Eberhard Karls Universität Tübingen

Wilhelm-Schickard-Institut
Lehrstuhl für Datenbanksysteme
Prof. Dr. Torsten Grust

DIPLOMARBEIT

Recycling Intermediate Results in Pipelined Query Evaluation

Fabian Nagel
2675608

Betreuer: Dr. Peter Boncz
Gutachter I: Prof. Dr. Torsten Grust

Tübingen, den 28. November 2010

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum Unterschrift

Contents

1 Introduction

1.1 Motivation

Commercial query optimizers today usually optimize each query invocation in isolation. Sharing recurring intermediate and final results between successive query invocations is ignored. It is left to the user to exploit sharing possibilities between overlapping queries in a workload. Only considering single query invocations for optimization does not realize the full optimization potential of a workload. The following queries illustrate scenarios where sharing intermediate or final results could be beneficial:

- Q1: *Which items have been ordered since January 1st, 2010 and cost more than $10k?*
- Q2: *Which items have been ordered since January 1st, 2010 and were delivered by 'UPS'?*
- Q3: *Which items have been ordered since June 1st, 2010?*
- Q4: *What is the revenue of each shop in Europe?*
- Q5: *What is the revenue of all shops in each country in Europe?*

Query $Q1$ and $Q2$ have a common sub-expression. Both queries select the same range of dates from a base table. Query Q1 could materialize the intermediate result R of that selection in main memory and the materialized result could then be used to answer the second query. If the cost of materializing the intermediate and later reading it is less than the cost of computing it twice, the workload will benefit from sharing the result between both queries. $Q3$ does not share an intermediate result with either of them. However, each tuple in the result of $Q3$ is contained in R. Instead of computing the result from scratch, a selection on R could be used to answer $Q3$. The final result of $Q3$ subsumes the intermediate result R. $Q4$ and $Q5$ are another example of queries where one subsumes the other. $Q5$ could be computed by performing an additional aggregation on $Q4$.
Sharing common results between successive query invocations can reduce the execution time of these queries and improve the overall response time and throughput of the entire workload. Workloads that exhibit common sub-expressions may benefit from sharing results between queries. This is often the case for typical data warehouse workloads. Although such workloads consist of hundreds of queries, these queries are often generated from a few patterns and thus have many sharing possibilities. Especially with user interaction, subsequent queries tend to only differ in a few parameters. If the user for example *zooms in* onto the relevant data or uses the OLAP[1] operation *roll-up* to further

[1] Online Analytical Processing

summarize a result by the hierarchies within dimensions, the result of the previous query could be utilized. Query $Q3$ from the previous example zooms in onto the result of $Q1$ and Query $Q5$ summarizes the result of $Q4$ by the country each shop belongs to.
Besides the presence of common sub-expressions in a workload, sharing intermediate and final results is more likely to pay off if the shared results are computationally expensive and have small result sizes. Both of these factors influence how often a result has to be reused before the cost of materializing it paid off. Typically, data warehouses store large volumes of data that are accessed by complex queries from data analysis or decision support applications. These queries usually access a substantial part of the data, making heavy use of aggregations and their results are typically small. Because of that complexity, their execution time is usually greater than the one of OLTP[2] queries. They are often submitted interactively and therefore require low response times.
Furthermore, updates on base tables that were used to compute intermediate results restrain the usefulness of the intermediate result. Data warehouses are relatively static with only infrequent updates.
So far it has been assumed that the characteristics of successive query invocations are already known when optimizing a query. However, a future workload is usually not known and it is not possible to already identify future sharing possibilities of an intermediate result. Therefore, intermediate and final results have to be materialized speculatively. Recycling [IKNG09] refers to materializing intermediate and final results and then trying to use these results to answer incoming queries instead of recomputing the intermediates from scratch. The term recycling was first used by Ivanova *et al.* [IKNG09]. The authors extended the database system MonetDB[3] [Bon02] with recycling capabilities. The recycler for MonetDB served as a trigger for this work which will transfer their original idea into the context of pipelined query evaluation. This work will describe the implementation of a recycling architecture for VectorWise[4] [ZBNH05, BZN05], a modern pipelined database system. The system is evaluated using the TPC-H[5] benchmark suite. TPC-H is a decision support benchmark.

1.2 Challenges

When implementing a recycler architecture, some of the following challenges have to be addressed:

1. *Which intermediate and final results should be materialized?*
2. *Should the recycler cache relations only or other query-execution generated data structures as well (e.g. hash tables)?*
3. *Where to materialize these results?*
4. *How to materialize these results?*

[2] Online Transaction Processing
[3] monetdb.cwi.nl/
[4] www.vectorwise.com
[5] www.tpc-h.org

5. *Would the recycler also work when storing the results in other storage layers than main memory?*

6. *How to deal with a restricted result cache size?*

7. *How to verify that a materialized result from the cache can be used to answer an incoming query?*

8. *Can the recycler identify additional materialized results that could be used to answer a query (subsumption)?*

9. *How to use the identified result to answer the query?*

10. *How to deal with updates on base tables that were used to compute some of the results in the recycler cache?*

The approach described in this work will present solutions to some of these issues. It will focus on exploring recycling in pipelined query evaluation. Since pipelined DBMS do not implicitly materialize all intermediate and final results, special attention has to be payed to the following issues:

1. *Which intermediate and final results should be materialized?*

3. *How to materialize these results?*

1.3 Related Work

Early work on caching intermediate and final results was already done in the 1980s. Finkelstein and Sellis [Fin82, Sel88] conducted initial analyses on how to take advantage of common sub-expressions within subsequent queries. Finkelstein [Fin82] identified the requirements of a system that uses cached intermediate results to compute future queries and gave a formal description of common sub-expressions. He also presented a matching algorithm for select-project-join (SPJ) queries. Sellis [Sel88] identified a comprehensive list of parameters that influence the worth of an intermediate result. The worth states how useful materializing a particular result is. He defined several replacement algorithms based on some of these parameters and LRU[6].
Since those early studies, there have only been a few contributors to the topic:
Chen and Roussopoulos [CR93] presented the *ADMS Query Optimizer*, which caches intermediate and final results, matches incoming queries with the cached results and integrates these results in the optimization of the query plan. The authors assumed that intermediate results are already generated during query computation. All of these results are then submitted to the result cache. If the cache is full, a replacement policy chooses materialized results to evinct in favour of new intermediate results. The replacement policy is derived from those suggested by [Sel88]. The matching algorithm used by the *ADMS Query Optimizer* is similar to the one proposed by [Fin82]. Its capabilities are also limited to SPJ queries.

[6] least recently used

Shim *et al.* [SSV96, SSV99] introduced the *WATCHMAN* Cache Manager. Its original form ([SSV96]) was limited to caching final results only. It introduced novel admission and replacement policies. The replacement policy considers for each final result the average rate of (the last K) references, the size and the computation cost. Since reference information is only collected for materialized results, the average rate of reference is not available when admitting a result. Therefore, the authors only consider the cost and size of a result in their admission policy. The computation cost of a result is obtained from the estimates of the optimizer. When the cache is full, all materialized results are sorted in ascending order of worth and the candidates for eviction are selected in sort order. The new result is admitted if it has a higher worth than the replacement candidates. If only reference statistics for materialized results are collected, it may lead to a form of starvation. This is because a freshly submitted result has incomplete reference information and is therefore among the first candidates to be evicted. When the result is submitted again, its reference information has to be collected from scratch and hence it is likely to be evicted another time. The authors addressed the problem of starvation by retaining reference information of materialized results for a while after the results have been evicted from the cache.
In [SSV99], the authors extended *WATCHMAN* to incorporate data cubes. If the result of a query is not materialized and the query is in a canonical form, the algorithm splits the query into two, one query for generating the data cube associated with the query and one for using it. The cache is then checked to see if the data cube or one that is subsumed by it is already materialized. The one with the least estimated cost to evaluate the query is used. If there is no usable data cube materialized or the cost of using the selected data cube is higher than executing the query from scratch, the cache policies decide on materializing the final result or the data cube of the query and then either execute the original query, or the split queries.
Rao and Ross [RR98] focused on reusing intermediate results within a single query invocation. They analyzed correlated queries and described how to identify the sub-tree of a correlated query that is not dependent on outer references, rewrite it to reveal a maximal uncorrelated sub-tree and then materialize that sub-tree for further uses.
Kotidis and Roussopoulos [KR99] presented *DynaMat*. It dynamically materializes final results of data cube queries as materialized views and uses these results later on to answer incoming queries. It uses very similar admission and replacement policies to those described for *WATCHMAN*. The authors furthermore described their approach on updating materialized views.
Roy *et al.* [RRS+00] introduced *Exchequer* as an entirely different approach to the previously described work. *Exchequer* merges the selection of materialized results for answering the current query with the optimizer and uses the past k[7] distinct query trees as a representative workload to evaluate the usefulness of materialized results and intermediates from the current query. The optimizer compares various permutations of the query tree in terms of execution cost and selects the cheapest. If one of the permutations is already materialized, the minimum of the estimated execution cost using the materialized result and using base tables instead is assumed. Including permutations when deciding which query plan to execute unveils additional sharing possibilities compared to only using the optimal plan.

[7] default: k=10

The content of the cache is reevaluated with each query invocation. Candidates are results which are already materialized and intermediates from the optimized query tree of the current query. These candidates are benchmarked against the representative set of the past k distinct queries. The representative set is a single consolidated directed acyclic graph (DAG) that contains the query trees of the last k queries and all permutations of each tree created in the optimizer. Consolidated means that equivalent sub-trees of different queries are unified. The goal is to materialize the subset of the candidates that fits the space restriction of the cache and minimizes the execution time of all queries in the representative set. Their algorithm starts with an empty set of results to be cached. In each iteration, it adds the result from the candidate set which reduces the computation cost of the representative set the most to the cache. Materialized results chosen in previous iteration are assumed to already be materialized. When the size restriction of the cache is met or none of the remaining candidates decreases the execution time anymore, the new cache content is determined and the not selected materialized results are evicted and selected results from the current query are materialized. *Exchequer* uses multi-query optimization techniques which the authors developed in [RSSB00].
Tan *et al.* [TGO01] introduced the *Cache-on-Demand* framework. Instead of materializing intermediates from the current query in order to reuse them in an uncertain future, it focuses on the present where 'perfect' knowledge on what is happening in the system can be assumed. To do so, it examines the queries that are currently running in the system, determines the common sub-expressions between the current and the running queries and identifies the intermediates of running queries that can be reused for the current query. It then materializes the ones that will decrease the total execution time of the current query and the currently running queries and use the materialized results to answer that query.
Ivanova *et al.* [IKNG09] introduced the *Recycler*, a recycling architecture for MonetDB which uses materialized intermediate and final results to answer incoming queries. This approach is described in more detail in Section 1.3.2

1.3.1 Associated Areas

Multi-Query Optimization

Like recyling, multi-query optimization (MQO) also exploits common sub-expressions in query evaluation. However, it requires queries to be submitted in batches of several queries. In contrast to recycling, a multi-query optimizer therefore knows all queries in a batch before any of them is executed. Furthermore, for each intermediate result it considers for materialization, it knows the number of queries in the batch that will profit from that result. All queries in a batch are optimized together in order to find a globally optimal plan for the entire batch. Identifying potential sharing possibilities within a batch and exploiting them by reusing their results is part of that optimization process. The selection of intermediate results to share between queries is integrated into traditional query optimization. The intermediates in the batch that contribute the highest reduction in execution time for the entire batch while fitting the space restriction are selected for materialization. Multi-query optimizers usually do not keep materialized results between different query batches. A good algorithm and implementation for

multi-query optimization is presented in [RSSB00].
Existing recycling implementations optimize each query in isolation and then only consider the intermediate results of the generated query plan for materialization. MQO, however, generates a joint plan for the entire batch and globally optimizes it to find further possibilities of sharing intermediate results that would not have been identified otherwise. These possibilities are permutations of the query plan that were not optimal when optimizing them in isolation but are optimal when sharing one of its results with other queries in the batch. For example the join $A \bowtie B \bowtie C$ and the join $A \bowtie B \bowtie D$ could reuse the join of $A \bowtie B$, but if $(B \bowtie C) \bowtie A$ turns out to be the locally optimal plan for the first query (e. g. due to the cardinality of the result or indexes), there is no possibility for reusing intermediates anymore if only local plans are considered.

Materialized Views

Intermediates cached in recycling can be seen as dynamic materialized views. Traditionally, materialized views are specified by the database administrator using a typical workload (e.g. from the past), a space restriction and specialized tools to assist the selection process. As a result, they are static and do not adapt to changes in the workload unless the database administrator intervenes again. Cached intermediates in contrast are materialized and evicted dynamically and therefore adapt to changing workloads.
Despite this difference, materialized views face a similar set of challenges. They have to find the best set of views to materialize given a space restriction (*view selection problem*), automatically find the best materialized views to use for an incoming query (*view matching problem*) and finally deal with updates on the base relations that have to be propagated to the materialized views (*view maintenance*).
Since materialized view selection is usually performed offline, it has more time to find the best views to materialize than recycling which is bound to selecting intermediates from the current query while evaluating it. Materialized view selection uses the additional time for considering global optimal plans as well. This includes materializing views that are not part of any permutation of a query considered by the optimizer, but still contribute to a lower execution time of a given workload. Materializing a data cube could be an example for such a view. Previous recycling implementations did not exhaust these possibilities or were limited to data cubes.

1.3.2 A Recycler for MonetDB

The major influence of this work is a paper by Ivanova *et al.* ([IKNG09]). It describes the realization of a recycler architecture for MonetDB called the *Recycler*.
In MonetDB, intermediate results are materialized as a by-product of query execution. The *Recycler* submits all of these intermediates to the recycler cache. Even when the cache is full, they are still submitted. However, the *Recycler* then removes intermediates already in the cache to create the required space. It uses a worth metric to compare between materialized result as described for *WATCHMAN*. The results with the least worth are removed from the cache until enough space is created to store the new intermediates in.
In MonetDB, a query is evaluated by several instructions. The input and output of most

of these instructions are intermediate results. The *Recycler* matches these instructions in run-time in order to find out if there is an intermediate in the cache that could be used instead of reevaluating that instruction. The matching process requires the *Recycler* to keep the results of all instructions that the intermediate is dependent on in the cache. This means that all intermediates used to create that intermediate have to be stored as well in order to keep its lineage. When evicting intermediates from the cache, only intermediates that no other intermediate is dependent on are possible candidates.
The *Recycler* supports an advanced form of subsumption for selections. If there is no exactly matching selection in the recycler, it tries to find materialized results that are subsumed by the intermediate or several materialized results that can be combined to obtain the intermediate result from.
There are many differences between the *Recycler* for MonetDB and one for a pipelined DBMS. Most of them can be explained by architectural differences. The column-at-a-time paradigm forces MonetDB to materialize all intermediate results and to process each column separately. Because intermediate results are already materialized by the system, materialization cost is irrelevant for the *Recycler* and there is no need to decide which ones to materialize. This means that all materialized results can be submitted to the *Recycler*. As soon as the cache is full, the recycler decides which ones to keep in the cache and which ones to evict in order to create space for new intermediates.
In a pipelined DBMS, an operation usually processes all relevant columns of an intermediate result at the same time. Materializing the result of such an operation limits recycling to only being able to use a materialized result, if it contains all tuples of one of the intermediates produced by the query or if it can be combined with other materialized results to form a result that contains these tuples. The materialized result then is a superset of the intermediate. If the intermediate result contains either a tuple or a column that is not part of the materialized result, the result cannot be used.
MonetDB on the other hand processes each column separately. Therefore each intermediate result is a single column. In order to be able to reuse such a result, the materialized column has to contain all values of the intermediate column from the query. However, not all columns of the result of a relational operation from the query need to be contained in the recycler anymore. In order to reuse the result of a relational operation from the recycler, it is sufficient if the columns of that result partially overlap with the columns of the result from the query. As soon as at least one of the columns of a result in the query matched with the recycler, that result can be used to answer the query. Even if the matched columns in the *Recycler* originated from results of different query invocations, they can still be used together to answer the query. All columns which are not materialized in the recycler will be evaluated separately. Because all dependent results have to be kept in the cache as well, the columns do not need to be recomputed from scratch. A selection can for example just be applied to the additional column using intermediates from dependent instructions instead of recomputing the entire selection. The recycler for MonetDB therefore can reuse materialized results at a finer granularity and is able to exploit more common sub-expressions.

1.4 Contribution

Ivanova *et al.* [IKNG09] showed, that recycling intermediate results can be very useful when processing workloads which contain common sub-expressions. However, the used DBMS (MonetDB) has some unique characteristics that favour recycling. The most important difference to most commercial DBMS is, that it materializes intermediate results while executing a query. This means, that all intermediates are already available in main memory. Therefore materialization is free and the recycler only has to decide which intermediate results to keep. DBMS that rely on pipelining do not materialize intermediate results and therefore materialization has its cost. Dependent on the cost of the operation and the size of its result, materializing the result of an operation can cost several times as much as the operation itself. The cost of materializing every intermediate result would most likely be too high in a pipelined environment. Not having to materialize every intermediate result is one of the advantages of such a DBMS.
In order to use recycling in a DBMS that does not materialize every intermediate results, only some intermediates should be materialized while executing incoming queries. The ones being materialized have to be chosen carefully as the selection of these intermediate is the most critical decision in such an architecture. Intermediates that lead to the lowest computation cost for the entire workload have to be selected. The recycler will define an advanced worth function to compare intermediate results and to decide which of them to materialize. The worth will be dependent on the cost of an intermediate result, its predicted probability to reoccur in a future query and its size. The prediction of the probability of a result will be more comprehensive than in previous recycler architectures.
The authors of [IKNG09] also use some kind of worth to compare intermediate results. However, they only use it to evict materialized results from the cache. New intermediates are always added. As a consequence, the computation cost as well as the size of each intermediate result is already known when computing its worth. In contrast, a recycler for a pipelined DBMS has to select intermediates for materialization. This decision has to be done before the result is produced. Therefore, the recycler needs to compute the worth of an intermediate result before its cost and size is known. The recycler uses two sources to approximate unknown factors: *History and Sampling*.
The recycler collects information from previous query invocations in a structure called the *recycler tree*. This information includes for example the computation cost of intermediates that have already been executed. When having to compute the worth of an intermediate result which is already in the *recycler tree*, that information is available in the tree and can be utilized.
In case the intermediate result has not already occurred in a previous query, there is no historic information on it. However, the recycler still needs to be able to compute its worth to decide whether to materialize the result. In that case, sampling is used to estimate the cost and size of an intermediate. Before deciding to materialize a result, it is buffered for a while and from the information obtained while buffering it, the total cost and size of the intermediate is estimated. These estimates are then used to compute the worth of the intermediate result.
Furthermore, the recycler will explore the possibility of using light-weight compression schemes [ZHNB06] provided by VectorWise. Compressing materialized results shrinks

their size, but also increases the cost of materializing them. It allows to fit more materialized results in the limited main memory space. Compressing materialized results is novel to recycling architectures.

