Total Operator State Recall — Cost-effective Reuse of Results in Greenplum Database

George C. Caragea*, Carlos Garcia-Alvarado†, Michalis Petropoulos* and Florian M. Waas*

Greenplum, A Division of EMC, 1900 S. Norfolk St. Ste. 305, San Mateo, CA 94403, USA

*⟨firstname.lastname⟩@emc.com
†carlos.garcia@emc.com

***Abstract*—Recurring queries or partial queries occur very frequently in production workloads. Reusing results or intermediates presents a highly intuitive opportunity for performance enhancements and has been explored to various degrees. Strategies suggested so far in the literature depend largely on speculative materialization of results in the hopes that they can be reused later on. However, a materialization is costly and conventional strategies run the risk that the initial investment cannot be amortized reliably unless the exact composition of the workload is known to the strategy *a priori*.**

In this paper, we develop an architecture for the reuse of intermediate query results that does not require a potentially costly up-front materialization. Rather, exploiting the fact that various query operations spill naturally to disk, we devise a management system that retains the operators' state in the form of the spill artifacts across queries and reuses them whenever appropriate. The resulting strategy is guaranteed never to perform worse than the original query in any significant way, yet, provides substantial performance boost whenever reuse of workfiles is possible.

We describe the architecture and the implementation of such a mechanism within a commercial database product and present preliminary performance studies.

I. Introduction

Query workloads generated by BI tools or reporting applications contain a significant number of recurring queries or partial query expressions used as building blocks for more complex queries (cf. [8], [3]). Very often a small number of reports is frequently executed either as exactly the same set of queries or with slight variations, *e.g.*, parameterized with different values. In addition, typical data warehouse schemas such as *star* or *snowflake schemas* require frequent "assembly" of information from dimension tables that is repeated in a wide variety of otherwise completely unrelated queries.

In data analytics as well as data warehousing, the dominant cost factor is the time and resources needed to execute analytics queries and reports. Vendors constantly strive to improve the response time of workloads and are eager to reduce the amount of resources required to compute query results. Reusing intermediate results across a workload enables cutting short the execution of queries by replacing a potentially highly compute and I/O intensive execution of a partial query with the results of a previous computation. Depending on the degree of overlap between queries, reusing results across a workload offers significant opportunities for performance improvements as well as resource savings. While an exact quantification is usually not available, in our experience the query-overlap in typical data warehouse workloads is significant, as common sub-expressions are recurring frequently.

The idea to reuse intermediates is highly intuitive and has been explored in various forms in the past, see *e.g.* [5], [8]. A variety of solutions has been proposed that requires the explicit declaration of query expressions to materialize. However, any materialization of intermediates poses a significant cost by itself—both in terms of time and disk space—that needs to be recovered by the reuse of the data set. In order to decide dynamically what results to materialize, any solution has to automate the cost-benefit analysis for the non-trivial cost of materialization and the expected amortization of it. Naïve approaches that materialize in an overly eager way are counterproductive as they cannot recoup the cost of materialization and, instead of improving the query times and resource usage, slow execution down at a considerable rate.

In this paper, we present an approach that does not require any up-front investment. In constrast to materializing speculatively and hoping to be able to recover the cost of materialization, we exploit natural materializations that occur in a variety of operators in the form of *workfiles*. In conventional query execution these files are discarded at the end of a query. Instead of deleting them, we retain and manage them. Any subsequent query is compared against the retained workfiles and in case of a match the workfiles are used rather than executing the corresponding partial query. The workfiles are retained based on their respective usage patterns and least-recently-used files are deleted if needed.

We propose an architecture to cache workfiles adaptively. The system consists of an augmentation of existing query execution mechanisms and a management component that indexes workfiles and manages available diskspace. The decision to reuse intermediates is done at execution time whenever a match between previously created intermediates and the current query is detected. In its current implementation we consider only exact matches of sub-expressions. As the experiments show, even with exact matches a significant performance improvement for representative workloads can be accomplished.

The remainder of this paper is organized as follows. In Section II we briefly outline the mechanics of workfiles and their use in modern-day query execution engines. Section III contains a detailed exposition of our proposed architecture

and its implementation. In Section IV we present a variety of experiments to illustrate the effectiveness of the technique and quantify the results. We discuss related work in Section V and conclude with Section VI.

