Revisiting Reuse in Main Memory Database Systems

Kayhan Dursun
Brown University

Carsten Binnig
Brown University

Ugur Cetintemel
Brown University

Tim Kraska
Brown University

ABSTRACT

Reusing intermediates in databases to speed-up analytical query processing was studied in prior work. Existing solutions require intermediate results of individual operators to be materialized using materialization operators. However, inserting such materialization operations into a query plan not only incurs additional execution costs but also often eliminates important cache- and register-locality opportunities, resulting in even higher performance penalties. This paper studies a novel reuse model for intermediates, which caches internal physical data structures materialized during query processing (due to pipeline breakers) and externalizes them so that they become reusable for upcoming operations. We focus on hash tables, the most commonly used internal data structure in main memory databases to perform join and aggregation operations. As queries arrive, our *reuse-aware optimizer* reasons about the reuse opportunities for hash tables, employing cost models that take into account hash table statistics together with the CPU and data movement costs within the cache hierarchy. Experimental results, based on our prototype implementation, demonstrate performance gains of 2× for typical analytical workloads with no additional overhead for materializing intermediates.

1. INTRODUCTION

Motivation: Reusing intermediates in databases to speed-up analytical query processing has been studied in the past [16, 29, 22, 14, 9, 32]. These solutions typically require intermediate results of individual operators be materialized in memory during query processing to be considered for reuse in subsequent queries. However, these approaches are not optimal for modern main memory databases. First, inserting additional materialization operations into a query plan results in an additional overhead that first needs to be amortized by subsequent queries that can reuse the materialized intermediates. Second, modern main memory DBMSs are typically heavily optimized for cache- and register-locality [17, 26, 23] and therefore attempt to minimize pipeline breakers.

To this end, adding additional materialization operations into a query plan not only adds additional traffic to the memory bus but, more importantly, also prevents important cache- and register-locality, which results in performance penalties. Consequently, the benefits of materialization-based reuse techniques heavily depend on the characteristics of the workload; i.e., how much overlap between queries of a given workload exists. In the worst case, if the overlap is low, then the extra cost caused by materialization operations might even result in an overall performance degradation for analytical workloads.

SIGMOD'17, May 14 - 19, 2017, Chicago, Illinois, USA

ISBN 978-1-4503-4197-4/17/05...$15.00

DOI: http://dx.doi.org/10.1145/3035918.3035957

The goal of this paper is to revisit "reuse" in the context of modern main memory databases [17, 1, 6]. The main idea is to leverage internal data structures that are already materialized by pipeline breakers during query execution. This way, reuse comes for free without any additional execution costs. Moreover, as we will show in our experiments, result reuse becomes more robust towards workloads with different reuse potentials and provides benefits for a wider range of workloads even if the overlap between queries is not so high.

In this paper, we present a new main memory database system called *HashStash* that reuses internal data structures. The focus of this work is on the most common internal data structure, hash tables (HTs), as found in hash-join and hash-aggregate operations. We leave other operators and data structures (e.g., trees) for future work.

Contributions: To the best of our knowledge this is the first paper that studies the reuse of internal data structures for query processing. As a major contribution, we present a new system called *HashStash* that extends a classical DBMS architecture to support the reuse of internal hash tables. The architecture of *HashStash* supports two reuse models:

(1) Single-query Reuse: In this re-use model, users or applications submit a single query to a *HashStash*-based DBMS just as in normal DBMSs. However, different from a classical DBMS, a *HashStash*-based DBMS identifies the best *reuse-aware plan* that leverages existing intermediate hash tables. To support this model, we extend the DBMS architecture by three components: (a) *a cache for hash tables* that keeps lineage and statistics information, (b) *a reuse-aware optimizer* that uses new operator cost models and enumeration strategies to determine which hash tables should be reused by which operators in order to minimize the total query runtime, and finally (c) *a garbage collector* that evicts hash tables from the cache as needed.

(2) Multi-query Reuse: Many analytical applications today execute multiple queries concurrently to analyze and report different aspects of the same data set. In order to support multiple queries that are submitted at the same time, we leverage the concept of shared plans as introduced in [11, 10] and extend them in the following directions: (a) we develop *shared reuse-aware plans*, i.e., shared plans can also re-use the hash tables in *HashStash* and (b) we extend the optimizer in *HashStash* to create optimal reuse-aware shared plans for a given batch of queries.

We evaluate the performance of *HashStash* under workloads with different reuse potentials. Our experiments show that *HashStash* outperforms materialization-based reuse for any of these workloads independent of the reuse potential.