1.5 Recycling Architecture

The recycler interacts with traditional query processing. When a query is submitted and transformed into a query tree, the recycler tries to match each node of that tree with trees from previous queries already in the recycler. After matching is finished, it submits the tree of the current query to the recycler. Then it checks if there are nodes in the query tree that were matched with a node in the recycler that is already materialized. If such a node is found, it uses the materialized result to compute the query instead of recomputing the entire sub-tree rooted by the node from the query. The recycler then inserts *Store operators* into the resulting query tree. A *Store operator* is a pipelined execution operator defined for the recycler. When executing the query, each *Store operator* passes the result produced in its sub-tree to its parent operator. It can also materialize that result if needed. The aforementioned worth metric is used to decide if that particular result should be materialized. It utilizes historic information stored in the recycler as well as information gathered through sampling in order to make an informed decision. After the query is executed, the recycler adds each result of the query that was materialized to the recycler cache.
The following chapters will describe the architecture of a recycler for pipelined query evaluation in more detail. Chapter 3 will investigate the storage of intermediate and final results. Chapter 4 will give additional details on matching and the recycler tree. Chapter 5 will define the Store operator and the worth metric. Chapter 6 will then describe the process of using a materialized result to answer the current query. The presented recycler implementation will then be evaluated in Chapter 7. Finally, Chapter 8 will conclude the work and briefly discuss possible future enhancements of the recycler.

2 VectorWise Query Processing

In order to understand how the recycler works, it is essential to understand how VectorWise processes an incoming query. The VectorWise server receives a query which has already been optimized in algebraic form from a client like the INGRES front-end. The VectorWise server processes that query in several stages. Figure 2.1 shows the main stages in the life-cycle of a query. VectorWise parses the received algebraic representation of the query into a tree structure. This tree structure is then manipulated and annotated by the rewriter. The rewriter contains several rewrite rules. It processes one rule after the other by calling the rule on each node of the query tree. Each rule can be either executed top-down or bottom-up. Top-down means that processing starts with the root node and then proceeds towards the leaves whereas bottom-up describes the opposite direction. After all rules have been applied to the query tree, the builder transforms it into an operator tree. The result of the query is then produced by executing that operator tree. The execution stage is described in more detail in the next Section. After the result has been produced, it is sent to the client and the server waits for the next query.

2.1 The Execution Stage

The VectorWise execution engine [ZBNH05, BZN05] uses a variation of the widely used tuple-at-a-time volcano iterator model [Gra94]. It combines the advantages of the column-at-a-time paradigm used in MonetDB [Bon02] like fast tuple processing with little overhead per tuple with the advantage of the tuple-at-a-time paradigm that intermediate results do not need to be materialized and a lower response time for the first result tuples. Like in the volcano iterator model, each operator implements the open-next-close interface, but instead of single tuples, entire vectors of values are passed through the pipeline. This leads to tight loops that expose more instruction level par-

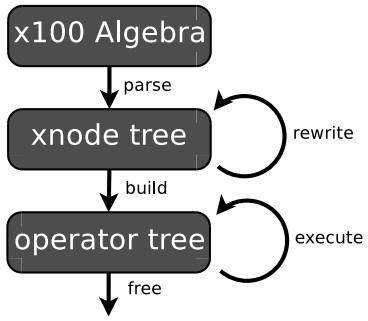

Figure 2.1: Lifecycle of a Query in VectorWise

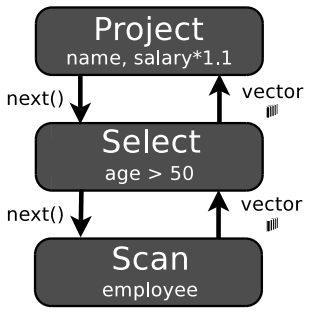

Figure 2.2: Pipelined Query Evaluation in VectorWise

allelism to the compiler and reduces the function call cost payed per tuple resulting in very high instructions per clock cycle (IPC). The vector size is typically 1024 tuples to ensure that all vectors currently in the processing pipeline fit into the CPU cache to provide faster access and fewer main memory reads.
To produce the result of a query, the server successively requests additional result tuples from the root operator of the operator tree. The root in turn requests tuples from its children and they from their children etc. As a consequence, the tuples flow from the leaf nodes towards the root. Each operator processes the tuples produced by its children and passes them on to its parent operator. The granularity tuples are returned in is a vector. Figure 2.2 illustrates the execution of a simple query in a pipelined DBMS. A segment of the operator tree is referred to as pipelined if all operators in that segment process tuples without having to see all data delivered by child operators in advance. An operator is called blocking if it requires the entire result of a child before it can produce any result tuples.

2.1.1 Some specialized Execution Operators

The Append Operator

The Append operator materializes the intermediate result produced by its child operator to an existing table stored in ColumnBM (CBM) [BZN05, ZBNH05, ZHNB06]. CBM is the storage manager of VectorWise and supports both, columnar storage (DSM[1] [CK85]) as well as PAX[2] [ADHS01]. It furthermore supports compression in either storage model. The definition of the table the result is stored in specifies the storage model used and whether the result should be compressed. The Append Operator furthermore passes the result to its parent to not interrupt the tuple flow in the operator tree.

The Reuse Operator

The Reuse operator is used to allow several operators in the same operator tree to share a result instead of having to recompute it for each operator. To do so, each produced tuple of that result is buffered until it has been consumed by each of them. When an

[1]Decomposition Storage Model - column-wise storage
[2]Partition Attributes Across - row-wise storage, but column-wise arrangement within each disk block

operator requests tuples from a Reuse operator, the request is satisfied using the shared buffers. Only if there are no new tuples in the buffers, the child is called to produce additional tuples. These tuples are then inserted into the buffers for all other consumers of the result.

3 Materializing Results

Since pipelined DBMS do not materialize all intermediate results, the recycler has to take care of the materialization process itself. This gives the recycler several options on how to materialize results. Materialized results could be stored in a simple in-memory structure organised by the recycler or in in-memory tables. The recycler will use the latter alternative. Although this alternative might expose additional overhead when materializing an intermediate result, it enables the recycler to use some of the already available features of tables like light-weight compression schemes. These can be used to shrink the size of intermediate results and therefore fit more of them in the limited main memory space.
To implement in-memory tables, a temporary columnspace[1] is defined. A temporary columnspace keeps all its blocks in main memory. It never writes blocks back to disk and never logs changes to the schema. Therefore the columnspace is not persistent. When the DBMS crashes or is shut down, the content of a temporary columnspace is lost. The recycler maintains a seperate catalog for all tables in that columnspace. That catalog is also not persistent and its content is reset with each server restart. All materialized intermediate results are added to that catalog instead of the system catalog.
The storage system of VectorWise offers several choices on how to set up a temporary columnspace and in-memory tables. These can be adapted to the needs of the recycler to improve its performance. The cost of materializing an intermediate result and the space the result occupies in main memory can be identified as the main factors. Both of them have to be kept as small as possible. However, they often behave in an opposing manner. The key is to find a good balance between both of them.
When creating the temporary columnspace, the block size of that columnspace has to be chosen. VectorWise usually uses huge blocks. The default block size is 512KB. Using larger blocks reduces the materialization cost. Figure 3.1 (l.) illustrates the decline in materialization cost when using larger blocks. The results were obtained by measuring the materialization cost of the aggregation (114003 tuples) in query Q3 of the TPC-H benchmark. However, many of the intermediates stored in the recycler tend to be rather small. Using huge block sizes would end up in unused space in blocks that contain results that are smaller than the chosen block size. The result of the first TPC-H query for example only contains 4 tuples. For all of the considered block sizes, this result will fit into a single block. The space it uses in the cache will be equal to the chosen block size. Therefore the recycler will favour smaller blocks.
When creating the table to store an intermediate result in, the recycler has several options. It can store each result either in DSM or PAX. DSM stores the result column-wise whereas PAX stores it row-wise regarding disk blocks. However, PAX internally arranges each block in a column-wise fashion. Figure 3.1 (r.) illustrates the cost of materializing the same result as before using either DSM or PAX. For all considered block sizes, DSM

[1] a tablespace for column data

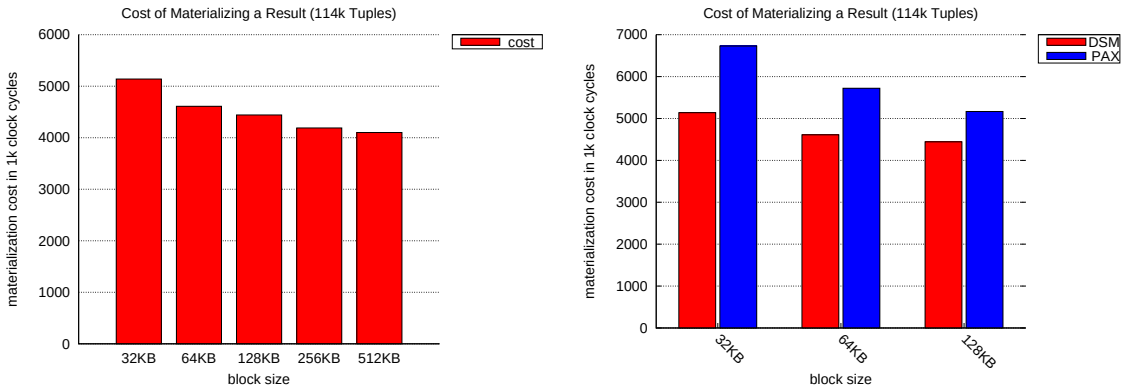

Figure 3.1: Benchmark of the materialization cost using various block sizes (l.) and storage types (r.) when compressing the result of the aggregation (114k tuples) from TPC-H Q3 with SF-10 and unclustered data

exhibits the smaller materialization cost. This suggests using DSM only. However, when considering small results again, DSM and PAX differ in the minimum space that a result requires. DSM requires at least one block for each column of the intermediate result. Using DSM for results that are too small to fill an entire block for each column, leads to unused space in these blocks and therefore to wasted storage space. PAX on the other hand only requires a single block as long as the result is no larger than that block. Since the decision whether to use DSM or PAX can be done for each table separately, the recycler will use PAX for small results. As soon as the estimated size of the result is big enough to fit a block for each column, DSM will be used instead.
Furthermore, the recycler can choose whether the result should be compressed. Compression increases the materialization cost but decreases the size of the result. Most of the additional cost of compression is accounted for by calibration. Before compressing a block, compression parameters are calibrated for that block using a fixed number of values that will be stored in that block. Examining these values includes sorting them and hence is very expensive. In VectorWise it is possible to adjust the number of values which are examined for each block. The default is 8192 values. This value is tuned for the default block size of 512KB. When using smaller blocks, the fraction of tupels from the result which are used for calibration increases as well as the materialization cost. Reducing the sampling size decreases the materialization cost, but also the quality of the chosen compression parameters. The final size of the materialized result can be used as an indicator of the quality of the compression. The smaller the result, the better the used compression parameters and the better the quality of the compression. Figure 3.2 shows the materialization cost and the size of two intermediate results for different sampling sizes. The results confirm that compression does not come for free. However, reducing the sample size leads to less of a difference while the quality of the compression stays quite stable. In the left graph of Figure 3.2, compression with the smallest sample size reduces the size of the result to 28% of the uncompressed size while costing 76% more. In the right graph, the result size was reduced to 54% and the materialization cost increased by 118%. The results do not provide a clear outlook on the usefulness of compression. It depends on the relative importance between size and materialization cost and the characteristic of each result like the achieved compression rate.

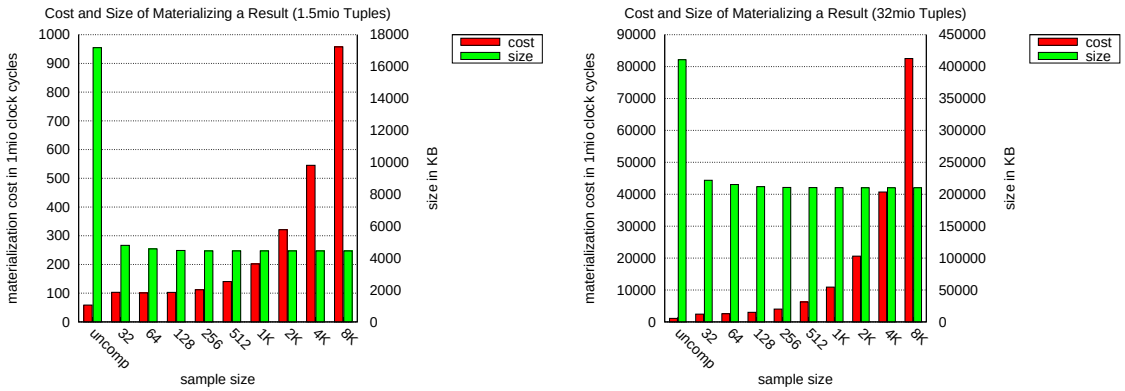

Figure 3.2: Benchmark of the materialization cost (l.) and size (r.) using various sampling sizes when compressing the result of the selection (32 mio tuples) on lineitem from TPC-H Q3 with SF-10 and unclustered data

Compression can be individually chosen for each column of each table.
To improve the situation for compression, the compression system could be changed. It could calibrate once for the entire result instead of calibrating for each block individually and only recalibrate when the exception rate surpasses a threshold. Furthermore, columns that do not have a high enough compression rate could be rejected for compression early and only blocks from columns with high compression rates are considered further.

4 Matching and the Recycler Tree

4.1 Introduction

For each materialized result stored in the recycler cache, the recycler has to be able to identify upcoming queries that can be answered using that particular result. A materialized result can be used to answer a query if the query tree contains a sub-tree that produces the same result. The materialized result can then be used instead of executing that sub-tree again. In order to check if the intermediate result which will be produced by the query and the materialized result from the recycler cache are the same, the (sub-)trees producing both results have to be matched. Two trees can be considered matching, if they produce the same result. Matching both trees requires the recycler to not only store the intermediate result, but also the tree which produced it. In order to find out if a materialized result can be used to answer a query, the query tree is matched bottom-up with the tree which produced the materialized result. The matching process starts with each leaf node until either the node is reached which produced the intermediate result or the matching is canceled because two corresponding nodes would have produced incompatible results.
Matching the intermediate results of a query with various materialized results in the recycler cache is performed by a bottom-up rewriter rule. The rule matches each node of the query tree separately with all candidate nodes in the recycler that still could produce the same result.
There are two sorts of matching nodes, exact match and subsumption. Exact match has been described earlier in this section. Two nodes match exactly if the trees rooted by them produce the same result. In this case, the materialized result can be unconditionally used to answer the query. An intermediate result of the query subsumes a materialized result, if the materialized result contains not only all the tuples of the intermediate, but also additional ones. In this case, the materialized intermediate can still be used to answer the query because it contains all needed tuples, but the result might have to be pre-processed before the query can use it instead of the original sub-tree. See Appendix A for a more detailed description of subsumption.

4.2 The Recycler Tree

As mentioned before, the query-tree which generated the materialized result has to be stored together with the result in the recycler. There are several options for how to store these trees in the recycler. Figure 4.1 illustrates three of them. Dark shaded nodes represent intermediate results which are materialized in the recycler. The first option has been assumed so far. Whenever a new result is submitted to the recycler, the part

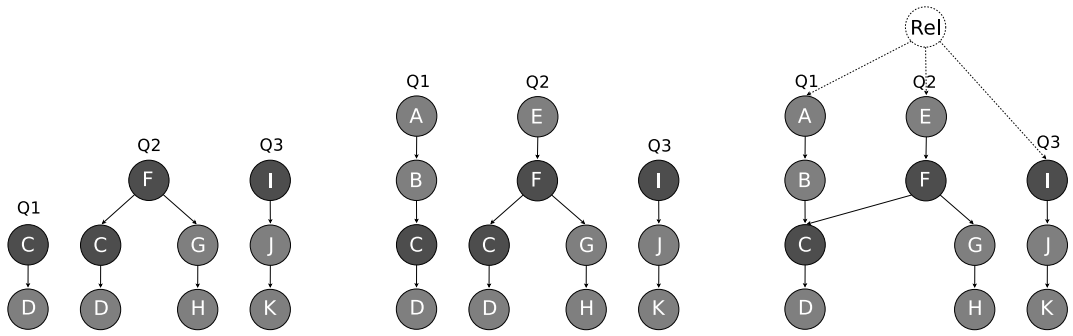

Figure 4.1: Possibilities for Storing Query Trees in the Recycler

of the query tree which produced that result is also submitted. A disadvantage of this approach is that statistics, such as the number of times a node was referenced by a query can only be collected for nodes stored in the recycler. As only materialized nodes and their sub-trees are stored in the recycler, only these have statistical information available. When deciding whether to materialize an intermediate result, the recycler cannot use any history based information to improve the decision because it only has incomplete statistics on past behaviours. Using only the information available would cause a freshly materialized node to be more likely to be evicted shortly after its submission than at a later time when sufficient information is collected. Another possible approach would be to store all query trees no matter if they contain materialized results. This option would allow collecting statistics on all nodes, but also expose additional overhead.
The recycler uses the third option in Figure 4.1. It stores all query trees, but links matching nodes together and adds a virtual root node to obtain a unified tree. Although the resulting tree is a directed acyclic graph (DAG), it is referred to as **recycler tree**. Node C in Figure 4.1 illustrates, how the copies of query trees from different queries share common nodes in the recycler tree.
Each incoming query tree is added to the recycler tree after matching is completed. Each node is added separately starting with the root node of the query tree. Nodes that do not match with a node in the recycler tree are copied to the recycler. For each new node, a structure to keep the statistics in is created and initialized. Furthermore, a pointer to the node's parent is added. This pointer will be needed to match the copied tree with upcoming query trees. Nodes that matched with a node in the recycler tree are not copied. Instead, they are linked to the node in the recycler tree that they were matched with. Once a matching sub-tree is identified, it is traversed in the recycler tree to update each node's statistics. The updating stops once a leaf node or a node which is already materialized is reached.