II. Preliminaries

Workfiles, or *spill files*, are a fundamental concept in query processing and implemented in one form or another in all commercial database systems [7]. We briefly review the basics to the extent needed for the recall mechanism that we will introduce in Section III.

A. Operators

As part of the query processing life-cycle, a declarative query as expressed in a query language like SQL is turned into a query plan consisting of *operators* (cf. *e.g.* [7]). For example, Figure 1 shows the query plan for the following SQL query against the well-known TPC-H schema [2]:

```
SELECT c.cust_key, SUM(o.total_price)
FROM customers c, orders o
WHERE c.cust_key = o.cust_key
GROUP BY c.cust_key
```

Query plans are typically trees of operators; in special cases they could also be DAG's. Individual operators encapsulate functionality of an extended relational algebra. The abstraction of query processing logic into cleanly delineated operators has proven highly effective and provides a number of benefits from an engineering point of view, including the ability to combine operators into complex query plans and reasoning over input and output streams. Internally, operators may implement complex logic for processing and combining tuples and leverage highly sophisticated data structures in the process.

Operators can be implemented elegantly as self-contained state machines with a simple, yet powerful API for data ingest and egress. One of the most popular interfaces in commercial database technology is the *iterator model* that processes one tuple at a time; other implementation choices are possible and have been pursued in different systems. For our purposes, the actual API is not material, however.

B. Workfiles

A workfile is the intermittent materialization of the state of an operator—in its entirety or in part—written to stable storage. Operations like *hash-based join (HashJoin)* or *hash-based aggregation (HashAgg)* need to process potentially large amounts of data before emitting a single result tuple. The data structures needed to maintain the internal state, in this case hash tables, may not fit into the available assigned main memory. By structuring the algorithm so it can process batches and writing out large chunks of data that are not currently used, the operator can function fully within a small, predefined amount of memory at the cost of additional I/O. This technique is usually significantly more effective than relying on the virtual memory subsystem to swap in and out pages as the OS is unaware of the semantics of pages and their access patterns.

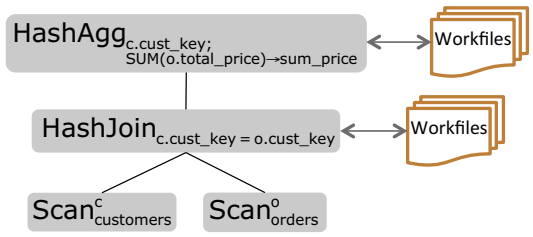


Fig. 1: Operator tree with workfiles

Operators most commonly implemented to spill, *i.e.*, to temporarily store state in workfiles, are HashJoin, HashAgg, Sort, and Spool. In the following we will focus on these only. However, the use of workfiles is not restricted to a particular set of operators and the use of workfiles in other memory-intensive operators is conceivable.

In the above list, Spool is distinguished as it consumes and buffers all of its input before emitting the first output tuple but does not perform any actual data processing unlike the others. In other words, the Spool operator scans its input and places a copy of each row in a table. Spool is the only operator whose state corresponds to the intermediate results of the subplan at this point. The other operators store state in the workfiles that reflects some stage of processing in a very operator specific layout, *e.g.*, data sorted in runs, layout of a hashing scheme, or data partitioned into hash table buckets. Also, workfiles may consist of multiple units, that is, consist of a set of files per operator. We refer to these as *workfile set* in the following.

C. Management of Workfiles

In most database systems, workfiles are created on demand at the point in time when the internal memory allotted to an operator is exhausted. Depending on the algorithm implemented, the number of workfiles may increase during processing: some workfiles may be discarded or rewritten. In addition, commercial implementations need to meet strong robustness requirements and for example guarantee the total resources assigned to workfiles stay within certain bounds and workfiles are deleted after the query is complete or the query has been terminated unexpectedly.

In summary, workfiles represent a form of intermediate result while processing a query. However, in most cases, they represent data in a processed form that contains additional algorithmic information. Depending on the operators in the subplan, the workfiles of an operator encode the results of a subplan rooted in this operator. These are the scenarios we will be addressing in the remainder of this paper.