While our results are focused on main memory databases, the reuse of internal data structures could also be applied in classical disk-based database systems. In these systems, the main bottleneck is typically disk I/O and thus effects such as register- and cache-locality play a less important role. However, since reuse of internal data structures avoids the re-execution of sub-plans, we still expect to see performance gains when applying our techniques in a disk-based database system. Only in cases where the reuse potential is low, the additional memory overhead for storing internal data structures might have a negative impact since the database buffer then has less space to cache pages from the base tables in memory, which might cause an overall slowdown.

Outline: The rest of this paper is structured as follows. Section 2 gives an overview of our suggested *HashStash*-based architecture to support single-query and multi-query reuse. Section 3 and Section 4 then present the details for each of these reuse cases and discuss novel optimization strategies to support them. Afterwards, Section 5 discusses how garbage collection works in *HashStash*. Section 6 presents our evaluation of our *HashStash* prototype. Finally, Section 7 discusses related work and Section 8 concludes with a summary and outlines potential future work.

2. HASHSTASH OVERVIEW

The main goal of *HashStash* is to leverage internal hash tables for reuse that are materialized during query execution. To achieve this, in *HashStash* we add the following components to a classical DBMS architecture (see Figure 1): (1) a *Reuse-aware Query Optimizer* (RQO) that replaces the traditional (non-reuse-aware) optimizer, and (2) a *Hash Table Manager* (HTM) that consists of a cache of hash tables and a garbage collector. In the following, we discuss each component individually and then present an example to illustrate the main ideas of *HashStash*.

2.1 Reuse-Aware Query Optimizer

The Reuse-Aware Query Optimizer (RQO) offers two interfaces for compiling and optimizing queries: a query-at-a-time interface for single-query reuse and a query-batch interface to support multi-query reuse.

Query-at-a-time Interface: This interface accepts a single query and returns an optimized reuse-aware execution plan. The main goal of the reuse-aware optimizer is to decide which hash tables in the cache to reuse such that the overall query execution time is minimized. In order to select the best reuse-aware execution plan, the reuse-aware optimizer enumerates different join orders and decides for each plan which is the best reuse case based on the hash tables in the cache. Different from a traditional query optimizer, our reuse-aware optimizer implements two important extensions: (1) In order to decide which hash table to reuse, the optimizer leverages the so called *reuse-aware cost models*. Different from normal cost models, reuse-aware cost models additionally take statistics of a candidate hash table into account in order to estimate the execution costs for the different reuse cases discussed before. (2) The reuse-aware optimizer implements benefit-oriented optimizations. The main intuition is that a plan is preferred over another if it creates hash tables that promise more benefits for future reuse even if the initial execution is slightly more expensive.

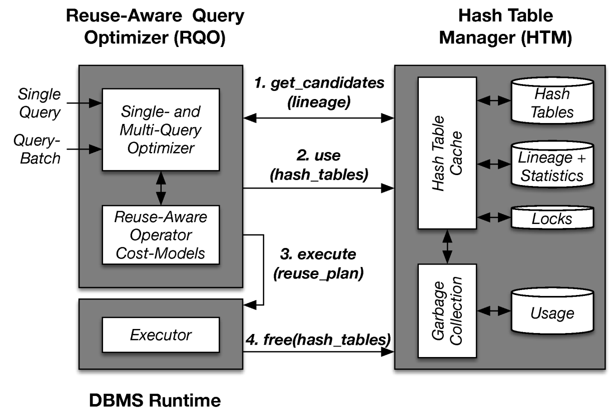


Figure 1: Additional *HashStash* Components

Furthermore, the *HashStash* optimizer supports four different cases for reuse-aware operators: *exact-*, *subsuming-*, *partial-*, and *overlapping-*reuse. This is different from the existing approaches in [16, 29, 22], which only support the *exact-*reuse, and the *subsuming-*reuse cases. The *exact* case enables a join or aggregation operator to reuse a cached hash table which contains exactly the tuples required by the query. In that case, complete sub-plans might be eliminated (e.g., the one which build the hash table of a hash-join). Compared to the case before, the *subsumption* case is possible when the reused hash table contains more tuples than needed. This might lead to false-positives, which need to be post-filtered by an additional selection. The *overlapping* and the *partial* case are different. Both cases allow the reuse of a hash table where some tuples are "missing". These tuples are added by *HashStash* during query execution. To support all these different reuse cases the optimizer applies different rewrites rules during optimization.