4.2.1 Linking common Sub-Trees

Whenever a node in the query tree does not match with any node in the recycler, but has a child that was successfully matched, that child is the root of a sub-tree that already exists in the recycler tree. Instead of also copying that sub-tree, the two sub-trees are linked together. This means that the node's child is replaced by the root of the corresponding sub-tree in the recycler. Linking common sub-trees ensures that such a

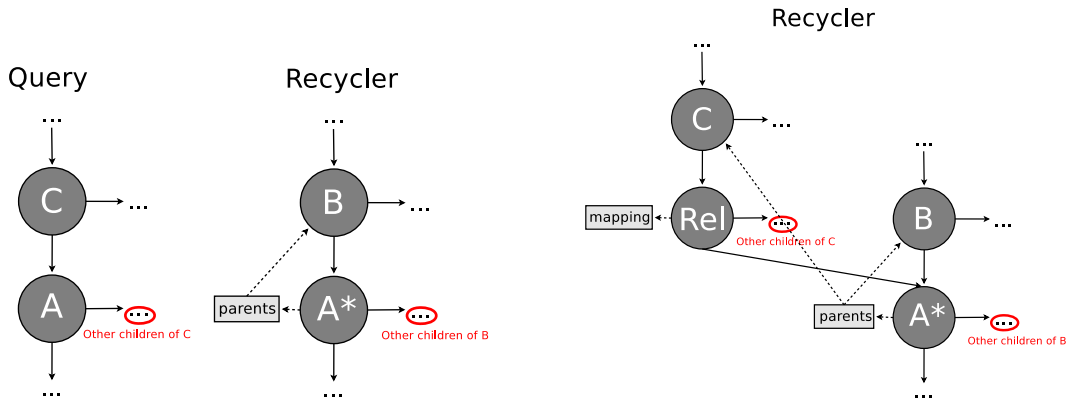

Figure 4.2: Linking an Incoming Query with a Matched Query from the Recycler

sub-tree is only present once in the recycler, no matter how many different queries it has been part of. This also means that each node of the sub-tree only has to be matched once for all of these queries. After linking, the root node of the shared sub-tree has more than one parent and each of its parents share a common child.
There are, however, problems associated with this approach, as shown in Figure 4.2 (l.). In VectorWise, each node only has one child pointer which points to a linked list that represents the node's children. Replacing one of these with the root of the shared sub-tree cuts off the node's remaining children and replaces them with the children of their original parent in the recycler tree. Replacing C's child A with the matched child $A*$ from the recycler tree also replaces C's other children with the children of B. Figure 4.2 (r.) shows a solution to that problem. When linking node C to the shared sub-tree, a virtual node (Rel) is added between C and the root node $A*$ of that sub-tree C. The virtual node Rel holds all remaining children of C. $A*$ is then added as the virtual node's only child. When adding the additional parent pointer to the root $A*$ of the shared sub-tree, it will point to C instead of the virtual node inbetween them. This ensures that the virtual node is not considered when matching, because the matching process navigates the recycler tree bottom-up through parent pointer and therefore just jumps over the virtual node.
However, the fact that two queries share a common sub-tree in the recycler tree does not mean that each of them uses the same names to identify columns. One query could for example have called the result of the same aggregation inside that sub-tree 'sum' and the other 'sum_qntty'. When passing a virtual node while matching, the names that were used in the shared sub-tree to identify results have to be replaced by the names the query that was linked to the sub-tree used to identify these results. If the query and the linked sub-tree use different identifiers for at least one of their columns, the virtual node has to provide a mapping that can be utilized to translate the names used in the common sub-tree to the ones used by the query. Since a mapping like this has already been created when the query was matched with the shared sub-tree, that mapping can be kept in the virtual node and used whenever a new query is successfully matched up to that node.

4.2.2 DAG Queries

Not only different queries can share common sub-trees. A single query can also use the same sub-tree more than once. The query tree of such a query forms a directed acyclic graph. Examples for such queries are queries 11, 13 or 15 of the TPC-H Benchmark. Shared sub-trees within a single query are already identified and exploited by VectorWise. VectorWise utilizes Reuse operators to only compute the resulting tuples of such a sub-tree once and then use these tuples for all further occurrences of the same sub-tree. In the query tree, a unique name is assigned to the first occurrence of a shared sub-tree. All other occurrences of that sub-tree are identified by that name instead of the sub-tree itself. Inserting such a query tree into the recycler tree requires the recycler to insert the common sub-tree only once and then link all other occurrences of the same sub-tree to that one. In contrast to linking sub-trees shared by different queries, no name mapping is required because the same sub-tree will always use the same names.
Furthermore, all Reuse operators have to be ignored when inserting the query tree to avoid having to match them later on.

4.2.3 Using the Recycler Tree

As mentioned before, the recycler tree is used to match incoming queries and to collect and utilize statistics on each of its nodes. The collected statistics include the measured cost for computing the result of each node, the number of times a result has been part of a query so far and, if known, the size of the result.
Some of the operations on the recycler tree require navigating the descendants of a node and some require navigating the ancestors. With each new query, the recycler tree grows in size. If the query matched with a sub-tree in the recycler, the root node of that sub-tree gets an additional parent and each node in its sub-tree gets additional ancestors. When navigating the ancestors of a node, the search space grows with each different query that uses that node. The number of descendants, however, is bounded by the individual size of a query and does not grow over time. Therefore, navigating descendants is preferable to navigating ancestors wherever possible.
Furthermore, the recycler tree has to be truncated from time to time to prevent it from becoming too large. Chapter 8 will outline how the tree could be kept to a manageable size.

4.3 Matching

4.3.1 Matching Leaf Nodes and Creating an Initial Mapping

Matching a query with the recycler tree always starts with matching leaf nodes, which are usually scans on base tables. At that point, all leaf nodes in the recycler tree are possible candidates. The number of these candidates depends on the size of the recycler tree. If the recycler tree is not truncated periodically, their number continuously increases over time. With each further matching step, the search space is drastically

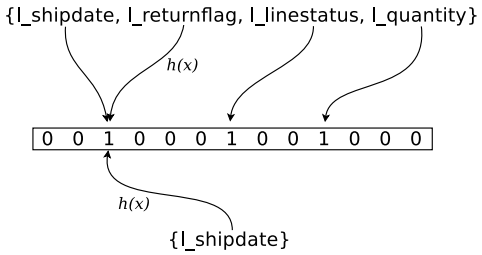

Figure 4.3: Using a Bloom Filter to test if a column is a member of a set of columns

reduced. The reason for this is that with each node in the recycler tree that did not match with the query, all of its ancestors are pruned. Since the recycler links matching nodes together, each node of the query tree usually has at most one matching node in the recycler tree. After successfully matching a node, the ancestors of the matched node in the recycler tree are the only candidate nodes for subsequent matching steps. When matching the node's parent in the next step, only the parents of the matched node have to be considered.

However, when matching table scans, all leaf nodes in the recycler tree are still possible candidates and have to be matched with each leaf node in the query tree. The effort spent on matching these nodes grows with the size of the recycler tree. In order to handle that effort, the number of leaf nodes in the recycler tree which are considered for matching has to be reduced as soon as possible.

A first step is to prune leaf nodes that do not use the same base tables. This is achieved by hashing each leaf node of the query on the name of the base table it scans from. The hash function indexes an array in the recycler that contains a pointer to a linked list of all leaf nodes in the recycler tree that hashed to the same value. Only leaf nodes contained in that list need to be considered further. Afterwards, the table name of the leaf node from the query is compared to each leaf node in that list.

Bloom filters can be utilized to further reduce the number of leaf nodes that need to be matched. If the table scan represented by a leaf node from the recycler does not contain all of the columns that are required by the query, that leaf node can be rejected early. Figure 4.3 illustrates how a bloom filter works. Each leaf node in the recycler tree contains a bit array. The name of each column which is scanned by a leaf node is hashed to one or more bits of that leaf node's bit array. These bits are then set. To check if all columns of a leaf node from the query are contained in a leaf node from the recycler, each of the column names from the query are tested if they hash to bits in the bit array of the node from the recycler that are already set. If one of the column names hashes to a bit that is not set, that column has not been scanned by the leaf node from the recycler and therefore does not need to be considered any further. If each column name hashes to bits in the bit array that were already set, it does not necessarily mean that both leaf nodes match as the bits could have been set by other columns as well. Using bloom filters only prunes additional leaf nodes from the list returned by the recycler, it does not avoid the more expensive matching process.

After pruning as many leaf nodes from the recycler as possible, the leaf node from the query tree is matched with all remaining nodes. Since table names have already been compared before, it is not necessary to match them again. The matching process checks

for each column which is scanned by the leaf node from the query tree, if that column was also scanned by the one from the recycler tree. If all columns of the leaf from the query tree are contained in the leaf of the recycler, the two nodes are considered matching.
However, each leaf node can assign a new name to the base table it scans from. The parent of that node then uses the new name to identify the table instead of the original one. When matching the parents of two scans which matched, but assigned different names to the same base table, the two nodes will identify the columns from these tables with different names although they refer to the same column. By the time the parents are matched, the information showing whether these columns had different names because they were from different tables or if they just were renamed is no longer available. In order to be able to match nodes that use different names for the same column, the recycler has to create mappings between these nodes. These mappings map each combination of column and table name used by a node from the query tree to the names which were used by the matched node in the recycler tree. If a node from the query is matched with a node in the recycler tree, the node from the query is annotated with the matched node as well as the mapping between both nodes. When matching that node's parent later on, the stored mapping can be used to try to match it with one of the matched node's parents. The mapping is updated with each matching step further up the tree.

4.3.2 Basic Matching

Matching an intermediate result with various materialized results in the recycler is performed by a rewriter rule. The rule matches each node in the query tree separately. It only considers some nodes of the query tree for matching. These nodes are referred to as relation nodes. Relation nodes represent a relational operation that will be performed on the results of its child relation nodes, like a selection or an aggregation. Relation nodes are characterized by another group of nodes in the query tree called expression nodes. Each node has a type and the rewriter rule applies different matching procedures dependent on the type of the relation node it is called for.
The matching procedure for most of these types has a similar structure. Algorithm 1 illustrates the matching process for an unary relation node. Matching always starts with checking if the node's children were successfully matched. If one of the node's children did not match with any node in the recycler, the node itself cannot match with any node and the matching process can be aborted. If the child of the current node matched, the node in the recycler which matched with the child as well as the name mapping between the child and the matched node can be obtained from the child. In a few cases, for example when matching the parent of a scan, it is possible to have more than one matching node and hence more than one mapping. If this is the case, the following procedure has to be repeated for each matching node. Candidates for matching are all parents of the node in the recycler which was matched with the child. Since the recycler linked matching nodes together, the number of parents can exceed one. Each parent of the matched node is then checked to see if it has the same type as the current node in the query. If the node which was matched with the current node's child and its parent originated

Algorithm 1 Match a node V_i

```
Input: V_i                                          /* node from the query tree */
  if child C of V_i matched then
    for all nodes M that matched with C do
      for all parents P of M do
        if V_i and P have same type then
          if child of P is a virtual node then
            copy mapping
            update mapping
          end if
          match relevant expression nodes and update mappings
          if relevant expression nodes matched then
            annotate V_i with the matched node and mapping
          end if
        end if
      end for
    end for
  end if
```

from different queries and therefore have a virtual node inbetween them as described in Section 4.2.1, the mapping is updated with the new names before continuing. Since the mapping which was obtained from the current node's child will also be used to match all not yet matched parents and is further needed when using a materialized result to answer the query, it has to stay intact and therefore all changes have to be performed on a copy.

The recycler then matches the relevant expression nodes that specify the parameters of the relation. Figure 4.4 illustrates matching the selection from query Q1 from the TPC-H benchmark with a selection in the recycler tree that originated from the same query but uses another table name. The mapping from the child node provides the mapping between the names used in both relations. The selection selects all tuples from lineitem with l_shipdate being greater or equal to a specific date. The column the selection is applied on and the value of the date are described by the expression nodes of the selection. Expression nodes are matched using a generic matching function regardless of the type of relation node that they describe. In each recursion, the matching function matches an expression node from the query with one from the recycler. When matching two expression nodes, the sub-trees of these nodes are evaluated first. If these sub-trees were successfully matched, the nodes themselves are matched. Expression nodes are matched differently depending on their type.

Figure 4.4 shows the expression nodes that describe the aforementioned selection and Algorithm 2 outlines the matching algorithm for the expression nodes which are relevant for that selection. The types of these nodes are *Operator*, *Ident*, *Qual*, *Call* and *String*. Matching expression nodes starts with matching the root node of the expression sub-tree from the query with the corresponding root node from the recycler tree. In the selection example, these are *Operator* nodes. An *Operator* node specifies the selection condition. When matching both nodes, it is checked if they have the same type and the same value. If that is the case, the children of both *Operator* nodes are matched. The children of

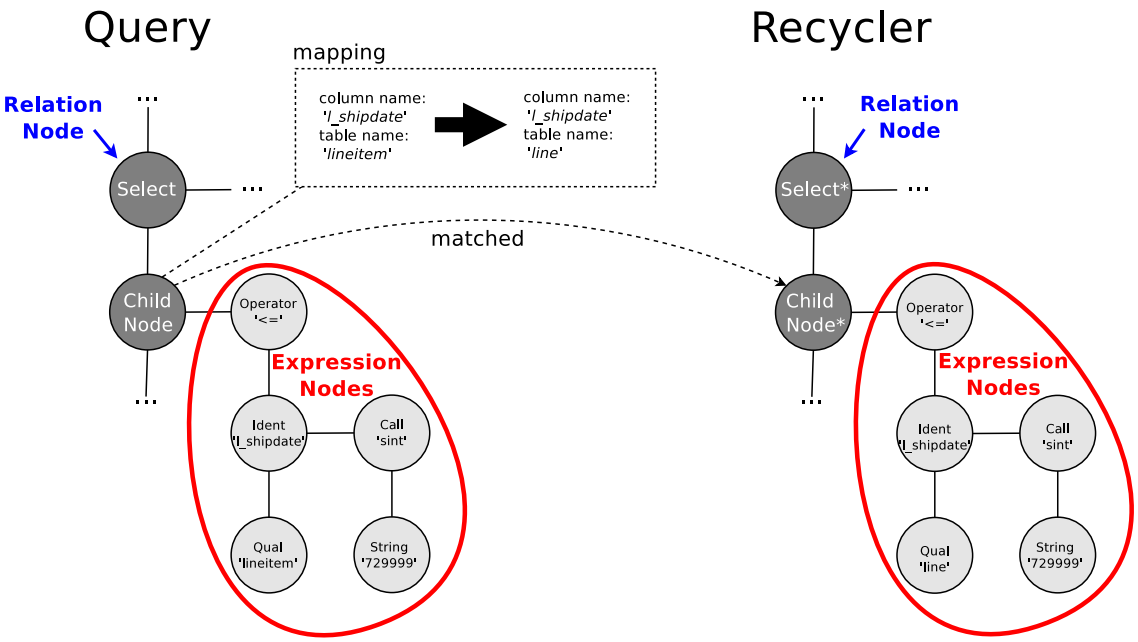

Figure 4.4: Matching the Selection from TPC-H Query 1

Algorithm 2 Match expression nodes of a node V_i

```
Input: expr1                                   /* expression node from the query */
Input: expr2                                   /* expression node from the recycler */
  if expr1 and expr2 have 'Operator' or 'Call' as type then
    if expr1 and expr2 have the same value then
      if children of expr1 and expr2 match then
        return true
      end if
    end if
  end if
  if expr1 and expr2 have 'Ident' as type then
    if expr1 and expr2 have the same value then
      if children of expr1 and expr2 have the same value then
        return true
      end if
    end if
  end if
  if expr1 and expr2 have 'String' as type then
    if expr1 and expr2 have the same value then
      return true
    end if
  end if
  return false
```

the *Operator* nodes are *Ident* and *Call*. The *Ident* node specifies the column name of the selection and the *Qual* node beneath it the corresponding table name. Both of them are matched in a single step. It is checked if both expression nodes have the same type and if a mapping exists that maps the column and table name used by the expression nodes from the query to the names used by the ones from the recycler. The *Call* node is matched the same way as the Operator node. The *String* node specifies the constant value of the selection predicate. Both *String* nodes are successfully matched if they have the same type and value. If all of these steps are successfull, the expression nodes of the selection can be considered matching.
If the expression nodes contain name assignments, the new names have to be added to the mapping which was obtained from the child. Since the mappings are added to the front of the list, the child's mapping is not changed and therefore it is not necessary to copy the mapping first.
If all of the previous steps have been successful, a matching node is found. That node and the corresponding mapping is then added to the node from the query.

4.3.3 Some Special Matching Cases

There are some relation nodes that have to be treated differently to what has been described in the previous section. These exceptions are described below:
Relation nodes that represent binary relations like joins have two children. Both of them need to match before the binary node can be matched. While matching the node, the mappings of both children have to be united.
Another special case are *As* nodes. They are used to assign a unified table name to the result of their sub-tree. Since these nodes only change the mapping, but not the result itself, they must always be successfully matched. When matching two nodes, both of them can be *As* nodes or either of them. If both of them are *As* nodes, the mapping of their child will be copied and then updated with the new table names. After that, both nodes will be considered successfully matched. If the node from the query is an *As* node, but the node it is matched with in the recycler has any other type than *As*, the mapping will again be copied and updated with the new table name. However, the *As* node from the query will not be considered matched with the node from the recycler. Instead it will be considered matched with that node's child. The child was already successfully matched with the child of the *As* node in a previous step. When the parent of the *As* is matched later on, it will again be matched with the same node in the recycler. However, it will use the mapping produced by the *As* node. If the node from the query has any other type than *As*, but the one from the recycler is an *As* node, the type of the parent of the *As* is checked. If the node from the query and the parent of the *As* node have the same type, the mapping is copied and updated with the new table name assigned by the *As* node and then the node from the query is matched with the parent of the *As* node.
The last special type of relation node are *Reuse* nodes as described in Section 4.2.2. When a query contains the same sub-tree more than once, a *Reuse Assign* node is used on top of the first occurrence of that sub-tree to assign a name to the sub-tree and when that sub-tree occurs again in the same query, the *Reuse Ident* node is used to identify the sub-tree with that name instead of the sub-tree itself. When matching an Assign node,

the name that is assigned to the sub-tree as well as a pointer to the *Reuse Assign* node has to be made globally available. The virtual root node of the query tree is accessible in each matching step and therefore used for this purpose. When a *Reuse Ident* node is matched, the name its sub-tree is identified with is looked up and then the corresponding Assign node is checked for a matched node in the recycler and a mapping. If the Assign node's child was successfully matched, the *Reuse Assign* as well as all *Reuse Idents* that use the same sub-tree are considered matched with the same node in the recycler.

4.3.4 Subsumption in the Recycler

The recycler as described above only supports a limited amount of subsumption. The only form it supports is column subsumption. This means that the result in the recycler includes not only the required result, but also other columns that the query did not request. However, the recycler is not limited to that form of subsumption. It is possible to implement individual matching functions for each operator that check for possible subsumption scenarios once exact matching has failed. For example, if exact matching a selection failed but the result of the selection in the recycler tree is materialized, the rewriter rule could check if the node from the query and the one from the recycler use the same column in their selection condition. If that is the case, it has to check whether the selection condition described by the Operator node could subsume the one from the recycler and if the specified value is within the required range. If these conditions are met, the materialized result can be used to answer the query. However the selection still has to be performed on the materialized result in order to select the required tuples. After subsuming a materialized result from the recycler, matching of the node's ancestors must be stopped. This is because there are some operations, like aggregation, where it is no longer possible to get the results for the subsumption from.
Appendix A will describe subsumption in more detail.