III. Caching of Workfile Sets

Figure 2 presents the architecture of the *workfile cache* that consists of two components: the *workfile depot* that stores on disk the actual workfiles generated by operators in query plans, and the *workfile metadata* that is an in-memory

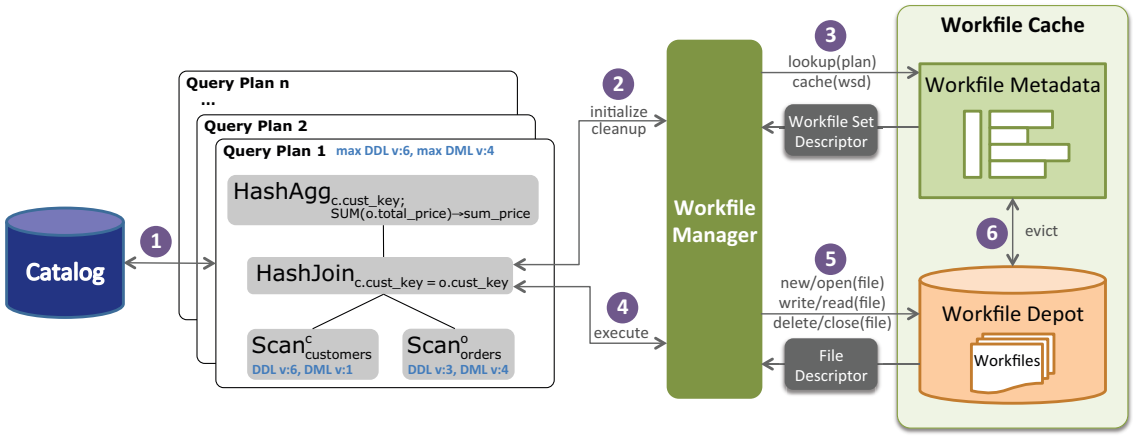


Fig. 2: Architecture of Operator Recall Framework

structure describing the contents of the workfile depot. These components and their interaction will be explained below using the same query plan as in Figure 1.

A. Workfile Manager

As shown, a query plan does not interact directly with the workfile cache, but rather through the *workfile manager*. The workfile manager is the API between the operators on one side, and the workfile cache on the other side. The reason for this additional layer is to abstract out the implementation of the workfile depot from the operators in the query plan and hence make it easily interchangeable. Moreover, the workfile metadata remains a rather simple data structure since it is agnostic of the specific workfile depot implementation used because the workfile manager is responsible of keeping the two components consistent.

The workfile manager is utilized during the *initialize*, *execute* and *cleanup* stages of each operator, as dictated by the iterator model of query execution.

Initialize. Each operator in a query plan knows if the subplan rooted at it is *deterministic*, that is, no volatile functions are used in the subplan, for example. This operator property is inferred during the query optimization phase and by consulting the catalog (Step 1 in Figure 2), and it is stored within the query plan. If the subplan is deterministic, the operator asks the workfile manager to look up if a reusable workfile set is cached for the given subplan (Step 2). The workfile manager is looking for the exact same subplan and does not consider containing subplans. If one is found in the cache, the workfile manager provides the operator with a *workfile set descriptor*, defined in Section III-B, and pins it in the workfile metadata so that it is not evicted by other operators within the same or other query plans during query execution (Step 3). If the subplan is not deterministic, the operator does not attempt to reuse existing workfile sets, or cache any for future use.

Execute. When reusing an existing workfile set, the operator utilizes the provided workfile set descriptor to identify the filenames that comprise the reusable set and asks the workfile manager to open them and start reading from them (Step 4). In turn, the workfile manager accesses the workfile depot and provides the cached data to the operator (Step 5). If a workfile set is not found in the cache and the operator needs to spill, then the operator creates a new workfile set descriptor and asks the workfile manager to add and write to new files in the workfile depot. Writing to new files may require evicting one or more existing workfile sets from the cache (Step 6) due to space limitations. The workfile metadata is agnostic of new workfile set descriptors until the cleanup stage.

Cleanup. When reusing a workfile set, the operator notifies the workfile manager, which un-pins it in the workfile metadata. If the operator has created a new workfile set that it is reusable, then it asks the workfile manager to add the corresponding descriptor to the workfile metadata. The corresponding files are already in the workfile depot. Adding a new workfile set to the workfile metadata may require evicting one or more existing workfile sets (Step 6). If the new workfile set is not reusable, that is, the subplan is not deterministic, the workfile manager deletes the corresponding files in the workfile depot and does not alter the workfile metadata in any way.