Figure 1 shows how the reuse-aware optimizer is integrated into *HashStash*. First, the optimizer enumerates different join orders and retrieves candidate hash tables for reuse. Once the optimal reuse-aware plan is found, the optimizer sends the information regarding which hash tables will be reused to the HTM for book-keeping as shown in step 2 (Figure 1). Finally, the optimizer sends the reuse-aware plan to the executor as shown in step 3 (Figure 1). Once the plan execution is finished, the DBMS runtime informs the HTM to release all used reused hash tables as shown in step 4 (Figure 1), which make them available for garbage collection for instance. Details about the query-at-a-time interface are described in Section 3.

Query-Batch Interface: The query-batch interface is different from the query-at-a-time interface since it accepts multiple queries submitted as a batch. Different from the query-at-a-time interface, subsets of queries submitted in the same batch can share the same execution plan; called *reuse-aware shared plan* in the sequel. The main difference of *HashStash* than the approach presented in [11] is to integrate the before-mentioned reuse techniques into shared plans. In order to find the best reuse-aware shared plan, we developed a novel reuse-aware multi-query optimizer that merges individual reuse-aware plans using a dynamic programming based approach. Details about the query-batch interface are described in Section 4.

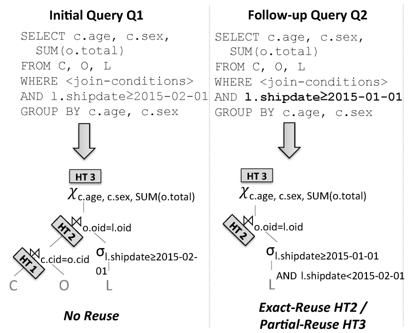


Figure 2: Reuse Example

2.2 Hash Table Manager

The two components of the Hash Table Manager (HTM) are the hash table cache and the garbage collector.

Hash Table Cache: The hash table cache manages hash tables for reuse; it stores pointers to cached hash tables, as well as lineage information about how each one of them was created. It also stores statistics to enable the cost-based hash table selection by the optimizer. For our initial prototype, we allow only one query to reuse a hash table in the cache at a time (except for the query-batch interface). However, for future work, we also plan to look into sharing the same hash table between concurrent queries.

Garbage Collector: The main goal of the garbage collector is to decide which hash tables to evict. The cache triggers the garbage collector when no more memory is available to admit new hash tables. Therefore, the garbage collector maintains usage information and implements eviction strategies to decide which hash tables to remove.

2.3 Reuse Example

Figure 2 illustrates a reuse example for a sequence of three queries from a data exploration session. The initial query $Q1$ executes an aggregation over a 3-way join of the tables `Customer`, `Orders`, and `Lineitem` for all lineitems shipped after `2015-02-01`. For this query no reuse is possible (since it is the initial one). However it materializes all three hash tables HT1-HT3 in the cache of *HashStash*.

The follow-up query $Q2$ then executes a query that differs from $Q1$ only in the filter predicate; i.e., it selects lineitems that shipped after `2015-01-01`. In order to execute $Q2$, *HashStash* can reuse hash table $HT2$ (exact-reuse) and thus avoids to recompute the join of `Customer` and `Orders`. Moreover, the hash table $HT3$ produced by the aggregation of $Q1$ can also be reused to compute the aggregation operator in $Q2$. However, since $HT3$ does not aggregate all required lineitems (due to partial-reuse), the base table `Lineitem` needs to be re-scanned for the "missing" tuples between `2015-01-01` and `2015-02-01`. These tuples are added to $HT3$ by the reuse-aware plan of $Q2$.

Finally, the last query $Q3$ is similar to $Q2$. The only difference is that it removes the group-by attribute `c.age`. For executing $Q3$, *HashStash* can directly reuse the hash table $HT3$ (exact-reuse). However, due to the removed group-by attribute, a post-aggregation operator needs to be added.

3. SINGLE-QUERY REUSE

In this section, we describe how to find the best reuse-aware execution plan for the query-at-a time interface. As discussed before, finding the best reuse-aware plan is implemented by the optimizer of *HashStash*. Therefore, we first give an overview of how the plan enumeration procedure of our optimizer works and then discuss the cost models of our reuse-aware hash-join and hash-aggregate operator. Afterwards, we present the details on how the matching and rewriting procedures in *HashStash* work to enable the different reuse-cases (exact, subsuming, partial, and overlapping). Finally, we discuss some benefit-oriented optimizations that increase the effect of reuse by spending initially a little more execution cost to create "better" hash tables.