5 Designing the Store Operator

5.1 Creating the basic Operator

The Store operator is a query execution operator that implements the iterator interface. Its purpose is to decide if the intermediate result produced by its sub-tree should be materialized to a new in-memory table and if so, to create that table and append the result to it. For results that are not being materialized, the Store operator needs to be as light-weight as possible. In both cases, the Store operator should not interrupt the tuple flow in the operator tree. The Store operator uses a worth function to decide which intermediate results to materialize. To obtain the worth of an intermediate result, information gathered from previous query evaluations as well as run-time measurements to estimate unknown parameters are used. A rewriter rule adds Store operators on top of any operator that produces a new intermediate result. While executing the query, Store operators dynamically decide at run-time which of these results are worth materializing.
An intuitive approach on how to implement the Store operator would be to just insert an Append operator between the Store and its child. This approach would allow materializing the result and removing the Append operator when deciding to bail out materialization. This would, however, require each Store operator to create a table and allocate memory for the Append operator before the query is executed. The Append would then start appending tuples as soon as the first ones are returned from the child. If, while producing the intermediate result, it turned out not to be worth materializing, the materialization process has to be aborted and the table dropped. The overhead of creating a table and starting to append to it has to be paid for each Store operator, even if the result is not being materialized. Especially for short-running queries, that overhead can become a significant factor. Furthermore, removing the Append operator after deciding not to materialize an intermediate result would corrupt the tuple flow in the operator tree. So far, the input of the Store's parent were the tuples produced by the Append operator. After the Append is removed from the operator tree, the parent still tries to get the next tuples from the Append instead of obtaining them through the child. Solving this problem would require additional memory allocations and logic in the Store operator.
A much cleaner approach would be to first buffer all incoming tuples for a while and then decide on materializing the result and only create the table and append to it if the result was deemed worthy enough to be materialized. The existing Reuse operator can be used in order to buffer these result tuples. Once a decision is made, the Append operator is used to store the result to a table. To provide a consistent view on the location of the produced tuples to the Store's parent, the Reuse operator is not added between the Store and its child. Instead the Store operator contains a separate pointer to the Reuse. As a consequence, from the perspective of the parent of the Store operator, the structure of the operator tree does not change and it can always obtain its input tuples

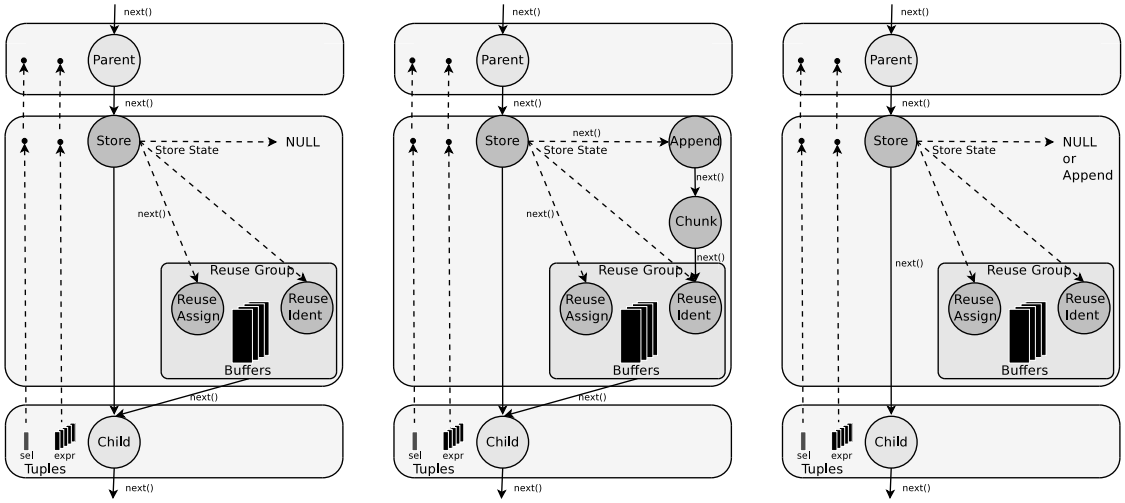

Figure 5.1: The Three Stages of the Store Operator: Buffer (l.), Append (m.) & Canceled (r.)

directly through the child.

5.2 The three Stages of the Store Operator

The Store operator has three possible stages: *Buffer, Append, Canceled*. The transition between these stages depends on the decision on materializing that specific intermediate result. Figure 5.1 shows the structure of the Store operator in all three stages. Algorithm 3 illustrates the next() calls within the Store operator's next() function.
The **Buffer** stage is the initial stage of all Store operators. In the Buffer stage, the Store operator buffers all tuples produced by its child operator until there is enough information available to make an informed decision on materializing the child's result. It utilizes the Reuse operator to buffer these tuples. Instead of directly requesting the next tuples from its child, the Store requests them through the Reuse operator. To keep the pipeline in order, it has to make sure that the Reuse in turn requests tuples from the child, instead of just using its buffers. Requesting tuples through the Reuse operator results in copying all tuples produced by the child to the buffers of the Reuse operator.
The **Append** stage follows the Buffer stage in case the intermediate result was decided to be materialized. In the Append stage, the Store operator creates a table to store the result of its child in and then allocates the Append operator. The Append uses another Reuse operator to obtain the already produced results from the buffers and store them into the table. In case the result is to be compressed, it also handles the compression. In the Append stage, next result tuples are requested from the child operator as described for the Buffer stage. However, as soon as there are enough buffered tuples available for appending, the Append is called by the Store operator to append these tuples to the table. Once a tuple has been consumed by the Store and the Append operator, that tuple does not need to be buffered anymore and can be overwritten by subsequent tuples. When the child has finished producing its result, all tuples in the buffers which have not been appended yet, are appended. Since the Append only consumes already produced

Algorithm 3 Next() Function of the Store Operator

```
if STAGE_CANCELED then
  request next tuples from child operator
else if STAGE_BUFFER || STAGE_APPEND then
  repeat
    request next tuples from Reuse operator
  until Reuse operator requested next tuples from child
end if
if STAGE_APPEND then
  if Append operator not allocated yet then
    create table and allocate Append operator
  end if
  if a full vector of new tuples is available in the buffers then
    append the next vector
  end if
  if child finished producing tuples but Append has not yet then
    while still tuples not materialized do
      append the next vector
    end while
  end if
end if
return  number of tuples produced by child operator
```

tuples stored in the buffers and never requests tuples from the child itself, there is no further synchronisation needed between the Store and the Append operator.
The last stage is the **Canceled** stage. This stage either follows the Buffer stage after the result was considered not being worthy enough for materialization or after the Append stage, when the Store operator decides that the first estimates for the result were not accurate enough and chooses to bail out materialization. After the Canceled stage has been reached, the Store operator can not change back to another stage anymore. The Store operator does not do any more processing once it reached the Canceled stage. It requests the next tuples directly from its child and passes the amount of tuples produced to its parent. It no longer uses the Reuse or Append operators. This stage is a lot faster than both of the other stages and is intended to be the usual case for most Store operators and should be reached as fast as possible.

Figure 5.2 compares execution times of the several stages and gives more of an insight on which parts of the Store operator the time was spent on. The data for the graphs was obtained from benchmarking a Store operator on top of the aggregation of query Q1 (4 tuples, l.) and a Store operator on top of the aggregation of query Q3 (114k tuples, r.) of the TPC-H benchmark. The scale factor used was 10. The big difference between the buffer and append stage is particularly striking. It shows the overhead of copying the entire result to tables instead of just buffering it. Most of that overhead can be explained by additional memory allocation and copying. Using in-memory tables instead of a simpler in-memory structure like these buffers was an early design choice for the recycler in order to get access to some of their features like compression.

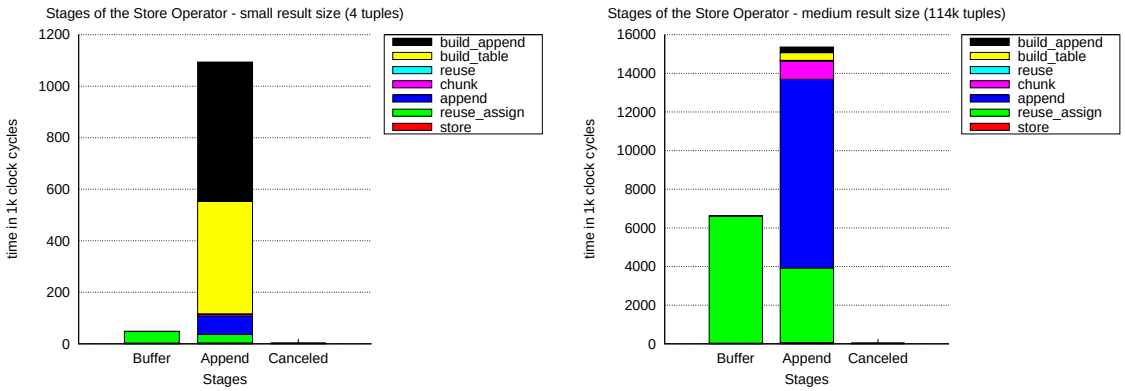

Figure 5.2: Stages Benchmark: The cost of various parts of the Store operator in each stage for a result with 4 tuples (l.) and one with 114k tuples (r.)

5.3 Introducing Progress Indicators into the Pipeline

To determine if an intermediate result should be materialized, the Store operator needs information on that result, like its computational cost and size. However, this information is not available until the computation of the result is finished. Since the Store operator cannot wait for that to happen, it needs to use estimates. To obtain these estimates, the Store operator must be aware of its current progress.

Previous work: Progress indicators have been introduced by [LNEW04, CNR04]. The proposed purpose was to inform the user about the progress of a long running query. The recycler uses a similar technique to calculate the progress of each operator while executing a query. The progress is used to estimate the final value of measured run-time parameters like cardinality or computational cost.

Segments of an operator tree that are pipelined have the same progress. These segments either start with a table scan or a blocking operator and end with the root or the child of a blocking operator. Partially blocking operators like hash join end the segment of the child used for building the hash table, but continue the segment of the child used for probing. Figure 5.3 shows an operator tree divided into such segments.

The described approach differs from the one presented by [LNEW04, CNR04] in the granularity the progress is determined for. The Store operator only needs the progress of each operator in the currently executing segment, whereas the approach by [LNEW04, CNR04] requires the progress of the entire query. Calculating the progress of a currently active segment can be done by only using information that can be collected while executing that segment. There is no information available for segments that have not produced any results yet. Since the progress of the entire query also contains these segments, their progress has to be estimated. The authors of [LNEW04, CNR04] combine measurements from already executing segments with estimates from the optimizer for not yet executed segments to obtain the progress of the entire query. This applies to the progress in terms of tuples processed as well as to the progress in terms of computation time. When referring to progress in the following, it always refers to the percentage of

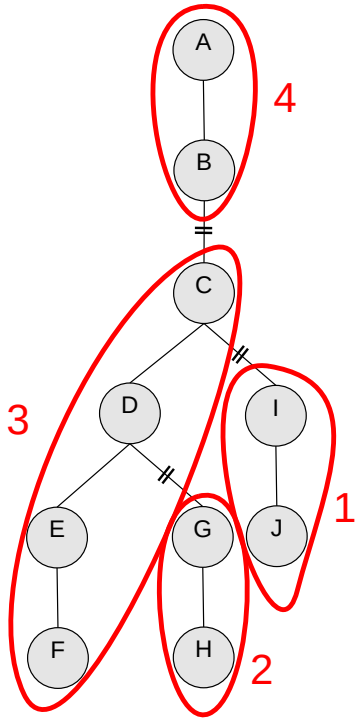

Figure 5.3: Segmentation of an Operator Tree through Blocking Operators B, C, D

tuples processed in a segment.

Trading progress: The leaf node of each of these segments is usually aware of its progress before it starts returning tuples to the other operators in its section. A table scan knows the total amount of tuples it has to scan as well as the amount it already has processed. A blocking operator like aggregation or sort requests all tuples from its child before it starts returning the first ones. Therefore, it also knows how many tuples it has received from the child and now has to process, as well as how many of them it already has processed. Since all other operators in the same section have the same progress, they can obtain their progress from the leaf node.

In order to implement progress indicators for an operator, two new fields have to be introduced to the operator structure: *progressed* and *total*. Progressed is the number of tuples that have so far been processed by the leaf node of the section and total is the amount of tuples that leaf node has to process in total. The progress of a node is then:

$$progress = progressed/total \tag{5.1}$$

The parameters that will be needed for the worth calculation of an intermediate result as described in the next chapter are the size of the result and the computational cost of that result. In order to obtain the size of a result, its cardinality needs to be estimated.

Cardinality and width: Estimating the cardinality of an operator requires the operator to keep track of the number of tuples it has produced so far. The progress of an operator can then be used to extrapolate the total cardinality of a result from the amount of tuples that operator has produced so far. The final size of an intermediate result can be derived from its total cardinality by multiplying it with its width. The width is the average space needed to store a tuple of that result. Section 5.4.2 discusses

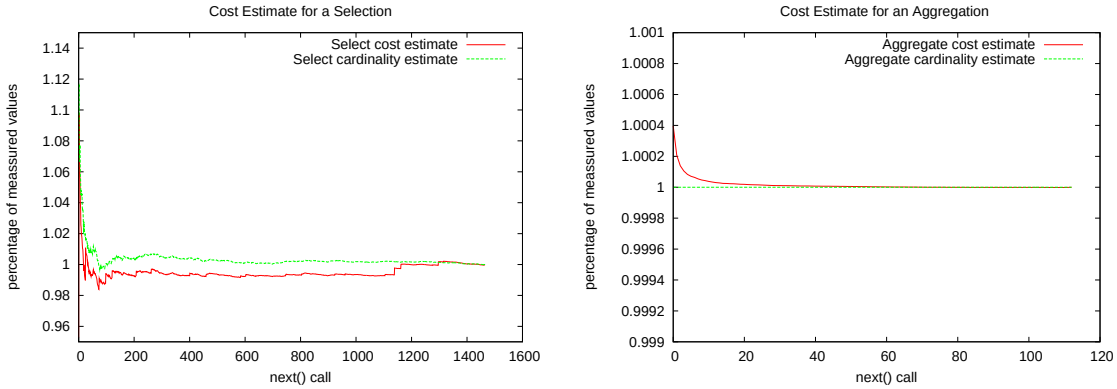

Figure 5.4: Cardinality and Computation Cost Estimates for each next() call of the selection on customer (l.) and the aggregation (r.) from TPC-H Q3

the size of an intermediate result in more detail.

Cost: Estimating the total computational cost of an operator requires the time the operator has so far spent on producing its result to be measured. This cost also includes the time the operator spent waiting for the child to produce tuples. Therefore the current cost of an operator is the cost the operator spent on producing its result so far subtracted by the fraction of that time the operator spent waiting for its child operators. The total computational cost can again be extrapolated from the current cost using the operator's progress.

Worth: The run-time estimates needed for the worth computation can be defined as follows:

$$estimated\ cardinality = produced\ tuples * (total/progressed) \tag{5.2}$$

$$estimated\ size = estimated\ cardinality * width \tag{5.3}$$

$$estimated\ cost = current\ cost * (total/progressed) \tag{5.4}$$

Amortization: Some operators spend a significantly larger amount of time on the first or an early next() call than on the remaining ones. This is especially the case for the building phase of a blocking operator. A hash join, for example, builds the entire hash table in the first next() call and sometimes creates bit filters to faster reject not included tuples in another next() call. Estimating the cost of such an operator results in an estimate that is significantly too high. To correct the estimate, all cost that is just spent once instead of with each next() call has to be measured and only included once in the estimate. To keep track of that additional cost, another new field has to be introduced to the operator structure: *amortize*. Each time an operator does work in a next() call that is only done once and therefore needs to be amortized, it measures that cost and adds it to *amortize*.

Equation 5.4 can be adapted to exclude only once paid cost from the extrapolation:

$$estimated\ cost = ((current\ cost - amortize) * (total/progressed)) + amortize \tag{5.5}$$

Microbenchmarks: Figure 5.4 shows the variation of the estimated cardinality and the estimated cost of a selection and an aggregation from their respective final values per next() call. The results were obtained by running query Q3 from the TPC-H benchmark

suite on a dataset with a scale factor of 10. The selection on customer was used for the first graph and the aggregation in the same query for the second. Both estimates were tracked after each next() call. Both graphs show that the cardinality and cost estimates are already after a few next() calls within a 2% range of the final result. In both graphs the first next call shows the highest cost estimate. The estimate then decreases with each next call until it stabilizes. For the selection, that bump can be explained by the fact that the sample size was too small to reflect the final selectivity of the result. The bump in the aggregation suggests that there might be instruction and data cache effects in the first next() call that then need some time to be amortized.

Total Cost: The total cost of the sub-tree of an operator can also be estimated using run-time estimates. However, some of the descendants of that operator are already finished and do not need their cost estimated. The estimated cost of a node and all its descendants is the sum of the estimated cost of that node and all descendants in the same segment and the measured cost of all other descendants. In order to estimate the cost of the node and all descendants in the same segment, the cost of all Store operators in that segment must be subtracted before extrapolating. Furthermore instead of the operator's amortized cost, the amortized cost of the entire segment has to be used.

5.4 Defining the Worth of an Intermediate Result

When deciding which intermediate results of the current query to materialize and if necessary which intermediate from the recycler cache to evict in favor of the new one, the **worth** can be used to differentiate between these results. The worth of an intermediate result $w(IR)$ can be defined as the gain in overall computation time when materializing that result while it is being produced by the current query and later reusing it for the computation of future queries. As a consequence, a positive worth suggests materializing a result and reusing it to compute subsequent queries, whereas a negative worth favors recomputing the entire query each time it occurs. When an intermediate result is submitted into the recycler cache, it initially has to pay the cost of materializing it. Each time it is used to answer a query, it gains from just having to read the result from the recycler cache instead of recomputing it. If the worth is positive, there will be a point in time where the cost saved by using the materialized result will exceed the materialization cost. The worth of an intermediate can be defined as follows:

$$w(IR) = ((cost_{comp}(IR) - cost_{read}(IR)) * num\_uses) - cost_{mat}(IR) \tag{5.6}$$

This initial definition assumes that all intermediates with a positive worth can be stored in the recycler cache and stay there until they are not needed anymore. In the more realistic case of restricted main memory space, not all intermediate results with a positive worth can be stored in the recycler and materialized results only reside there for a limited amount of time until they are evicted to adapt the recycler content to changing workloads. Since an intermediate result can only be used to answer a query while it resides in the recycler cache, the number of future uses of the result is not the total amount anymore, but just the uses while it is materialized in the recycler pool. Further-

more, when deciding which of these intermediates to keep in the recycler, the total worth of all intermediates in the cache has to be maximized. In order to be able to compare the worth of intermediates of different sizes, the worth of each intermediate has to be normalized by its size. Choosing intermediates with the highest worth per space unit improves the usage of the restricted memory space. The definition of the normalized worth is shown below:

$$w(IR) = \frac{((cost_{comp}(IR) - cost_{read}(IR)) * num\_uses) - cost_{mat}(IR)}{size(IR)} \tag{5.7}$$

The worth of an intermediate result changes over time. Once the result has been materialized, the result does not have to pay the materialization cost anymore and can only gain from further uses.
So far, the formula has assumed perfect knowledge of the costs as well as of future occurrences of queries using the materialized result. However, when the recycler needs to make a decision on materializing a result, this knowledge is not available. Especially the number of future uses will not be known until it is already too late to use them. The next sections will adjust the worth definition to the information that is available or can already be estimated when the recycler needs to make a decision on materializing the intermediate result.