B. Workfile Metadata

The workfile metadata maintains a hash table that stores workfile set descriptors. The key of each workfile metadata entry is generated by hashing the subplan rooted at an operator, while the value is a workfile set descriptor that comprises:

- a common prefix of all workfiles, uniquely generated for each operator in a query plan
- the number of workfiles in the set
- the type of workfiles in the set, such as, compressed
- additional operator-specific metadata required to reuse the workfile set

Using the information captured in the descriptor, the workfile manager can locate the corresponding workfiles in the depot and provided the content to an operator.

The workfile metadata is shared among all query plans that are executed in parallel at any given point in time and supports high levels of concurrency. Every operator of every query plan

that can reuse workfile sets performs a lookup before starting execution, and possibly an insert at the end of execution.

C. Cache Invalidation

To compute the hash code of a subplan, the workfile manager serializes the subplan. The structure and the metadata accessed by the plan is not sufficient information though to determine whether a workfile set is reusable in the future, since the database schema or instance might change between the creation and possible reuse of workfiles. For that reason, the maximum *DDL version* and *DML version* observed in a plan also need to be added to the hash code.

Each metadata item accessed by the plan has a given DDL (schema) and a DML (instance) version recorded in the catalog that doesn't change for the duration of the transaction. These versions are monotonically incremented and are unique across all metadata items in the catalog when a DDL or a DML command is executed. Recording in the hash code the maximum DDL and DML versions across all metadata items in the plan sufficiently identifies the point in time that the workfile set was created. As an example, consider the query plan in Figure 2, where the maximum DDL version comes from the `customers` table and the maximum DML comes from the `orders` table. If right after the execution of the query plan in Figure 2, a DML command is executed targeting the `customers` table, the DML version of `customers` will become 5. If at this point the query plan in Figure 2 is re-executed, it will not reuse the cached workfile set, because their maximum DDL and DML versions do not match.

The workfile manager uses versioning as a passive invalidation scheme for the cache. Instead of actively invalidating existing entries, the workfile manager admits new entries with higher DDL and DML versions. Overtime, cache entries with stale versions will not be used and an LRU eviction policy will age them automatically out of the cache over time.

D. Cache Admission

For each new and reusable workfile set, the workfile manager consults the *admission policy*, which determines if the set is to be cached or not even though the corresponding subplan is deterministic. The factors that are taken into consideration are (i) disk space allowed for total cached workfiles, and (ii) disk space used by the new workfile set. A system configuration parameter determines the maximum amount of storage that can be used to store cached workfile sets. To avoid situations where a large portion of the cache has to be evicted in order to accommodate a new entry, a maximum size limit is set for admitting entries in the cache. If the total size of workfiles in a set is larger than 20% of the total disk space allocated for the workfile depot then the workfile manager does not add the set to the cache. If a newly generated workfile set does not meet this criteria, its files will be deleted from disk at the end of the query, as is the case when a workfile set is not cacheable.

E. Preserving Execution Semantics

The workfile manager ensures that the presence of the cache does not alter the execution semantics of a query plan.

Operator	Queries in which operator spills
HashJoin	2, 7, 8, 9, 10, 12, 13, 18, 21, 22
HashAgg	2, 10, 13, 16, 18, 20
Sort	9
Spool	-

TABLE I: Occurrance of spilling operators in TPC-H

Consider the following example: the size of the workfile depot is 100 GB. Currently running queries are using 30 GB for their workfile sets. The workfile depot stores 50 GB of cached, not currently reused by any operator, workfile sets. This leaves 20 GB of free disk space for new workfile sets. A new query starts executing, and an operator starts creating workfiles. After creating 20 GB of workfiles, the workfile manager realizes that the depot is out of disk space. At this point, to ensure that this query's execution can succeed, the workfile depot will trigger evictions from the workfile metadata and delete workfile sets from the depot. After a certain amount of disk space is freed up, the operator execution resumes and it continues creating workfiles. Note that if the operator creates more than 70 GB of workfiles, the query will fail with an "out of disk space" error, which is consistent with the regular behavior.

IV. Quantitative Analysis

We implemented the architecture described in Section III within the Greenplum Database [1]. We added new workfile manager and workfile cache components, and we augmented all the four operators capable of spilling (HashJoin, HashAgg, Sort and Spool) with new functionality that enables them to query the workfile metadata and "recall" their state from a set of workfiles stored in the workfile depot.