3.1 Reuse-aware Plan Enumeration

The plan enumeration algorithm in *HashStash* can be applied to complex nested SQL queries. In order to simplify the presentation, we first show the basic procedure that only enumerates different join plans for a given SPJ (select-project-join) query. More complex queries including aggregations and nesting are discussed at the end of this section.

Basic Procedure: Algorithm 1 shows the basic recursive procedure for enumerating different reuse-aware plans for SPJ queries based on a top-down partitioning search strategy. This procedure enumerates different partitions of a given join graph G where the partition function produces two connected sub-graphs (i.e., no cartesian products are enumerated). Similar to existing top-down strategies it memoizes the best plans and avoids that the same partitioning is evaluated twice (line 3). Different from existing top-down partitioning search strategies such as [8], Algorithm 1 additionally implements the following ideas to support reuse-aware plans: (1) when partitioning the join graph into a left and right partition of G, the algorithm enumerates the different candidate hash tables (including a new empty hash table) for the right partition G_r and the left partition G_l that can be reused for building the hash table of a hash join (line 8 to 16 and 19 to 27). (2) Another difference from existing top down enumeration algorithms is the ability to rewrite the respective sub-plan that would reuse a given candidate hash table (line 9 and 20). This rewrite possibly eliminates the complete sub-plan (i.e., in the best case G'_r (line 9) and G'_l (line 20) might become an identity operation over the reused hash table if an exact-reuse is possible). We discuss details of the rewrite procedure for all four different reuse cases (exact-, subsuming-, overlapping-, and partial-reuse) later in this section. (3) The last difference is that the cost estimation (line 13 and 24) uses the cost models for the reuse-aware join and aggregation operator to estimate the runtime costs when reusing a given candidate hash table.

Complex Queries: The general idea to support more complex queries is similar to exciting optimizers. First, nested queries are unnested using joins if possible. Second, if unnesting is not possible for the complete query, the enumeration procedure shown before is applied to each query block individually. In *HashStash*, query blocks can be in the form of either SPJ (select-project-join) or SPJA (select-project-join-aggregation) queries. In order to extend Algorithm 1 for SPJA queries, we only need to iterate over all candidate hash tables as well as an empty (new) hash table for the additional aggregation operator in the SPJA block and then apply the rewrite rules for SPJA queries (cf. Section 3.3).

3.2 Reuse-Aware Operators and Cost Models

In the following, we discuss the reuse-aware join and aggregation as well as their cost models for reuse.

Algorithm 1: Plan Enumeration in *HashStash*

```
Input   : Join Graph G of SPJ Query Q
Output: Reuse-Aware Execution Plan P
Algorithm getBestReusePlan(G)
if bestPlans[G] ≠ NULL then
    return bestPlans[G];
else
    foreach partition (G_l, G_r) in G do
        candHTs ← getCandHTs(subPlan(Q, G_r));
        candHTs ← candHTs ∪ createNewHT(G_r);
        foreach candHT ∈ candHTs do
            G'_r ← rewritePlan(G_r, candHT);
            P_l ← getBestReusePlan(G_l);
            P'_r ← getBestReusePlan(G'_r);
            curTree ← createPlan(P_l, P'_r, candHT);
            if cost(curTree) ≤ cost(bestPlans[G]) then
                bestPlans[G] = curTree;
            end
        end
        candHTs ← getCandHTs(subPlan(Q, G_l));
        candHTs ← candHTs ∪ createNewHT(G_l);
        foreach candHT ∈ candHTs do
            G'_l ← rewritePlan(G_l, candHT);
            P'_l ← getBestReusePlan(G'_l);
            P_r ← getBestReusePlan(G_r);
            curTree ← createPlan(P_r, P'_l, candHT);
            if cost(curTree) ≤ cost(bestPlans[G]) then
                bestPlans[G] = curTree;
            end
        end
    end
    return bestPlans[G];
end
```

3.2.1 Reuse-Aware Hash-Join

A reuse-aware hash-join (RHJ) works similarly to a traditional hash-join; i.e., the join first builds a hash table from one of its inputs and then probes into the hash table using each tuple of the other input. However, an RHJ has two major differences: (1) in the build phase, the RHJ operator might need to add the "missing" tuples into the reused hash table (for overlapping- and partial-reuse), and (2) in the probe phase, the RHJ operator might need to post-filter false-positives (for overlapping- and subsuming-reuse); i.e., tuples that are stored in a reused hash table but not required to execute the current join operator. Running the join without post-filtering would produce false-positives during the probing phase.