5.4.1 Materialization and Reading Cost

The cost of writing an intermediate result to the recycler as well as the cost of reading it from the recycler is dependent on the size of the result. All other fractions of these costs can be assumed to be constant. This assumption leads to the following variation of the normalized worth defined in Equation 5.7:

$$\begin{aligned} w(IR) = & \frac{((cost_{comp}(IR) * num\_uses}{size(IR)} \\ & - \frac{\cancel{size(IR)} * const_{read} * num\_uses}{\cancel{size(IR)}} \\ & - \frac{const_{mat} * \cancel{size(IR)}}{\cancel{size(IR)}} \end{aligned} \tag{5.8}$$

The formula now consists of three parts: the total computational cost per space unit, the total reading cost per space unit and the materialization cost per space unit. Using the assumption from above, the materialization cost per space unit is a constant factor for each intermediate. When comparing two intermediates, this factor has no influence on the decision and does not need to be considered any further. However, results which have already been materialized have no materialization costs. When comparing a result that has not yet been materialized with one that has, the materialization cost still matters and should be used for a correct comparison of these results. The materialization cost equals the cost of the Store operator in its Append stage. Since each Store operator has to decide on materializing a result before it reaches the Append stage, the materialization cost cannot be estimated using dynamic cost estimates. It could be estimated

using the afore mentioned constant, but that constant would be system dependent. In its current state, the recycler does not use the materialization cost for computing the worth of a result. Ignoring the materialization cost leads to a more aggressive admission policy where an intermediate that saves more computational cost than another is always favored, even if that means replacing an already materialized result with one which is only marginally better.
The reading cost per space unit still contains a non-constant factor. However, that cost is typically only marginal compared to the cost of recomputing the entire result from scratch and therefore will also not be considered any further.
This results in a simplified worth definition:

$$w(IR) = \frac{cost_{comp}(IR) * num\_uses}{size(IR)} \tag{5.9}$$

5.4.2 Size of an Intermediate

As mentioned before, the size is used to normalize the worth to a unit of space in the recycler. The size of an intermediate result can be estimated using the estimated cardinality as described in Section 5.3 and multiplying it with the width of the result.

$$size(IR) = card(IR) * width(IR) \tag{5.10}$$

The width of a result is the sum of the widths of all columns in that result. The width of a column can be obtained from the vector representing that column in the Store operator.
For string columns, it is more complicated because the vector representing it does not contain the strings themselves, but only pointers to their location. To estimate the width of a string column, the SQL type of the column could be used to obtain the maximum number of characters allowed in it. That number could then be used to compute an upper bound of the column's width. Another approach would be to use the average width of a small sample of these string values as an estimate of the width of a column. For example the Store operator could compute the average width of all string values in the column's first result vector and then use that width for all size estimations of the operator. The recycler uses the latter approach.
All size estimates refer to storing the intermediate result in an uncompressed table. When using compression, the recycler furthermore has to estimate the compression ratio of each column. Alternatively, it could use constants for each column type and use them to estimate the final size of a result.
After an intermediate result has been materialized, its actual size is known and can be used instead of estimating. The recycler stores the size of an intermediate in the statistics of the corresponding node in the recycler tree. When the materialized result is evicted from the cache, the recycler still has its result size and can use that size to calculate the intermediate's worth the next time the same node is referenced by a query. Furthermore, the recycler stores the final cardinality and the estimated width of each

intermediate result in the recycler tree. When the same result is executed again, the size of the result can already be computed before the query started executing.

5.4.3 Cost to Compute an Intermediate

The **total cost** $C(V_i)$ of an intermediate result is the cost spent in computing that result from base tables only. It grows with the height of the intermediate in the query tree and the final result of a query always has the highest total cost. It can be estimated in the Store operator while computing the result as described in Section 5.3. As soon as the computation of the result finished, its exact value is known. This value is then stored in the statistics structure of the corresponding node in the recycler tree. The total cost of each intermediate result of a query is stored in the recycler tree, even if that intermediate was not materialized. When the same result is recomputed by a later query, the actual cost can be read from the recycler tree and then used instead of the estimate. The stored total cost of a result is always replaced by a newer measurement of it to reflect the most current cost of that result. When the system load or memory usage change, the total cost of an intermediate result adapts to the changed conditions after the next execution of that intermediate is finished.

Interaction between Materialized Results

When materializing an intermediate result, the next time that result is part of a query, its result is used instead of recomputing it from base tables. Therefore the computation cost of an intermediate result is influenced by already materialized results in its descendant tree. Materialized descendants decrease the computation cost of an intermediate result. As a consequence, the cost measured in the Store operator is no longer the total cost. However, the total cost can still be obtained from the measured cost C' as shown below:

$$C(V_i) = C' + \sum_{\substack{j \in \text{used} \\ \text{materialized} \\ \text{results}}} (C(V_j) - cost_{read}(V_j)) \tag{5.11}$$

When computing an intermediate result, not all materialized results of its descendants are used. Only materialized results are used which are not descendant of another materialized result. All usable materialized results can be found using a depth first search from the intermediate that needs to be computed. The search does not continue with the sub-tree of a node whose result is already materialized. These descendants of an intermediate result will now be refered to as 'direct materialized descendants' (DMD). Likewise will materialized ancestors of an intermediate that do not have any descendants materialized between them and the intermediate be called 'direct materialized ancestors' (DMA). If, for example, the nodes F, G and J in figure 5.5 were materialized, F would be the only direct materialized descendant of E and hence Q2 would be answered using the materialized result of F only. F would be the direct materialized ancestor of G and J.

When comparing the computation cost of intermediate results, the cost reduction due to direct materialized descendants has to be considered as well. Reducing the total cost by the fraction of the cost which is contributed by the sub-tree of all direct materialized descendants leads to the definition of **relative cost** $R(V_i)$. The relative cost is the cost to compute an intermediate result using its direct materialized ancestors and assuming that they can be directly used without having to scan them first. The relative cost is used to compute the worth of an intermediate result. It can be computed from the total cost $C(V_i)$ as shown below:

$$R(V_i) = C(V_i) - \sum_{\substack{j \in \text{direct} \\ \text{materialized} \\ \text{descendants}}} C(V_j) \tag{5.12}$$

Adding or removing a materialized result from the recycler changes the relative cost of all of its direct materialized ancestors in the recycler tree. The cost of finding these ancestors increases with a growing recycler tree because the search space increases with each parent added to the node or one of its ancestors. The cost for navigating all decendants of a node in the recycler tree is bounded by the individual size of the query and therefore does not grow over time. As a consequence, the relative cost of an intermediate result is not stored in the statistics of the corresponding node in the recycler tree, but instead recomputed from the node's total costs whenever it is needed. Recomputing the relative cost only requires navigating its descendants as shown in Equation 5.12.

Algorithm 4 Computing $R(V_i)$ for a (known) Intermediate Result

```
Input: V_i                                  /* corresponding node in the recycler tree */
  rcost ← total cost of V_i
  for all children V_j of V_i do
    rcost ← rcost − DMA_TCOST(V_j)
  end for
  return rcost

  function DMA_TCOST(V_i) do
    if V_i is materialized then
      return total cost of V_i
    else
      sum ← 0
      for all children V_j of V_i do
        sum ← sum + DMA_TCOST(V_j)
      end for
      return sum
    end if
  end function
```

Algorithm 4 illustrates how the relative cost is computed from the total costs annotated in the recycler tree.
If the worth has to be computed for an intermediate result that has not yet been executed and hence does not have its total cost stored in the recycler tree, the relative cost

can be estimated using dynamic cost estimates:

$$R(V_i) = C' - \sum_{\substack{j \in \text{used} \\ \text{materialized} \\ \text{results}}} cost_{read}(V_j) \tag{5.13}$$

Descendants which are currently being materialized have to be treated like already materialized nodes. In Equations 5.12 and 5.13, their cost has to be subtracted just like the cost of already materialized descendants.

5.4.4 Number of Future Uses of an Intermediate

It is not possible to know the number of future uses of an intermediate result without knowing the exact future workload. Since future workloads are usually not known, we must try to compute the probability that an intermediate result will be used by a future query. Two indicators can be identified that can be used to predict that probability. The behaviour of previous queries is often a good approximation of what will happen in the future [IKNG09, RRS+00, SSV96]. If an intermediate result was often computed to answer queries in the past, it is likely that it will be used to answer future queries as well. The other indicator is the specificity of an intermediate result. The more specific the result, the less likely it is, that the result will be used by a future query. If there is historic information available, the probability obtained from that information should be favoured to that from specificity.

Recycler Tree Analysis

Since the recycler stores query trees from all incoming queries, links matching parts together and stores statistics for each node, there is a lot of historic information available to extract the probability of a future use of an intermediate result from. Figure 5.5 shows a fictitious recycler tree to illustrate how to obtain probabilities from it. The information collected within the recycler tree can be interpreted in various ways. A first approach would be to assume that the recycler contains the entire workload and that that workload will not change in the future. Therefore all upcoming queries will be already known to the recycler and each node in the recycler tree will only be used by queries it is already part of. The more often a node was part of a query in the past, the more likely it is to also be part of an upcoming query. This concludes that the probability $P(V_i)$ of such a node is the number of times it has been part of a query (R_i) divided by the total number of queries (N): $P(V_i) = \frac{R_i}{N}$. The number N of total queries is stored in the recycler structure and the number of past references R_i is stored in the node's statistics. Each time a query tree is integrated into the recycler tree, the total number of queries is increased as well as the number of references to each node in the query. The example recycler tree in Figure 5.5 has witnessed a total of 15 queries. Node A and node F have each been part of seven of these. Therefore the probability of both nodes is $P(A) = P(F) = \frac{7}{15}$.
This approach can be enhanced by the assumption that workloads change over time

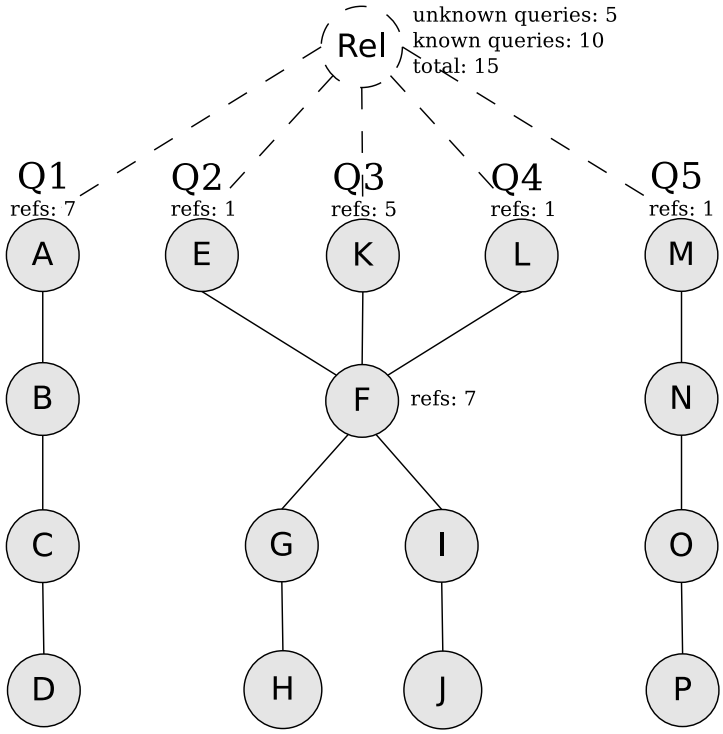

Figure 5.5: Fictive Recycler Tree

and that the recycler tree contains an excerpt of such a workload. As a consequence, future query invocations will not only include queries that are already present in the recycler tree, but also yet unknown ones. An incoming query is considered known, if it was successfully matched to the root with a query already in the recycler tree. An unknown query might still have common sub-trees with queries in the recycler, but will only partly match with any existing query. The probability presented in the previous approach is therefore the conditional probability $P(V_i|Old)$, given that an upcoming query is already known. The next step is to define the probability $P(V_i|New)$ of a known node being part of an as yet unknown query. The historic information collected within the recycler tree can again be used to predict that probability. If a known node is part of a new query, either it or one of its ancestors is the last known node of that query. The more direct parents a node has in the recycler tree, the more often it was the last known node of a new query in the past and hence the more likely it is to be the last known node of a new query in the future as well. The same applies to each of the node's ancestors in the recycler tree. However then, that ancestor was the last known node in the unknown query. The total amount of such splits of a node and all of its ancestors into their parents is the number of different queries that node has been part of in the past. Since all of these different queries have been unknown when they were first submitted to the recycler, it can be assumed that the more different queries a node has been part of in the past, the more likely it is to be part of a yet unknown query again in the future. Given that an upcoming query is not yet known, the probability of a node V_i is the number of different queries C_i that the node was part of in the past divided by the total number of different queries N_n in the recycler tree: $P(V_i|New) = \frac{C_i}{N_n}$. To keep track of the amount of different queries a node has been part of, C_i has to be introduced

to each node's statistics. Whenever a query tree of an unknown query is integrated into the recycler tree, N_n and C_i for each node of that query are increased by one. Node F of the recycler tree in Figure 5.5 is a good example to illustrate the probability of a node being part of an unknown query. Node F has been part of three of the five different queries in the recycler tree. The probability that F will be part of an yet unknown query is hence $P(F|New) = \frac{3}{5}$.
To compute the total probability $P(V_i)$ of a node V_i to be part of the next query, the probability that a future query is already known $P(Old)$ and the probability that a future query is unknown $P(New)$ have to be predicted as well. Assuming that the ratio between new and already known queries will stay the same in the future as it was in the past, the probability of a new query is the fraction of new queries N_n to the total number of queries N. This results in $P(New) = \frac{N_n}{N}$ and $P(Old) = 1 - P(New)$. These probabilities can be combined to formulate a final probability $P(V_i)$ for a node V_i:

$$\begin{aligned} P(V_i) &= P(V_i|New) * P(New) + P(V_i|Old) * P(Old) \\ &= \frac{C_i}{\cancel{N_n}} * \frac{\cancel{N_n}}{N} + \frac{R_i}{N} * \frac{N - N_n}{N} \end{aligned} \tag{5.14}$$

Interaction between Materialized Results

As was the case for computational cost, $P(V_i)$ is also dependent on other materialized results in the same query tree. There is a difference between 'being part of a query' and 'being used to answer a query'. If a node is part of a query, it does not necessarily mean that its result will be used to answer that query. If, for example, node B and C of Figure 5.5 were materialized and the current query was Q1, the query would be answered using B only. Having more than one intermediate in the same query tree materialized, influences the probability that some of these results will be used by future queries. This applies to $P(V_i|Old)$ as well as $P(V_i|New)$.
To incorporate the interaction between several materialized results in the same query tree into $P(V_i|Old)$, the number of times the result of V_i would have been used to answer a query in the past given the current recycler state has to be used instead of the number of times V_i has been part of a query. A materialized result can only be used by queries that have no ancestors of it materialized as well. This indicates that the number of times the result would have been used to answer a query is the number of times it has been referenced by a query that has no ancestor of it materialized at the moment. Instead of using the total number of references R_i for computing the probability of an intermediate result, that number has to be reduced by the sum of references that were contributed by queries that would not use that particular result anymore:

$$R_i' = R_i - \sum_{\substack{j \in \textit{direct} \\ \textit{materialized} \\ \textit{ancestors}}} R_j \tag{5.15}$$

When materializing node K and L in the recycler tree of Figure 5.5, the probability that the result of F will be used by an already known query decreases from $\frac{7}{15}$ to $\frac{7-5-1}{15} = \frac{1}{15}$

because F will no longer be used by the queries Q3 and Q4.
Such an interaction also exists when considering the probability $P(V_i|New)$. However, an unknown query will only use a materialized ancestor instead of V_i, if that ancestor or one of its ancestors is the last known node. If the last known node is V_i or a descendant of all of its materialized ancestors, V_i will be used instead. To obtain the probability $P(V_i|New)$ of a node that has one or more of its ancestors materialized, the number of different queries C_i' in the recycler tree that would have used the result of V_i when they were submitted have to be considered instead of the total number of different queries that contain V_i. C_i' can be defined as the total number of different queries the node has been part of in the past subtracted by the number of different queries each of its direct materialized ancestors were part of, but the first one:

$$C_i' = C_i - \sum_{\substack{j \in \textit{direct} \\ \textit{materialized} \\ \textit{ancestors}}} (C_j - 1) \tag{5.16}$$

The first query each materialized ancestor was part of is subtracted because when that query was submitted, the node was not known yet and hence could not have been used. Materializing nodes E, K or L of Figure 5.5 does not influence the probability of a new query using F because for F to be used by a new query, it has to be rooted by a different node than E, K or L and the fact that F previously was able to answer three of five new queries can be assumed to still be true in the future.

Managing Reference Statistics

Computing R_i' and C_i' from R_i and C_i, respectively, requires to find all direct materialized ancestors of V_i. As mentioned before, the effort to navigate ancestors in the recycler tree grows with the size of the tree whereas navigating descendants is bounded by the individual query size and does not grow over time. Therefore, instead of computing R_i' and C_i' each time a worth needs to be computed, R_i' and C_i' are stored in the statistics of the corresponding node in the recycler tree instead of R_i and C_i. Their value changes whenever a direct materialized ancestor of the result is added or removed and whenever a query uses that node.
Algorithm 5 describes the update process of all affected nodes after a new materialized result has been added to the recycler. Whenever an intermediate result finished materializing or an already materialized result is removed from the cache, R_i' and C_i' of all direct materialized descendants and all nodes touched before a direct materialized descendant is reached are updated.
Figure 5.6 illustrates the management of reference statistics in the recycler tree. Darker shaded nodes indicate that the result of the node is materialized in the recycler cache. When a node has finished materializing, it decreases the reference statistics of its descendants by the number of its references. However, the reference statistics of all descendants of an already materialized descendant are not changed. After node B finishes materializing in $t = 1$, it decreases the reference statistics of C and D by 7. It does not proceed past D because D is already materialized and already decreased the number of references for E before.