To evaluate the performance of the optimization, we conducted experiments using the TPC-H Decision Support Benchmark [2]. For our experiments, we used a database with of a scale factor of 10 (SF10), referring to a size of approximately 10 GB. Our tests were executed using a dual-processor quad-core Intel Xeon 2.4GHz with 16 GB RAM running RedHat Enterprise Linux. Note that the workfile depot disk size threshold was not reached for any of the experiments described below, and therefore no evictions were triggered. Evaluating the performance of the cache replacement algorithm is outside the scope of this paper.

In our first experiment, we evaluated the potential improvements of the recall scheme under ideal conditions. For this, we ran each of the 22 TPC-H queries when the workfile depot is fully populated, i.e. when every workfile metadata look-up resulted in a hit. As some of the queries did not create any state files on disk, they did not benefit at all from the existence of the workfile cache. For these queries, the running time overhead was negligible, as described in the next experiment. From the set of 22 queries, twelve created workfiles and benefited from the cache. Table I summarizes the operators that interacted with the workfile manager in these queries.

Figure 3 shows the performance improvement for all the TPC-H queries relative to the existing system. The speedup varies from 1.18 for query 22 to 39.92 for query 8. For only

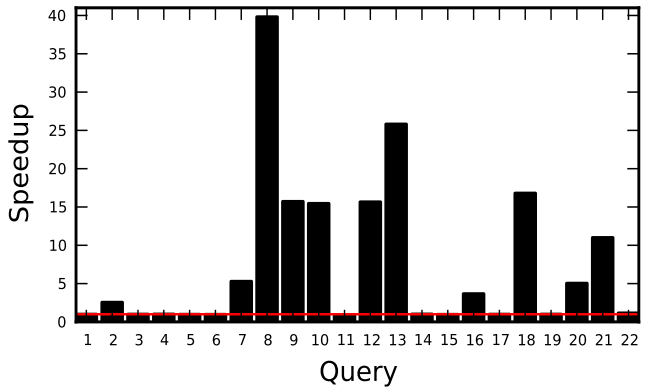


Fig. 3: Speedup for TPC-H queries with fully populated workfile depot

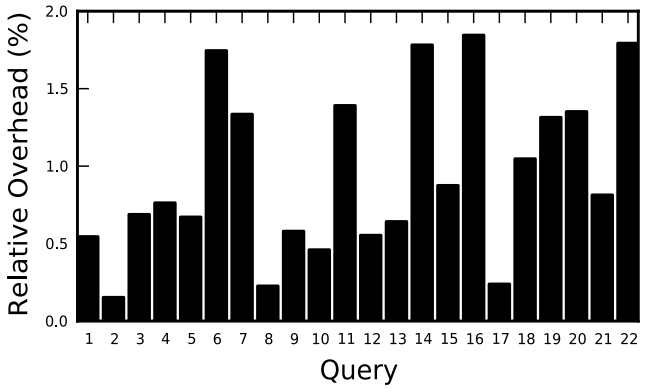


Fig. 4: Runtime overhead for TPC-H queries with empty workfile depot

the queries that spill, the total speedup is 8.45, while across the entire 22 query workload we observed a 1.95 speedup.

Next, we investigated the overhead imposed by our implementation. As operators already create state files as needed in the current system, the only additional costs are incured from (i) the management of the workfile metadata (look-ups, inserts and evictions) and (ii) writing additional metadata to disk to be able to recall the operator state at a later time and (iii) monitoring disk space usage in the workfile depot.

To evaluate the overhead, we ran all the 22 TPC-H queries using two instances of the database: (i) with the feature disabled, where operators fall back on their default behavior with no recall and (ii) the system with recall enabled but with an empty workfile depot. This is the least favorable case where we incur all the overhead of the scheme with none of the benefits.

The results of this experiment are shown in Figure 4. For the individual queries, query 16 shows the highest overhead of 1.85% additional execution time. Most queries exhibit overhead of less than 1%, and for the entire 22 queries in the workload, the running time is only 0.82% higher with the workfile cache enabled. We consider such overhead negligible in practice for analytical and data warehousing workloads.

