mode because they were the only ones that could benefit from the technique. All three queries benefit strongly from proactive query plans.

Detailed Trace. To better understand what happens in each concurrent stream, we reduced the number of streams to 8 (fitting the 8 available cores) and the number of queries per stream to 6 ($Q1$, $Q8$, $Q13$, $Q18$, $Q19$ and $Q21$). For queries $Q1$ and $Q19$, the proactive versions were used. A detailed trace of the execution of this workload is shown in Figure 9. For each query of every stream (server thread), we show the timeframe in which it has been evaluated. Grey means that the query materialized a result, whereas light grey means that the query reused a materialized result. Dark grey is used when the query both reused a result and materialized another one. Speculation was turned on, so every query either materializes or reuses its final result. In what follows, S_n refers to stream n. S_3 executes the first instance of $Q1$. It speculatively materializes the intermediate result created by the proactive rule (see Section IV-B) and the final result of the query. S_8 computes the same intermediate result concurrently and therefore stalls until the result is produced by S_3. That result is used to evaluate the query and the final result is materialized. All other streams reuse the intermediate result and materialize their final results. $Q8$ is initially executed by stream S_3. This stream only materializes the final result of the query. When the query is executed for the second time by S_2, the recycler materializes an intermediate result ($\sim$ 270MB) that is shared by all instances of $Q8$. S_4, S_5, S_6, S_7 and S_8 reuse that result and materialize their respective final results. S_4, S_5 and S_6 each materialize another intermediate result ($\sim$ 400MB) that is shared by some instances of $Q8$ and has occurred in the workload for the second time. S_1 reuses one of these results (now occurring for the third time). $Q13$ is executed by S_1, S_2, S_6 and S_8 without reusing any materialized results. All streams speculatively materialize their respective final results. The other streams then reuse these final results. Note that S_4 has to wait for S_2 to produce its final result before it can reuse it. S_7 executes the first instance of query $Q18$ and materializes the final result. S_3 then materializes a large ($\sim$ 1GB) intermediate result that is shared by all instances of $Q18$, and its final result. S_2 and S_6 reuse the intermediate result and materialize their respective final results. All other streams reuse the final results that have been materialized previously. This requires S_1 and S_5 to wait for S_6 and S_2, respectively. The first stream to execute $Q19$ is S_2. It speculatively materializes the intermediate result created by the proactive rule (see Section IV-B) and the final result of the query. All other instances of the query reuse the intermediate result. S_8 has to stall until the intermediate result has been produced. S_4 executes the first instance of $Q21$. It materializes three large intermediate results (two of them up to $\sim$ 2GB each) and the final result. The results are already materialized the first time they are executed, because the recycler knows from concurrent query invocations that the results are frequent. All three intermediate results are shared by all instances of query $Q21$, but we see that S_1, S_5 and S_7 have to wait until these results are produced.

Matching Cost. Finally, we tested how well our algorithms scale with the size of the recycler graph. We measured the cost of matching a query tree with the recycler graph and updating the latter with non-exact matching nodes for the TPC-H workload. The cost of matching depends on the number of recycler graph nodes the query has to be matched with. This in turn depends on the number of leaf node candidates as well as the number of parents of each matching node that have to be checked and, hence, on the size of the recycler graph. We measured the matching and insertion costs for a 256-stream run of a total of 5632 query invocations. The result is shown in Figure 10 for all 5632 query invocations (left) as well as for each individual query (right). The cost grows with the size of the recycler graph, but moderately. Note that even the highest matching cost is orders of magnitude lower than the evaluation cost of the query without recycling; the highest matching cost is 2ms where queries typically run in 0.3s-11.3s.

VI. Related Work

Recycling aims to reuse results computed as part of query evaluation to improve the execution time of a partially unknown workload. Multi-query optimization and materialized views have similar goals and use structures similar to the recycler graph. In multi-query optimization (MQO) [15], queries are submitted in batches. All queries in a batch are optimized together in order to find a globally optimal plan for the entire batch. Identifying sharing possibilities within a batch and exploiting them by reusing their results is part of this