Algorithm 5 Updating R'_i and C'_i in the recycler tree after materializing an intermediate

Input: V_i /* corresponding node in the recycler tree */

UPDATE_REFS(child of V_i, R'_i, C'_i)

function UPDATE_REFS(V_j, R'_i, C'_i) **do**
  $R'_j \leftarrow R'_j - R'_i$
  $C'_j \leftarrow C'_j - C'_i$
  if V_j is materialized **then**
    return
  else
    for all children V_k of V_j **do**
      UPDATE_REFS(V_k, R'_i, C'_i)
    end for
  end if
end function

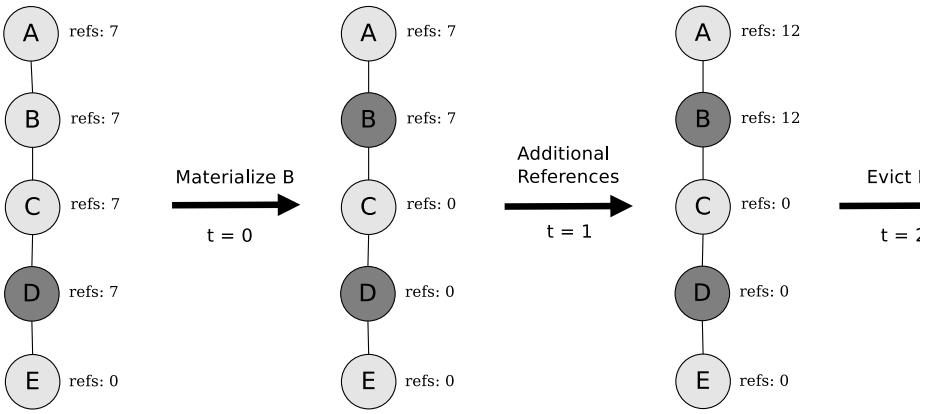

Figure 5.6: Updating the Reference Statistics for each Node

Whenever an incoming query references a node in the recycler tree, the reference statistic of that node is increased. Nodes which are descendants of already materialized nodes are not considered because the query uses the materialized node instead of recomputing its descendants. The additional references due to incoming query invocations in $t = 2$ do not increase the reference statistics of descandants of B because B is already materialized and each of the queries use B instead of recomputing B's descendants.
When a materialized node is evicted, the reference statistics of all descendants of that node whose number of references has been descreased when the node was materialized, are increased by its current number of references. When B is evicted in $t = 3$, the number of references of its descendants are increased by 12. The number of references still does not change for E, because an additional invocation of the same query will use the result of D instead. C and D now have the same number of references, as if B was never materialized.

Lazy Aging

So far, it has been assumed that all references to a node contribute equally to its probability. But workloads may change over time and the more recent the reference to a node is, the more likely it is to still be part of the current workload. If a node was for example heavily referenced in the far past, but since then never touched again, it is likely that the queries using it are no longer part of the current workload and that the node is therefore not likely to be referenced again even though it has lots of references. To reflect the over time decreasing contribution of a reference to an intermediate's probability, all statistics that are based on references have to be multiplied by a constant α slightly smaller than one each time a new query is added to the recycler ("aging"). Aging not only reduces the contribution of a reference to a node's probability with each incoming query, but also decreases the amount of the reduction. However it would not be feasible to navigate the entire recycler tree and adapt each node's references with every query invocation. Instead, the number of the query (QID) that adapted the node's references for the last time is recorded in the node's statistics and whenever that node is referenced again, all multiplications since the last reference are done at once ("lazy aging"). With each reference to a node, 5.17 has to be applied to all references in that node's statistics:

$$refs\_new = refs * \alpha^{QID_{curr} - QID_{last}} \tag{5.17}$$

Specificity of an Intermediate

When historic information does not provide any indication on the likeliness of an intermediate result to be part of an upcoming query again, the specificity of that result can be considered to estimate its probability. This is the case if a part of a query has occurred for the first time and therefore has no past information stored to differentiate between its nodes. With just using historic information, the root node of such a query will most likely have the greatest worth and hence be the only node considered for materialization.

This would be a good decision assuming that all intermediates of that query part will only be used by the same query in the future. If there are different queries, that have some intermediates in common, the materialized root node could not be used to answer them unless the common sub-tree is the entire query. The more specific a result is, the less likely it is to be used by another, yet unknown query. Including the specificity of an intermediate result will increase the chance that useful intermediates will be materialized already before enough information is available to make an informed decision based on previous behavior. Without using specificity, these intermediates would still be materialized if they are worthy enough, but only at a later time, when enough historic evidence is collected.
The specificity of an intermediate result increases with each operator in its sub-tree and is dependent on that operator's type and specificity. For example a select with a high selectivity produces a more specific result than one with a low selectivity. Estimating the specificity of each intermediate result is too complex for use in the recycler. Instead the specificity can be approximated by assuming that each operator increases the specificity of an intermediate result by the same amount. This suggests using the tree height as a measurement for specificity.
The height could be included as a minimum value for C_i:

$$(C_i)_{min} = \beta^{inverse\ height}, \quad \beta < 1 \tag{5.18}$$

When copying a node from a query tree to the recycler and creating its statistics structure, C_i could be initialized by $(C_i)_{min}$ instead of 1. This would mean that the first query a node was part of contributes less to the total number of different queries the node is part of.

5.5 Adding Intermediates to the Recycler

After finding a formula to calculate the worth of each intermediate result in the current query and the recycler cache, that formula has to be used to decide which node(s) of the current query to materialize. The goal is to maximize the total worth of all materialized results in the recycler. Candidates for materialization are all nodes in the operator tree which have a Store operator added on top. Ideally, the worth of all these nodes as well as all possible combinations of them would be computed and then the set of intermediates that contribute the highest total worth would be materialized. For n nodes this would result in $\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = \sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$ worth computations. A full recycler cache requires even more worth computations and comparisons in order to find materialized results that can be evicted in favour of new intermediates. The amount of worth computations and comparisons has to be reduced significantly in order to make it applicable to the recycler. In the following it is assumed that the recycler will only select the result with the highest worth whenever it has to pick a result for materialization. In case more than one result would have contributed a higher worth to the recycler cache, the results that were not selected will be considered again the next time they occur in a query.

The recycler treats Store operators that have historical information available in the corresponding node in the recycler tree differently to Store operators that are executed for the first time. If the query tree has common sub-trees with the recycler tree, the Store Operator that corresponds to the root node of each of these sub-trees is defined as Master Store. A Master Store and all of its descendants have historical information. Their worth can be computed before executing the query. When allocating a Master Store, it identifies all descendant Store operators and invokes worth computations on each of them. Then it selects the result in its sub-tree with the highest worth and changes its stage to Append. The stage of all other Store operators in the sub-tree is set to Canceled. Store operators that have historical information available do not require buffering or allocating Reuse operators when in Canceled Stage. For sub-trees that have historical information available, the decision on which result in the sub-tree to materialize could alternatively already be done in the rewriter. When deciding in the rewriter, Store operators are no longer needed for that sub-tree. Instead, the rewriter rule could insert Append operators on top of operators that produce the result that was decided to be materialized.
When there is no historical information available for a result, the decision on materializing that result has to be done using dynamic estimates while executing the query. This requires the result to be buffered for a while until the estimates can be assumed to be close enough to the correct value. An estimate is assumed to be close enough if its value does not differ more than a certain percentage between two successive next() calls. As described in Chapter 5.3, not all parts of an operator tree are executed at the same time. Each blocking operator divides the tree in two parts being executed separately. Deciding which result to materialize at once for all results in the query tree that require estimates would result in buffering entire results for segments that are already finished. Therefore, the decision can only be done in isolation for each of these segments. Figure 5.3 illustrates the division of an operator tree into several segments that have to be considered separately. When adding Store operators on top of results that require estimates, the Store operator beneath each blocking operator and the Store operator on top of the root node of the operator tree are defined as Master Stores. These operators identify all buffering Store operators in their segment and invoke the worth computation on all of them as soon as the estimates are considered close enough to the actual value. The result with the highest worth in each segment is then materialized, all other Store operators in the segment are set to Canceled. Materializing one result for each segment can result in a too high materialization cost. Heuristics based on the cost and size estimates can reduce the number of materialized results in each operator tree. Examples for such heuristics are presented in Chapter 7.

5.6 Evicting Intermediates from the Recycler

Since there is only a limited amount of main memory space available, the size of the recycler cache has to be restricted. Once the recycler cache is full, a new intermediate result can only be added if there is a set of materialized results that has a lower worth than the intermediate and occupies at least as much space. Before materializing the new intermediate, all materialized results in that set have to be evicted to create the

necessary space for the new result. To optimize the usage of the limited memory space, the total worth in the recycler cache has to be maximized. When adding a new intermediate to a full cache, the set of materialized results has to be found that, after evicting it and adding the new intermediate, results in the highest total worth. Finding that set requires the recycler to compute the worth of all materialized results in the cache. It is not possible to reuse worth computations for future comparisons, because the worth of a result may change with every new query invocation and every added or removed materialized result.
As mentioned in Section 5.4.3 and 5.4.4, removing a materialized result increases the probability $P(V_i)$ of its direct materialized descendants and the real cost $R(V_i)$ of its direct materialized ancestors. Removing materialized results from the recycler might therefore increase the worth of other materialized results as well as the worth of the intermediate result which was selected for materialization. Adding an intermediate result might decrease the worth of some materialized results in the recycler cache. Also computing the effect of adding and evicting materialized results on all other results in the recycler cache would result in lots of additional worth computations. To reduce the number of necessary worth computations, the recycler assumes that these interactions balance each other out.
To further reduce the amount of necessary worth computations, the recycler limits the number of materialized results the intermediate is compared to. Instead of comparing the intermediate to every materialized result in the recycler cache, it is only compared to a subset of its content. The intermediate is added to the recycler, if there is a set of materialized results in that subset that has less worth than the intermediate and will free up enough space after eviction. Because not all materialized results are considered for eviction, the chosen set of materialized results might not be the ones with the lowest worth in the recycler. Furthermore, an intermediate that should have been added to the recycler might be denied because it was unlucky and all materialized results considered had a higher worth. These problems are compensated by the fact, that the worthy result can be assumed to occur again in upcoming query invocations. The result will then have another chance to be added to the recycler and might then be more lucky.
Within the recycler, all materialized results are organized in a list structure. The subset of materialized results the intermediate is compared to are NUM successive elements of that list. Each time a new intermediate is tested, the comparison starts with the first element that was not considered the last time. When the end of the list is reached, the comparison continues with the first element. Algorithm 6 illustrates how it is decided whether an intermediate should be added to a full recycler cache and which materialized results to replace in favour of the new one. The worth of every considered materialized results is computed and the ones that have less worth than the intermediate are added to the candidate list in the order of their worth. If the combined size of all elements in the candidate list is smaller than the size of the intermediate, the intermediate will not be materialized. If the size is greater or equal, the materialized results are evicted in order of increasing worth until enough space is freed to add the new intermediate.

Algorithm 6 Adding an Intermediate Result I to a full Recycler Cache R

```
Input:
    I,                                                    /* intermediate */
    R = {M_0, ..., M_k},   /* recycler cache containing materialized intermediates */
    s                                  /* pointer to next element in recycler cache */
NUM                                                       /* length of search */
C ← ∅                                  /* candidate list, sorted by increasing worth gain */
sz ← 0
for i = s to NUM do
  if worth(M_i) < worth(I) then
    C ← C ∪ M_i
    sz ← sz + size(M_i)
  end if
end for
if sz < size(I) then
  set stage of I's Store operator to Canceled
else
  sz ← 0
  for i = 0 to |C| − 1 do
    if sz < size(I) then
      remove M_i from R
    end if
    sz ← sz + size(C_i)
  end for
  set stage of I's Store operator to Append
end if
```

6 Using Materialized Results

After choosing which intermediate results should be materialized and materializing them to the recycler cache, the results need to be used to answer incoming queries. A materialized result can be used to answer a query, if the result was produced by a node of one of the sub-trees in the recycler tree that got matched with the query. In order to use that result, the recycler has to replace the query's sub-tree rooted by the node producing the result with a sub-tree that reads the result from the recycler cache and pre-processes it to make it compatible with the rest of the query tree. This can include simply renaming the table and column names to the ones used by the query or extracting a subsuming result from the materialized result before using it to answer the query.
A rewriter rule is used in order to identify materialized results that can be used to answer the query and to adapt the query tree to use these results instead of recomputing their sub-tree. The rule is applied after the rewriter rule that matches each node of the query tree with the recycler tree and before the rule that adds store operators to the query tree. In contrast to the two other rules, this one is top-down. Top down ensures that only the highest materialized node of a matching sub-tree is replaced. As preparation for this rule, the matching rules annotates the query tree with table names and the name mappings of all matched nodes in the query tree that are already materialized in the recycler cache. That information is needed to build the sub-tree that uses the materialized result.
The rule then checks if the current node is annotated with a table name. If so, it generates the sub-tree needed for using that result and replaces the node and its sub-tree with the new one. The replaced sub-tree will no longer be considered by the rule and therefore no other materialized node in that sub-tree will be used anymore.
The new sub-tree is constructed by a table scan that scans the materialized result from the recycler cache and a projection on top of it. The projection is used to rename the table and column names to the ones expected by the query. The name mapping and the table in the recycler cache are utilized for that purpose.

7 Evaluation

This chapter will evaluate the implementation of the recycler as described in the previous chapters.
There are a few differences between the version of the recycler used for evaluation and the one described previously. The main difference is that the recycler implementation restricts the size of the recycler cache by the number of materialized results in the cache[1] instead of the space that the tables occupy in it. This faciliates the cache management because when the recycler cache is full, each new result has to replace exactly one materialized result in order to be submitted to the cache.
To assist the evaluation process, the recycler implements two routines which are invoked using VectorWise algebra:

> **print**: When calling print, the server prints the current recycler tree. Each node of the printed tree structure is annotated by information collected on the node such as its number of references, its computational cost or if the result of the node is currently materialized in the recycler cache. The print function is used to visualize which nodes of the recycler tree are materialized.
> **flush**: Flush resets the content of the recycler. After calling it, the recycler tree and the recycler cache are reset to their initial state. Before benchmarking the recycler, the server is usually 'warmed up' by running several queries. Afterwards, flush is used to reinitialize the recycler. Benchmarks are then executed on a warm server.

The functionality of the recycler was tested using all 22 queries of the TPC-H benchmark. All tests were performed on a dataset with scale factor 10. The tests were performed on a four core Intel machine with 8GB main memory using Fedora 12.

7.1 The Workload

In this section, the recycler is evaluated using the same TPC-H workload that was already utilized in [IKNG09]. The authors used this workload to evaluate their caching policies. The workload consists of 200 queries, 164 of which are unique. The authors examined the potential overlap in MonetDB instructions among derivates of the same TPC-H query pattern and, out of the 22 TPC-H queries, selected the 10 queries with the greatest potential overlap for the workload. These queries are: 4, 7, 8, 11, 12, 16, 18, 19, 21 and 22. The workload is constructed using 20 instances of each query pattern. Some of these 20 instances are different derivates and some are repetitions of the same derivate. Derivates of the same TPC-H query have the same structure, however some of

[1] this choice is rather simplictic and was chosen due to time contraints

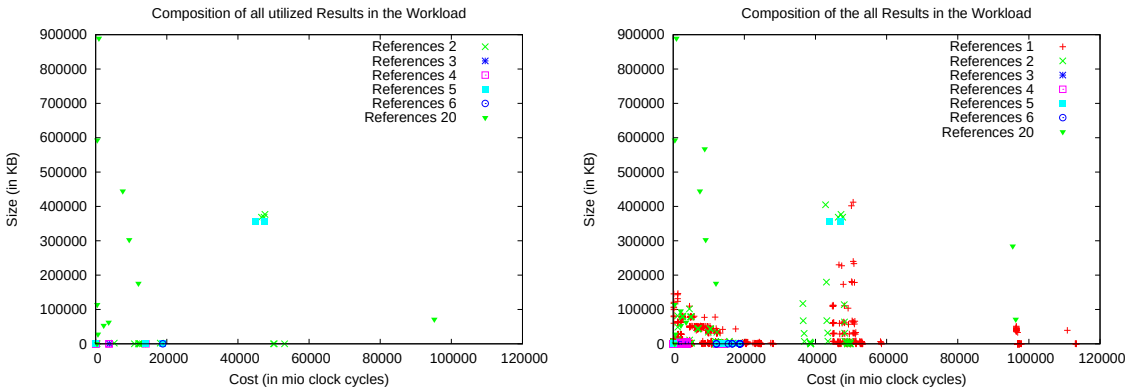

Figure 7.1: Composition of the Workload

their parameters differ.
The workload contains two kinds of sharing possibilities. 36 of the 200 query invocations are repetitions of queries that have already been executed before. These queries could be answered using the materialized final result of a previous invocation of the same query. The second sharing possibility is to reuse common sub-trees from different derivates of the same query pattern. Although these queries use different parameters, they often share common sub-trees. These sub-trees typically make up only a small fraction of the query tree and close to the leaf nodes.
The workload contains a total of 44 common sub-trees. Sub-trees that are contained within another common sub-tree but have references from additional queries are counted separately. Therefore a recycler architecture cannot exploit all of these sub-trees to their full extend. Figure 7.1 (l.) shows the cost, size and number of references of the result with the highest worth for each of these sub-trees. Figure 7.1 (r.) shows the cost, size and number of references for each result in the workload.

7.2 Modes used for Evaluation

Naive: The Naive mode is VectorWise with recycling turned off.

History: The History mode only uses historic information to decide which results to materialize. As a consequence, it only materializes results that have already been executed and therefore have historic evidence that they might occur again in a future query. Since all information that is required for the worth computation is available for all results which are considered for materialization, the decision on which result to materialize can already be carried out when allocating the Store operators as described in Chapter 5.5. As a result, the History mode no longer requires buffering of results.
Since results are only considered for materialization the second time they occur in a workload, a result has to occur at least three times before the recycler can benefit from reusing it. Therefore, the History mode performs badly for results that only occur twice in the workload.

Mixed (History and Speculation): The Mixed mode is designed to overcome this obstacle. For results that have already been executed and hence have historical information, the recycler proceeds as described above. For all other results, it uses speculation. Speculation materializes results speculatively in the hope that they will occur again in the future. If a frequent result is materialized the first time it is executed, the recycler can already benefit from reusing it the second time it occurs in the workload. Speculation relies on dynamic estimates and therefore requires buffering the result until a decision is made.
The first benchmarks of the Mixed mode unveiled several problems when:

1. *Too much buffering*: There are several reasons for Store operators buffering too many tuples: If the number of tuples that are produced by each operator in a pipelined segment decreases too much from the beginning of the pipelined segment to the Master Store, the early operators have to buffer too many tuples before the Master Store has enough tuples to base its decision on. Furthermore, if the cost estimate changes too much between successive next() calls, the entire segment has to buffer too many tuples until the estimates stabilize and the Master Store can decide.