Finally, we evaluated the impact of the cache for a synthetic workload meant to emulate a real-life analytical setup. As shown in [12], workloads running over a database by various reporting and BI workloads exhibit a high percentage of partially overlapping queries. We used the TPC-H benchmark queries to construct a similar workload in order to evaluate the improvements of the caching scheme in production.

The following methodology was used to create the mixed workload. We generated five instances for each of the 22 query templates in the TPC-H workload, by plugging in randomized sets of parameters using the QGEN tool. We then produced a random sequence of queries from this set where each instance appears exactly three times, resulting in a stream of 330 queries.

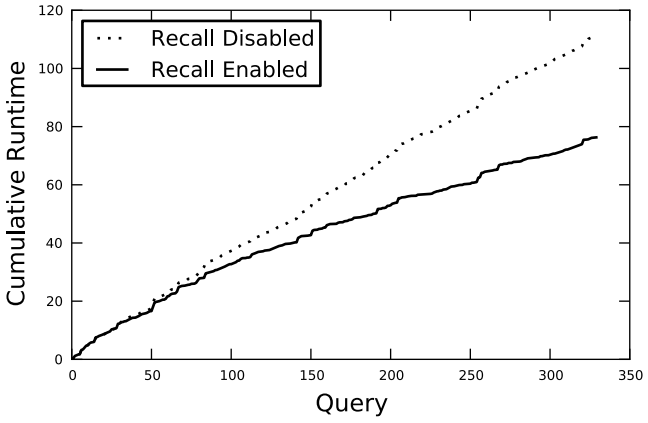
In Figure 5a we compared the workload running time with and without caching. While the workfile depot is empty, there is no perceivable benefit from the caching scheme. As the depot is being populated, the query response time starts to decrease as more queries recall saved state. Figure 5b displays the filling rate of the workfile depot. The depot is fully populated after query 262, resulting in a most spilling queries using recalled operator states. For the entire workload of 330 queries, the speedup relative to no caching is 1.47, a significant improvement accomplished with no user or DBA intervention.

V. Related Work

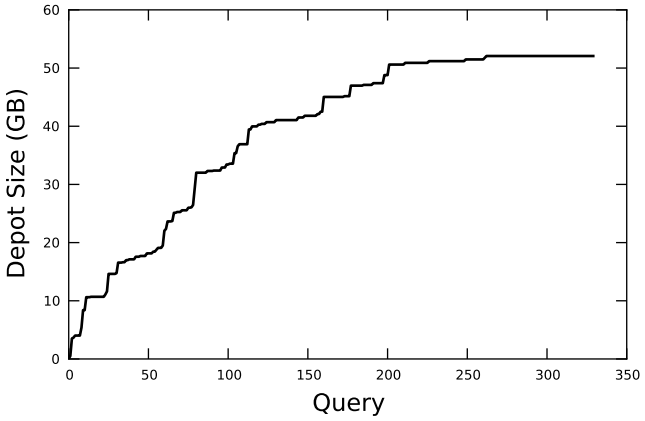
A significant amount of work has been done to reuse results of analytical queries. Early attempts focused on caching the entire query results [11]. These attempts evolved into intermediate result reutilization, and include materialized views, intermediate result spooling, and operator level spilling [13], [10], [8]. Nevertheless, it has regularly been considered that managing intermediate results adds a large amount of overhead that yields a small benefit.

Early work on materialized views [5] centered around modifying the query optimizer to produce cheaper plans with materialized views. The decision of which intermediate results to cache was left as an open problem later explored in [9], [4], [13]. These papers represent industry efforts that made available capturing intermediate results into production systems. However, the main problem with materialized views is that intermediate results have to be explicitly selected. This selection problem leads to the materialization of intermediate results based on historical workloads that cannot adapt to new workloads. In our approach, we adapt to changing workloads by maintaining relevant spill files based on our eviction policy.

The usage of spooling (write an intermediate result into a temporary table) is an alternative approach explored in [10], [6]. In [10], a window-based algorithm is proposed for storing and reusing intermediate results. The depot is maintained through a benefit estimation analysis in a sliding window. A multi-graph is used to represent the content of the depot. In addition to the spooling, we do not add the overhead of building a DAG to decide which intermediate results to store, since our approach relies on the inherent characteristics of analytical queries that reuse results frequently. In [6], the authors proposed an intermediate result reutilization framework



(a) Cumulative runtime for a mixed workload with and without recall



(b) Size of workfile depot on disk

Fig. 5: Performance and depot size for a mixed workload

for multi-query optimization called 'cache-on-demand'. This approach focuses on spooling the results of joins that are known to be reused. Unlike the previous approach, we focus on reusing spill files of memory-intensive operators that represent a much cheaper operation than spooling. The drawback of using spool files has limited their applicability in production systems.