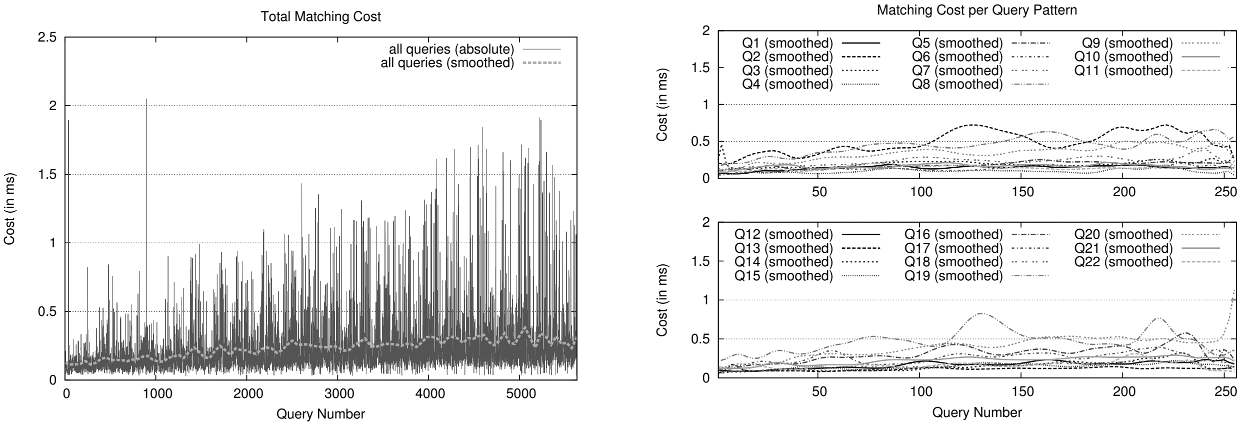


Fig. 10. Matching cost for 256-stream throughput run: all 5632 queries (left) and per query type (right)

optimization process. The MQO optimizer knows the exact number of reuses of a result in the query batch. Reusing materialized results between different batches is not common. MQO differs from recycling by requiring that queries are submitted and optimized in batches and only exploits sharing possibilities within a single query batch. Materialized views [9], traditionally, are specified by the DBA using a typical workload (*e.g.,* from the past), a space restriction and specialized tools to assist the selection process. They differ from recycling as they are static and cannot adapt to changes in the workload unless the DBA intervenes again.

Studies on how to (automatically) identify and take advantage of common sub-expressions within subsequent queries had already been conducted in the 1980s by Finkelstein [6] and Sellis [17]. Later work presented several architectures for recycling [3], [10], [11], [14], [16], [18], [19]. They differ from our techniques in one or more of the following aspects: (*a*) they assume that the results are materialized as a by-product of the execution paradigm, (*b*) they initially admit every result to the recycler cache, (*c*) they only consider caching final query results or data cubes, (*d*) they are limited to a small number of previous/concurrent queries to base the materialization decision on, and/or (*e*) they only manage reference statistics for already materialized results, which may lead to starvation (a new result is likely to be evicted because it has no reference information attached to it).

VII. Conclusions

We presented a recycling architecture for pipelined query evaluation that automatically identifies and exploits re-occurring query patterns by selectively materializing intermediate results, and reuses these to accelerate subsequent queries. It continuously adjusts to a workload without need for DBA intervention. To counter the cost accompanied by materializing results in pipelined query evaluation, we introduced a specialized *benefit* metric to decide which intermediate results to materialize. This benefit metric is fueled by (*a*) a structure called the *recycler graph* that tracks references and execution statistics from previous query executions, and (*b*) runtime estimates for unknown parameters. We also presented subsumption and proactive recycling as techniques to increase the recycling potential of a workload. We evaluated our recycling system using the same real-world workload (*SkyServer*) as in [10] and the standard TPC-H benchmark. Our results show substantial benefit of recycling in pipelined query evaluation.

Acknowledgments We would like to thank Milena Ivanova for supplying the SkyServer dataset used in [10] and the MonetDB and Vectorwise teams for their support.

References

[1] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, 2002.

[2] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *VLDB*, 2005.

[3] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: integrating query result caching and matching. In *UMIACS-TR-93-106*, 1993.

[4] G. Dantzig. Discrete-Variable Extremum Problems. *Operations Research*, 5(2), 1957.

[5] P. Deshpande, K. Ramasamy, A. Shukla, and J. Naughton. Caching Multidimensional Queries Using Chunks. In *SIGMOD*, 1998.

[6] S. Finkelstein. Common expression analysis in database applications. In *SIGMOD*, 1982.

[7] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *TKDE*, 6(1), 1994.

[8] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *DEBULL*, 18(2), 1995.

[9] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *ICDT*, 1997.

[10] M. Ivanova, M. Kersten, N. Nes, and R. Gonçalves. An architecture for recycling intermediates in a column-store. In *SIGMOD*, 2009.

[11] Y. Kotidis and N. Roussopoulos. DynaMat: a dynamic view management system for data warehouses. In *SIGMOD*, 1999.

[12] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *TODS*, 6(2), 1981.

[13] G. Luo, J. Naughton, C. Ellmann, and M. Watzke. Toward a progress indicator for DB queries. In *SIGMOD*, 2004.

[14] P. Roy, K. Ramamritham, S. Seshadri, P. Shenoy, and S. Sudarshan. Don't Trash your Intermediate Results, Cache 'em. *CoRR*, cs.DB/0003005, 2000.

[15] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.

[16] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In *VLDB*, 1996.

[17] T. K. Sellis. Intelligent caching and indexing techniques for RDBs. *Inf. Syst.*, 13(2), 1988.

[18] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic Caching of Query Results for Decision Support Systems. In *SSDBM*, 1999.

[19] K. Tan, S. Goh, and B. Ooi. Cache-on-Demand: Recycling with Certainty. In *ICDE*, 2001.

[20] M. Zukowski, P. Boncz, N. Nes, and S. Héman. MonetDB/X100-A DBMS In The CPU Cache. *DEBULL*, 28(2), 2005.