2. *Too many results are materialized*: Since the decision on which results to materialize is done for each blocking segment separately, the recycler materializes one result for each segment until the recycler cache is full. Each TPC-H query used in the workload contains between three and seven segments and some of the results are as big as one gigabyte. This leads to too high materialization cost for each query.

3. *Speculation is often favoured*: There is a huge cost and size difference between nodes close to leaves and nodes close to the root. Nodes close to leaves tend to have a low computation cost and big result sizes whereas nodes close to the root have a high computation cost and small result sizes. Both factors favour nodes close to the root when comparing their worth. The difference between nodes close to leaves and nodes close to the root often dominates the reference factor. This results in a higher worth of speculative results that are close to the root compared to results that have several references but are close to leaf nodes.

The first two problems lead to too high overhead for speculation and the last problem causes the recycler to not realize all sharing possibilities unveiled by the history approach.

MixedH (Mixed with Heuristics): To deal with the problems of the Mixed mode, several heuristics are introduced:

1. *Limit buffering and result size*: Each Store operator cancels buffering and continues in the Canceled stage once the size estimate suggests that the result will not fit into a single block. This heuristic ensures that the recycler only considers very small results which are cheap to materialize and that it cancels buffering results that will exceed the size of a block as soon as possible. In most cases, this decision can already be performed in the first next() call after the Store operator's child has produced the first result vector. Early canceling Store operators that have to materialize results which are too big reduce the cost of buffering significantly.

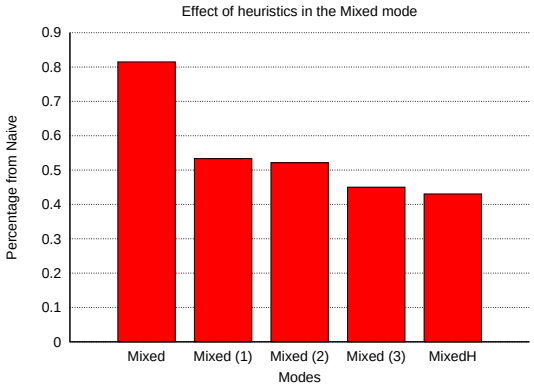

Figure 7.2: Evaluation of the Heuristics in the MixedH mode

2. *Define a minimum cost/size ratio for buffering*: When the Master Store of a segment chooses a result to materialize, only results that have a *cost/size* ratio which is higher than a threshold are considered. A high enough threshold ensures that not every segment materializes a result.

3. *Always favour results with historical information*: This heuristic ensures that a speculative result never replaces a result which has historical evidence that it occurs more than once in the workload. Furthermore, a result with historical evidence always replaces a speculative result, even if the speculative result has a higher worth.

The main goal of these heuristics is to reduce the overhead of speculation and to only use speculation for results that are very cheap to materialize but result in a high gain when reusing.
Figure 7.2 shows the relative execution time of the workload with the standard Mixed mode and with each of the aforementioned heuristics enable separately. Each of the heuristics improve the execution time of the workload significantly, however, their combination is only marginally better than the third heuristic alone.

Perfect Oracle: The last mode is the Perfect Oracle mode. This mode was introduced to simulate a recycler with 'perfect knowledge'. In the Perfect Oracle mode, the entire workload is first run against the recycler with materialization turned off. This run builds the recycler tree for the workload. Each node now has historical information and, in particular, knows its total number of references. After the first run has finished, materialization is turned back on and the number of references to each node is reduced by one to make sure that nodes that only occurred once in the workload no longer have any references. Then the entire workload is run again using the History mode. The execution time of the second execution of the workload is measured.
In contrast to the History mode, the recycler already knows which nodes are going to be frequent in the workload when the nodes are executed for the first time. If a node occurs more than once in the workload, the recycler materializes the result the first time it is computed and reuses it for all other occurrences.

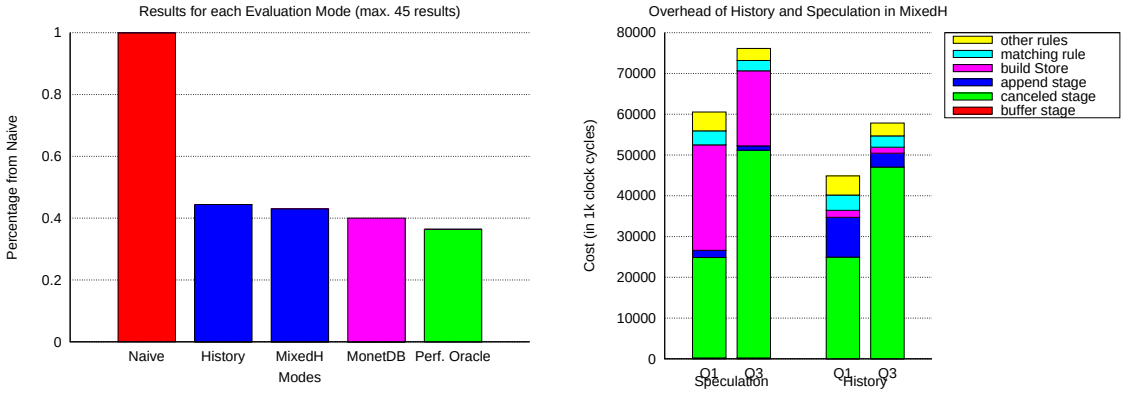

Figure 7.3: Results of the various modes of the Recycler (l.) and the Overhead of Recycling (r.)

7.3 Results

Figure 7.3 (l.) shows the improvements that can be achieved using the presented modes of recycling. All modes reduce the execution time of the workload significantly compared to the naive approach. However, the improvement of the MixedH mode by adding speculation to the History mode is only marginal. It is far away from the possible improvement that is suggested by the Perfect Oracle mode. The figure furthermore shows the best improvement that was achieved by [IKNG09] on the same workload using MonetDB. Recycling in MonetDB performs better than the presented Recycler for VectorWise, however, the difference is only marginal.
Figure 7.4 shows the cost, size and number of reuses of all results that are in the recycler cache after the execution of the workload has finished for each of the three modes. The maximum number of materialized results in the recycler cache has been set to 44. Since the workload only contains 44 beneficial results, this only poses a limitation for the MixedH mode. Figure 7.4 confirms the expected behaviour of the three modes. The History mode materializes each result the second time it occurs in the workload and reuses it from the third occurence. The MixedH mode behaves like the History mode, but in addition, speculatively materializes some of the results with a size of 32KB and a $cost/size$ ratio beyond the threshold of the second heuristic. While executing the workload, the recycler speculatively materializes more results than the ones shown, however, these results are not reused and are evicted before the workload finishes executing because of the recycler cache restriction of MixedH. The Perfect Oracle mode materializes each result the first time it occurs in the workload and reuses it from the second occurrence. Some of the most frequent results in the workload are used by different derivates of the same query as well as by repetitions of the same derivate. Usually, the recycler materializes the root node of a repeating query and therefore the frequent result will no longer be reused by that query. The three modes differ in the number of reuses of these results. History uses the more frequent results more often because each repeating query has to occur more often in the workload before its result is materialized and can be reused. The Perfect Oracle mode, on the other hand, can reuse the repeating queries earlier and therefore causes fewer reuses of materialized results which are their descendents. Speculation behaves like Perfect Oracle for results that were speculatively

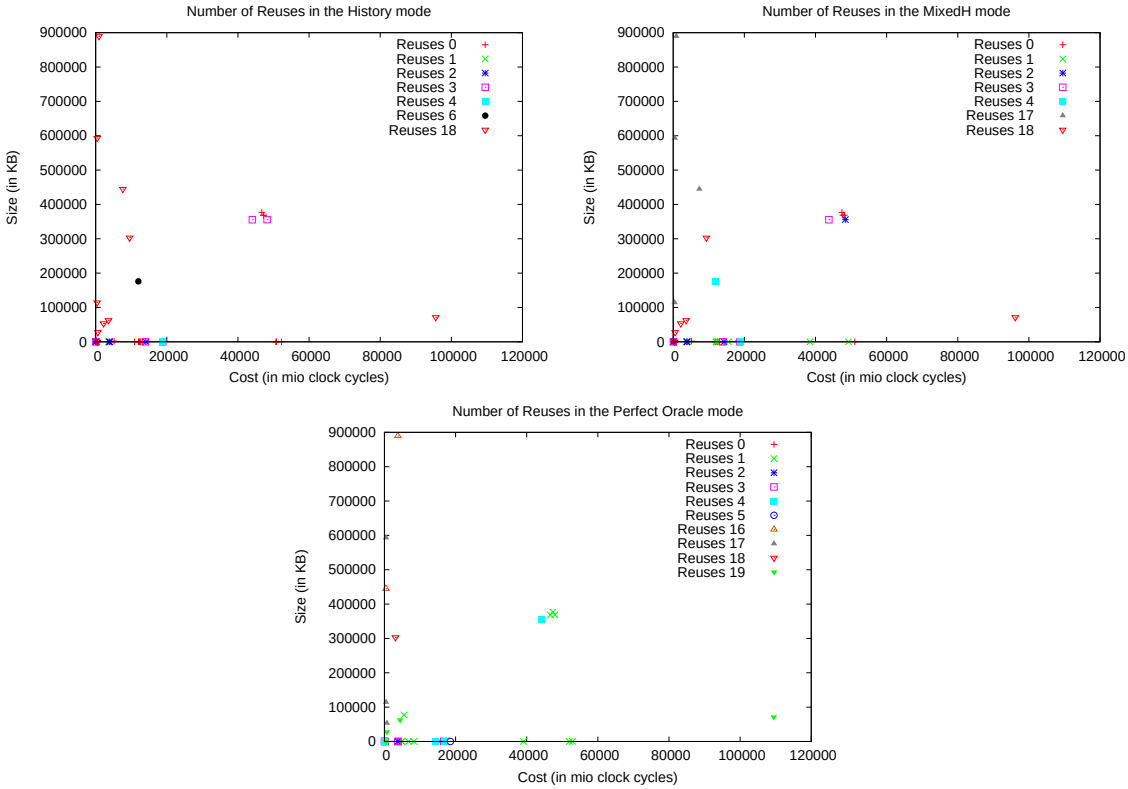

Figure 7.4: Results Reused by the History, MixedH and Perfect Oracle mode

materialized the first time they occured and like History for all other results.
The fact that the cost of the Perfect Oracle mode differs from the cost of the other modes can be explained by cache effects. Although the database server has been warmed up for each mode in a similar fashion, the results were often materialized by different query invocations. Since the same result executed by a different query invocation can have different computational cost because of different buffer pool contents, the times measured and stored for these results may differ.

History: Figure 7.3 (l.) shows a significant improvement in the execution time of the workload when using recycling in the History mode. However, the execution time of the History mode as well as the execution time of the MixedH mode when using historical information from previous query evaluations could be further improved.
Figure 7.3 (r.) shows the overhead of the recycler when evaluating queries 1 and 3 of the TPC-H benchmark using the MixedH mode, both with all historical information available and without any information. If there is no historical information available, speculation is the only option for the recycler. Both queries were run on a warm server with an empty recycler cache. When using historical information, the recycler tree contained only the query tree from a previous execution of the same query. For speculation, the recycler tree was empty. Both modes only materialized one result. The materialized results were close to the root and had a similar size.
The cost of the overhead shown in Figure 7.3 (r.) is under 1% of the entire evaluation cost of the query. The possible gain by the suggested improvements is therefore rather

small.
When using historical information, the biggest contribution to the overhead is from executing Store operators in their Canceled stage, followed by the cost of the Store operator in the Append stage. Most of the cost of Store operators in their Canceled stage is contributed by Store operators that have to pass huge results to their parent operator. The cost of the Canceled stage can be entirely removed by using a rewriter rule to decide which result with historical information to materialize, as suggested in Chapter 5.5. The rewriter rule then only adds Append operators on top of operators that produce results which were decided to be materialized. Store operators which would start in their Canceled stage are never added.
The cost of appending a (small) result when using historical information is significantly higher than the cost of appending a comparable result when using speculation. Most of the difference is caused by allocating the Reuse Group and both Reuse operators. When materializing a speculative result, these operators are already allocated because of buffering whereas when materializing a result with historical information, the operators are allocated when creating the table and allocating the Append operator. The Store operator has to use the Reuse operators instead of only using the Append to make sure that the parent receives the correct result at all times. However, when deciding on which result to materialize in a rewriter rule as described above, the rewriter rule can insert an Append operator instead of the Store operator. This reduces the cost of materializing the result and saves the cost of allocating Store and Reuse operators.

Speculation: Figure 7.3 (l.) only showed a small improvement when extending the History mode with the addition of speculation. The improvement could be a lot bigger as indicated by the result of the Perfect Oracle run. When comparing the overheads shown in Figure 7.3 (r.), the main difference between using historical information and using speculation is the building cost of the Store operator. The additional cost when using speculation is caused by allocating the Reuse Group and both Reuse operators which are used for buffering and later materializing the result. Since buffering is required to enable the use of dynamic cost estimates, this cost cannot be reduced without changing initial design decisions. The cost of the Store operator in its Buffer stage and the cost of materializing a result have already been significantly reduced by the use of heuristics and both no longer have a significant effect on the execution time of the workload.
A possible improvement would be to only insert one buffering Store operator on top of the root node of each query tree. This would result in only using speculation for the final result of a query. Although this would be a restriction to the recycler architecture, the speculative results which are materialized by the MixedH policy are predominantly close to the root node and do not exceed one result per query.
Figure 7.1 shows the cost and size of all potential reuse possibilities in the workload. Extending the History mode with speculation creates one further reuse possibility for each of these results. Since the first heuristic of the MixedH mode only allows materializing results that fit a single block, only about 37% of the cost improvement that could be achieved by also using speculation in the workload can be realized. The difference between the cost of executing the workload using the History mode and the cost of executing it using the Perfect Oracle mode represents the maximum improvement that could be achieved by enabling speculation. 37% of that improvement is 1min 4sec. The

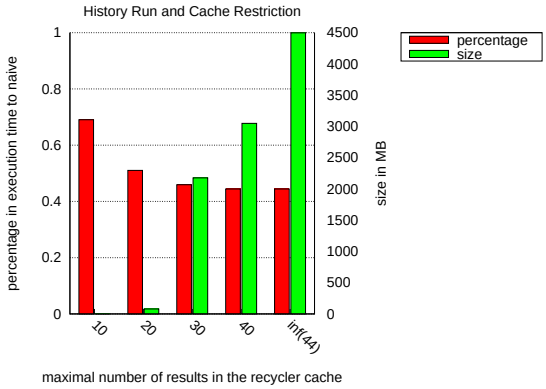

Figure 7.5: Evaluation of the Recycler with different Result limits

recycler, however, only manages an improvement of 30sec over the History mode when using MixedH. This difference can be explained by the additional overhead shown in 7.3 (r.) as well as the cost of materializing results that are not referenced in the workload anymore and missing possible chances because of a restricted recycler cache. For the given workload, the MixedH mode speculatively materialized 72 results. Only 6.9% of them were reused by successive queries and another 6.9% of them were evicted before they could have been reused. The latter case could be prevented by an unlimited recycler cache size. However, further benchmarks did not show a speed improvement by doing so.

Cache Restriction: The cache restriction has been introduced to limit the amount of main memory that is used by the recycler. The experiments from the previous paragraphs were all obtained using an unlimited cache for the History mode and a limit of 44 results for MixedH. The cache was limited by the number of results in the recycler cache. Figure 7.5 shows the improvement of recycling using the History mode with various cache limits and the size of the recycler cache after the workload has finished executing. For the given workload, an unlimited cache gives the best results. However, it also results in the biggest recycler cache size. Reducing the maximal number of materialized results in the cache reduces the improvement realized as well as the resulting recycler cache size. The biggest increase in cache size is between a maximum of 20 and 30 results. Although an unlimited recycler cache results in the best execution time for the given workload, limiting the maximum number of results to about 20 results in the better usage of main memory space. It consumes significantly less space for an only marginally worse result. If there is not enough main memory space for an unlimited recycler cache, using a limit of 20 results already achieves a good improvement in execution time by using a lot less memory space.
Limiting the size of the recycler cache by the number of materialized results has several disadvantages. While executing the workload, the size of the recycler cache varies heavily. Especially before the maximal number of materialized results is reached, the recycler materializes all results and reaches a much larger size than after executing the workload. Furthermore, the recycler does not differentiate between small and large results in the recycler cache limit. Although the recycler cache could contain many very small results, their amount is also limited by the cache restriction. These problems could be addressed by restricting the recycler cache by its size.

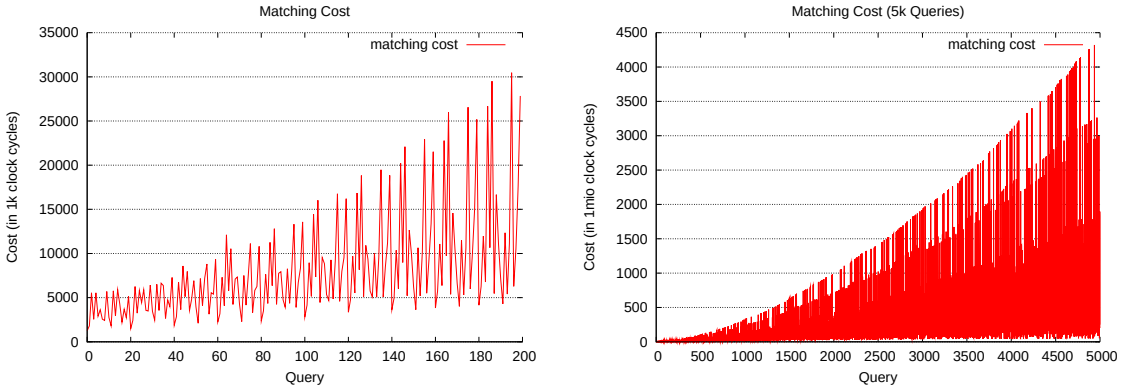

Figure 7.6: Cost of Matching for the Evaluation Workload (l.) and a Workload contaoning 5k Queries

Furthermore, the cache restriction serves as an additional heuristic. It reduces the number of results that are materialized by providing a dynamic minimum worth. However, the heuristic is only active once the recycler cache is full. Until the cache is full, it does not restrict the number of materializations. Therefore, the recycler uses additional heuristics like the second heuristic of the MixedH mode. A similar heuristic would also be conceivable for results with historical information, however, it did not show any benefit for the given workload. Once the cache is full, the minimum worth provided by the recycler cache replaces the static threshold. This applies to speculative results as well as to results with historical information.