Finally in [8], the authors propose a variety of algorithms for dealing with the reusage of intermediate results in MonetDB. Their approaches reuse intermediate results of partial materializations, which result in some overhead due to their operator-based 'intermediate result copy' that had to be included into their architecture. In addition, their approach for intermediate result reutilization does not support pipelined execution, which is the dominant model for most commercial database systems.

In this work, we focus on extending database systems to take advantage of workfiles that have to be computed to the granularity level of memory-intensive operators. Since workfiles are computed regardless of the decision of keeping them, the only perceivable overhead is in their management.

VI. Summary

Reusing results in the form of intermediates or entire query results is known to increase performance and query throughput in query processing drastically. While extremely intuitive in principle, realizing efficient reuse mechanisms in a commercial database system is a challenging task: modern query execution engines do not materialize intermediates after each operation, hence, preserving state is tied to a certain cost. A strategy is only successful if the cost is amortized by re-using the saved state. Moreover, mechanisms based on historic query patterns are of limited use in ad-hoc query scenarios where the workload is not known *a priori*.

In this paper, we presented an architecture that reuses workfiles that are a natural by-product of most query engines anyways. A variety of frequently used operations spill to disk in the form of temporary workfiles. By caching and re-using these workfiles the execution of queries that contain the exact recurring pattern can be cut short resulting in a reduced query response time. As our experiments with an implementation in a commercial database system show, the proposed mechanisms can lead to substantial savings in typical analytical query workloads. Our mechanism is fully self-managing and does not require any guidance be it from administrators or tuning advisor tools.

In the future, we plan to extend our methodology to enable query subsumption instead of requiring exact matches between the incoming query and the existing workfiles.

References

[1] "Greenplum MPP Database," http://www.greenplum.com.
[2] "TPC Benchmark^{TM} (TPC-H): Decision Support," 1999.
[3] "TPC Benchmark^{TM} (TPC-DS): The New Decision Support Benchmark Standard," 2012.
[4] S. Agrawal, S. Chaudhuri, and V. Narasayya, "Automated selection of materialized views and indexes in SQL databases," in *Proc. of VLDB Conference*, 2000, pp. 496–505.
[5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *Proc. of IEEE ICDE Conference*, 1995, pp. 190–200.
[6] S. Goh, B. Ooi, and K. Tan, "Demand-driven caching in multiuser environment," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 1, pp. 112–124, 2004.
[7] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Comput. Surv.*, vol. 25, no. 2, pp. 73–170, 1993.
[8] M. Ivanova, M. Kersten, N. Nes, and R. Gonçalves, "An architecture for recycling intermediates in a column-store," *ACM Transactions on Database Systems*, vol. 35, no. 4, pp. 24:1–24:43, 2010.
[9] K. Ross, D. Srivastava, and S. Sudarshan, "Materialized view maintenance and integrity constraint checking: Trading space for time," in *ACM SIGMOD Record*, vol. 25, no. 2, 1996, pp. 447–458.
[10] A. Safaeei, M. Kamali, M. Haghjoo, and K. Izadi, "Caching intermediate results for multiple-query optimization," in *IEEE/ACS International Conference on Computer Systems and Applications*, 2007, pp. 412–415.
[11] P. Scheuermann, J. Shim, and R. Vingralek, "Watchman: A data warehouse intelligent cache manager," in *Proc. of VLDB Conference*, 1996, pp. 51–62.
[12] V. Singh, J. Gray, A. R. Thakar, A. S. Szalay, J. Raddick, B. Boroski, S. Lebedeva, and B. Yanny, "Skyserver traffic report the first five years," Microsoft Research, Tech. Rep., December 2006.
[13] D. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. Lohman, R. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, and et al., "Recommending materialized views and indexes with the IBM DB2 design advisor," in *Proc. of IEEE International Conference on Autonomic Computing*, 2004, pp. 180–187.