Cost of Matching: One concern for the presented recycling architecture is the cost of operating a growing recycler tree, in particular the cost of matching. Chapter 4 suggested to regularly trim the size of the recycler tree in order to keep the cost of operations like matching in a manageable dimension. However, in the current state of the recycler, the recycler tree grows infinitely. Figure 7.6 (l.) shows the matching cost for each query in the workload. It is clear that the cost paid for matching increases with the size of the recycler tree. Furthermore, there appear to be several groups of queries with different gradients. These groups could be composed of queries that use the same number of base tables and therefore have to initially check a similar number of candidate table scans in the recycler tree. The bloom filter approach which was proposed in Chapter 4 to speed up the matching of table scans is not yet integrated into the recycler.
Although the cost of matching was still in a manageable magnitude ($< 0.5\%$ of the execution time of each query) for the 200 query workload, the experiment was repeated with a bigger workload to get a better insight on the scale of the problem. The workload which has been created for that purpose consists of 5000 invocations of the 22 TPC-H queries with randomly assigned parameters. This results in an overlapping workload with 4600 unique queries. The results are shown in Figure 7.6 (r.). The cost of matching increases heavily until at the end it reached the dimension of some of the cheaper queries in the workload. The graph again indicates that there are several groups of queries which are differently affected by a growing recycler tree.
The results of both experiments show that the number of queries in the recycler tree has to be limited and that matching has to be further optimized to keep the cost of

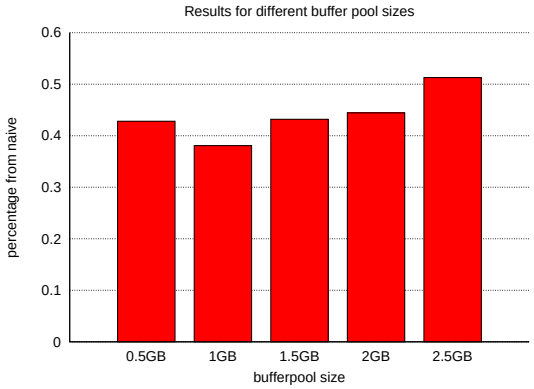

Figure 7.7: Influence of Disk I/O on the Recycler

Buffer Pool Size	0.5GB	1.0GB	1.5GB	2.0GB	2.5GB
Recycler ON	25m30.929s	22m38.417s	17m37.011s	15m54.594s	9m2.084s
Recycler OFF	59m36.131s	59m28.561s	40m46.656s	35m48.694s	16m4.660s

Table 7.1: Absolute number of the results shown in Figure 7.7

matching under control.

Influence of Disk I/O: An important influence on the performance of the recycler is disk I/O. When using a materialized result to answer a query, the recycler benefits from the time saved by executing that result. The more computationally expensive a result is, the more can be gained from reusing it. An important factor of the cost of executing a query is the number of blocks that have to be fetched from disk. Disks are typically several orders of magnitude slower than main memory and reading blocks from disk therefore slows down query execution. As a consequence, a recycling architecture performs better if there is a lot of disk I/O involved. This is the case for large datasets or small buffer pool sizes. If the buffer pool size is too small to hold beneficial blocks until they are required again by a successive query, thrashing occurs. As a result, each block required by the query has to be read from disk.
In the previous experiments, the recycler has always used the default buffer pool size of 2GB. Figure 7.7 and Table 7.1 show the influence of various buffer pool sizes on the performance of the recycler. The results were obtained by using the History mode with an unlimited recycler cache size. The benchmarks confirm the general tendency that the effect of recycling increases with smaller buffer pool sizes. However, the effect of recycling is less at 0.5GB than at 1.0GB. The following theory could explain this behaviour. Without recycling, thrashing occurs at both buffer pool sizes. This results in comparable execution times. The execution times of both buffer pool sizes with recycling turned on should also stay at about the same level. However, thrashing is reduced by the recycler using materialized results and therefore eliminating disk I/O. As a result, blocks stay for longer in the buffer pool. At 1.0GB they stay long enough to become useful for subsequent queries, hence the 1.0GB performs better than 0.5GB.

8 Conclusion and Future Work

This thesis has presented a recycling architecture for pipelined query evaluation.
The Recycler materializes intermediate and final results as temporary tables in main memory. Additionally, it stores the query trees of all incoming queries in a structure called the *recycler tree*. It then compares the query tree of each incoming query with the recycler tree to identify materialized results that could be used to answer the query. If such a result is found, the query is answered using the materialized result instead of recomputing it from scratch.
The selection of results to materialize has been identified as the most crucial part for a recycler architecture in a pipelined database system. Therefore, the recycler has introduced novel methods for selecting results. It uses historic information as well as dynamic estimates to decide which results to materialize. Historic information is collected while evaluating a query and then stored in the recycler tree. Dynamic estimates are used whenever there is no historic information available. It is obtained by buffering the results for a while and then using the measured run-time parameters to estimate their final values.
The recycler has introduced two policies to decide which result to materialize: History and MixedH. History is solely based on historical information. MixedH is based on a combination of historical information and dynamic estimates. Whenever there is no historical information for a result available, MixedH use dynamic estimates to speculatively materialize results.
The usefulness of recycling depends on the characteristics of the workload. The possible improvement achieved by recycling is dependent on the fraction of the execution time of the entire workload that is caused by recomputing results that have already been computed by previous query invocations. The number of reuse possibilities that the recycler is able to exploit is dependent on the recycling policy and the size of the recycler cache. History has to compute each result at least twice before it can start reusing it whereas speculation can potentially already reuse the result the second time it occurs in the workload.
Although the workload used for evaluating the recycler has especially been tailored for recycling, there are also real-world workloads that can benefit from recycling. M. Ivanova *et al.* [IKNG09] has, for example, shown that SkyServer[1] workloads contain many overlapping queries and can benefit significantly from recycling.
The evaluation of the recycler showed a significant improvement of the execution time of the workload with recycling. The results obtained with recycling in a pipelined DBMS have been shown to be competitive with the results obtained in MonetDB. However, the improvement of adding speculation to the history mode has been rather marginal. The effectiveness of speculation could improve with other workloads that contain more identic queries with small result sizes.

[1] http://cas.sdss.org/

The results suggest that recycling could also be a viable feature for commercial DBMS. Recycling could, for example, be manually turned on for workloads that contain many overlapping queries. However, before recycling could be used in a production environment, the impact of a growing recycler tree on the cost of matching queries has to be controlled.
The following will present further research topics:

Trimming the Recycler Tree

As mentioned in Chapter 4, the recycler tree grows infinitely in its current implementation. With each new query, the memory space that the tree requires increases. Furthermore, the cost of matching an incoming query with the recycler as well as the cost of navigating the recycler tree towards its root increases. Therefore, the tree needs to be regularly truncated. To do so, the recycler could periodically iterate over the root nodes of each unique query in the recycler tree. If the aged references value of one of these nodes is smaller than a threshold (e.g. 0.6), the query rooted by that node is removed from the recycler. If the query has common sub-trees with other queries in the recycler tree, these queries have to remain in the recycler if the root nodes of these sub-trees have an aged reference value above the threshold. All nodes that are not shared with other queries in the recycler are removed from the recycler tree. Nodes that are shared remain in the tree and are further accessible by their other parents. However, their references have to be reduced by the amount contributed by the removed query. If another query in the recycler tree is entirely contained in the tree of the query which will be removed, the recycler has to make sure that the root node of that query is added as child of the virtual root node in the recycler tree. After removing a query, the total number of queries in the recycler tree has to be reduced.

Integration into the Optimizer

So far, the decision which materialized results to use for answering an incoming query has been rather trivial. There has been at most one option for each node and the result from the recycler has always been assumed to be faster than recomputing the result. However, when integrating subsumption (see Appendix A), these assumptions may not be valid anymore. Each node can have several results that can be subsumed to obtain the needed result. The recycler then has to select the cheapest one. Furthermore, using some of these materialized results might even be more expensive than recomputing the entire result. In order to choose the best plan for the query, the selection of materialized results to answer the query with has to be integrated into the optimizer. The optimizer can then select the best query plan from the ones generated by the optimizer and the ones that use materialized results.
Furthermore, only using the already optimized query plan to find materialized results that could be used to answer a query might miss potential reuse possibilities. For example different join orderings that were not chosen by the optimizer could unveil further sharing possibilities. Using these possibilities might result in a cheaper execution plan than the one originally chosen by the optimizer. When the selection of materialized

results to answer the query with is integrated into the optimizer, the recycler can find additional sharing possibilities in all permutations of the query tree that the optimizer considers and then the optimizer selects the cheapest query plan including all discovered sharing possibilities for execution.
Finally, when integrating the selection of materialized results with the optimizer, the optimizer could incorporate available indexes on base tables (or materialized results) into the decision whether to use a result for answering an incoming query.

Updates on Base Tables

So far, it has been assumed that there are no updates in the workload. Updates have been assumed to be done offline and the state of the recycler is reset after updating the tables. The additional consideration of updates is a huge challenge for a recycler architecture. Most previous recycler architectures have ignored updates or chosen very simple update schemes.
There are several possibilities on how to handle updates. The easiest is to flush the entire recycler cache whenever an update, insertion or deletion occurs. The recycler could use the flush function to simulate such a behaviour. A slightly more advanced strategy would be to only remove materialized results that are affected by the update. A more complex strategy would be to update the materialized results themselves. There are two ways of updating materialized results: by recomputing the entire intermediate result or through incremental updates. Incremental updates only update the part of the result that changed due to the update. Recomputation is not feasible in a recycler context because, if it is worthy enough, the result would be materialized again the next time it occurs in a query. Incremental update on the other hand is not possible for all results. If the materialized result, for example, is an aggregation that computes the minimum of all values, deletion of the tuple that contributed the minimum, would make it impossible to obtain a new minimum without reevaluating the aggregation. Incremental updates can be performed eagerly when the update happens or lazily when the materialized result is used the next time. The latter one seems to be more appropriate for a recycler architecture.
Materialized view maintenance [GM95] is a related research area that deals with the problem of updating materialized views.

Increasing Subsumption Possibilities

The recycler only considers intermediate and final results for materialization that are produced while executing the current query. For each of these results, the query produces only the minimum amount of tuples required to answer the query. However, the recycler could also consider making queries larger and more expensive in order to increase the amount of subsumed future queries that benefit. It could for example create bigger range selections than required by the query or materialize entire data cubes and then use these to answer the current and future queries.

Caching Structures other than Tables

The recycler could not only be capable of sharing intermediate and final results between different queries, but also side products of the execution process. It could be extended to also share structures used within an operator. To reuse such a structure, the operation creating it and the one using it only have to partially match. The parameters for creating the structure must match between both operations. The structure can then be used different by both operations. An example of such a structure are hash tables. If two aggregations, for example, have to create the same hash table, that hash table can be shared between both of them even if both aggregations produce incompatible results. A more VectorWise specific example are selection vectors. Instead of applying the selection on each column, VectorWise generates a selection vector that indicates the positions in each column that contain qualifying tuples. Materializing selection vectors would allow to add additional columns to the materialized result of a selection without recomputing the selection.

Dynamically adapting the Recycler to the Workload

Different workloads exhibit a different degree of shared sub-expressions. Workloads that exhibit only a few overlapping queries need fewer results materialized than workloads with lots of sharing possibilities. If the workload does not contain a lot of overlap, the unused space in the recycler cache could better be returned to the buffer manager to buffer disk pages than reserving it for the recycler. The recycler could dynamically adapt the ratio between space reserved for the recycler cache and space used by the buffer manager depending on the characteristics of the workload.
Furthermore, some workloads benefit more from speculation because most shared sub-expressions have exactly two references and others benefit more from history. The recycler could dynamically favour the policy that suits best for the current workload.

Multi-layer Recycler

The performance of the recycler on other levels of the storage hierarchy like flash drives or even magnetic disks could be further investigated. From these observations, a multi-layer recycler could be constructed which uses multiple storage layers to store materialized results in. It could, for example, be faster to materialize the huge result of a selection on disk than recomputing it from the base table. The worth function would have to be adapted to include the decision on the storage layer an intermediate will be stored in.

A Subsumption

Materialized intermediate and final results can not only be used to answer queries that have to compute exactly the same result, but also for queries that have to compute a result which is a subset of the materialized result. Such a result subsumes the materialized result. In order to use the materialized result to answer the query, it first has to be refined. Because materialized result that exactly match do not need any refinement and therefore are cheaper to use, a recycler always checks for materialized results that exactly match first and only if no exact match is found, does it then try to identify results that can be subsumed.
This chapter will describe some additional sharing possibilities through subsumption. The following will assume that there is a materialized result M in the recycler and an intermediate result I in the current query. M is a superset of I and therefore can be used to compute I instead of computing it from scratch using base tables.

Column Subsumption

Column subsumption is the form of subsumption that is closest to exact match. M consists of all columns required by I as well as some additional columns that are not needed. If M has been produced from a previous intermediate result R by the aggregation $M = {}_{age}\mathbb{R}_{sum(salary),min(salary)}(R)$ and the query has to produce the result $I = {}_{age}\mathbb{R}_{sum(salary)}(R)$, the materialized result M can be used to answer the query. The intermediate result can be obtained by only scanning the required columns from the materialized result. Column subsumption does not need any further refinement. The recycler is able to handle this form of subsumption.

Tuple Subsumption

Tuple subsumption is the most typical form of subsumption. It was also described in [RSSB00, RRS+00, IKNG09]. M includes all tuples required by I. However, the operation that produced M was coarser than the one producing I and therefore M has additional tuples. To obtain I from M, the more fine-grained operation which produces I has to be applied on M to extract the required tuples.
Examples:
Selection: If the selection which produced M from the previous result R was $M = \sigma_{age<18}(R)$ and the query has to produce the result $I = \sigma_{age=16}(R)$, the result of I can be obtained using the materialized result: $I = \sigma_{age=16}(M)$. The result of a range selection can be reused to answer other range or equality selections if all selected tuples have also been selected by the materialized result.
Aggregation: If the aggregation which produced M from the previous result R was

$M = {}_{age,dno}\mathbb{G}_{sum(salary)}(R)$ and the query has to produce the result $I = {}_{age}\mathbb{G}_{sum(salary)}(R)$, the result of I can be obtained using the materialized result: $I = {}_{age}\mathbb{G}_{sum(salary)}(M)$. A materialized result of an aggregation can be used to answer other aggregations if their *Group By*-list subsumes the one of the materialized result and the operation allows subsumption. Furthermore, if R contains a hierarchy along a dimension, as is often the case in dimension tables from star schemes, the aggregation on the more fine-grained hierarchy level can also be used to answer the more coarse-grained one. If, for example, M was produced by $M = {}_{city}\mathbb{G}_{sum(revenue)}(R)$ and I is $I = {}_{country}\mathbb{G}_{sum(revenue)}(R)$, then I can be obtained by $I = {}_{country}\mathbb{G}_{sum(revenue)}(M)$.
After finding a materialized result that can be refined to answer a query, matching is usually stopped. If matching was to continue, the refinement would have to be applied on all other materialized results that are found further up the query tree. For some results, this is not possible. If, for example, the first subsumption found between two query trees is a range selection and the second one found is an aggregation that returned the average of all values, it is no longer possible to obtain the result of one of the averages from the other one.

Union

Union is not a form of subsumption, but it is related to it. If there is no result in the recycler that exactly matches with the query and none that can be subsumed, the recycler could combine several materialized results to obtain a result that can be used to answer the query. The recycler architecture presented in [IKNG09] is able to combine several results to answer a query.
If the recycler, for example, has the two selections $A = \sigma_{age<18}(Relation)$ and $B = \sigma_{15<age<25}(Relation)$ materialized and the query has to compute $\sigma_{10<age<20}$, the two materialized results can be combined and afterwards refined to answer the query: $C = \sigma_{10<age<20}(A \cup B)$.
In VectorWise, it would even be possible to efficiently obtain the result of an intermediate using one or more materialized results that only cover a large fraction of the selection range. VectorWise tracks the minimum and maximum value of each block. If there is a small range in the selection that is not covered by the results from the recycler, these values can be used to find all blocks that contain the required tuples. These tuples can then be combined with the materialized result.

Permutation

Permutation is another topic loosely related to subsumption. If optimized query trees are exactly matched node after node, permutations of the query tree are not considered. An example for permutations are different join orderings (join associativity). Using another permutation of the query tree might unveil additional materialized results that could be used and therefore might improve the execution time of the query. Multi-query optimizers like the one presented in [RSSB00] usually exploit different permutations of the same query.

Bibliography

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[Bon02] Peter A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, NL, May 2002.

[BZN05] PA Boncz, M. Zukowski, and NJ Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of International conference on verly large data bases (VLDB) 2005*, VLDB '05. Very Large Data Base Endowment, 2005.

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14:268–279, May 1985.

[CNR04] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 803–814, New York, NY, USA, 2004. ACM.

[CR93] ChungMin Melvin Chen and Nicholas Roussopoulos. The implementation and performance evaluation of the adms query optimizer: integrating query result caching and matching. Technical report, College Park, MD, USA, 1993.

[Fin82] Sheldon Finkelstein. Common expression analysis in database applications. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, SIGMOD '82, pages 235–245, New York, NY, USA, 1982. ACM.

[GM95] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Bulletin of the Technical Committee on*, 51:3, 1995.

[Gra94] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6:120–135, February 1994.

[IKNG09] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. An architecture for recycling intermediates in a column-store.

In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 309–320, New York, NY, USA, 2009. ACM.

[KR99] Yannis Kotidis and Nick Roussopoulos. Dynamat: a dynamic view management system for data warehouses. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 371–382, New York, NY, USA, 1999. ACM.

[LNEW04] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 791–802, New York, NY, USA, 2004. ACM.

[RR98] Jun Rao and Kenneth A. Ross. Reusing invariants: a new strategy for correlated queries. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 37–48, New York, NY, USA, 1998. ACM.

[RRS+00] Prasan Roy, Krithi Ramamritham, S. Seshadri, Pradeep Shenoy, and S. Sudarshan. Don't trash your intermediate results, cache 'em. *CoRR*, cs.DB/0003005, 2000.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 249–260, New York, NY, USA, 2000. ACM.

[Sel88] Timos K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Inf. Syst.*, 13:175–185, April 1988.

[SSV96] Junho Shim, Peter Scheuermann, and Radek Vingralek. Watchman: A data warehouse intelligent cache manager. 1996.

[SSV99] Junho Shim, Peter Scheuermann, and Radek Vingralek. Dynamic caching of query results for decision support systems. *International Conference on Scientific and Statistical Database Management*, 0:254, 1999.

[TGO01] Kian-Lee Tan, Shen-Tat Goh, and Beng Chin Ooi. Cache-on-demand: Recycling with certainty. In *Proceedings of the 17th International Conference on Data Engineering*, pages 633–640, Washington, DC, USA, 2001. IEEE Computer Society.

[ZBNH05] M. Zukowski, PA Boncz, NJ Nes, and S. Héman. MonetDB/X100-A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28:17–22, June 2005.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.