For each candidate hash table HT that can be reused to compute a given join, the optimizer in *HashStash* needs to estimate the total runtime costs. In the following, we explain the details of our cost model.

Cost Model: The main components that determine the cost of an RHJ are the resize cost c_{resize}, the build cost c_{build} and the probe cost c_{probe}.

$$c_{RHJ} = c_{resize}(HT) + c_{build}(HT) + c_{probe}(HT)$$

Our cost model for the RHJ explicitly considers the cost for resizing the hash table, c_{resize}. In order to minimize the cost of resizing in *HashStash*, we use a hash table that implements extendible hashing using linked lists for collision handling. Thus, instead of re-hashing all entries, only the bucket array needs to get resized and entries can be assigned to the new buckets lazily.

The costs for building and probing of an RHJ are different from a traditional hash-join and depend additionally on two parameters: (1) the contribution-ratio *contr* and (2) the overhead-ratio *overh* of a candidate hash table HT. The first parameter, the contribution-ratio *contr*, defines how much of the data in the candidate hash table HT already contributes to the operator if that operator reuses this hash table; i.e., this data does not need to be added to the hash table anymore and makes the build phase faster. For example, if $contr(HT) = 0.5$ then only 50% of missing tuples need to be added to the hash table HT during the build phase. The second parameter, the overhead-ratio *overh*, defines how much unnecessary data is stored in the hash table; i.e., this data contributes to the total memory footprint of the hash table and makes the building and probing phases slower since the hash table might spill out of the CPU caches. The overhead-ratio also determines the additional cost needed to post-filter false positives. For example, if $overh(HT) = 0.7$ then 70% of tuples in the hash table are not required by the RHJ. In the sequel, we discuss how to use both parameters (*contr* and *overh*) to model all of the four reuse-cases (exact, subsuming, partial, and overlapping).

In the following equations, we show how *HashStash* estimates the costs of the build phase $c_{build}(HT)$ and the probe phase $c_{probe}(HT)$ of an RHJ using these two parameters.

$$c_{build}(HT) = \underbrace{|Builder| \cdot (1 - contr(HT))}_{\text{\#tuples to insert}} \cdot \underbrace{c_i(htSize, tWidth)}_{\text{cost of a single insert}}$$

$$c_{probe}(HT) = \underbrace{|Prober|}_{\text{\#tuples to probe}} \cdot \underbrace{c_l(htSize, tWidth)}_{\text{cost of a single lookup}}$$

The build cost $c_{build}(HT)$ is determined by the number of missing tuples that need to be inserted times the cost of a single insertion c_i into the resized hash table. The probe cost $c_{probe}(HT)$ is determined by the number of tuples that need to probe into the hash table times the lookup cost c_l for a single probe into the hash table.

The cost of a single insert/lookup c_i and c_l depend on two parameters: (1) the memory footprint of the resized hash table $htSize$ (shown in the following equation) and (2) the width of a tuple $tWidth$ stored in the cached hash table HT. While the memory footprint $htSize$ determines if a hash table fits into the CPU caches or not and thus influences the insert/lookup costs, the tuple width $tWidth$ determines the number of I/O operations required to transfer a tuple between main memory and CPU caches. Since the hash table might contain more attributes than needed by the query (e.g., for post-filtering), the tuple-width $tWidth$ might actually be larger than a hash table's tuple-width that we would create individually for this query.

Moreover, for estimating the build and the probe cost, we need to be able to estimate the cost of a single insert/lookup (c_i and c_l). However, these costs depend on the specific hash table implementations and other hardware-dependent parameters; e.g., how prefetching into CPU caches is implemented. Therefore, these costs need to be determined by a set of micro-benchmarks which calibrate these parameters. The details of the micro-benchmark can be found in the Appendix A.2.

3.2.2 Reuse-Aware Hash-Aggregation

Similar to the reuse-aware hash-join (RHJ), the reuse-aware hash-aggregate (RHA) can reuse an existing cached hash table. Similar as for the RHJ, the RHA might also need to add "missing" tuples (for overlapping- and partial-reuse) and post-filter tuples (for overlapping- and subsuming-reuse). In the following, we discuss the cost model, that estimates the runtime cost of an RHA for a given hash table.

Cost Model: For a given candidate hash table, the optimizer estimates the total runtime costs of a reuse-aware hash aggregate as shown by the following equation:

$$c_{RHA} = c_{resize}(HT) + c_{insert}(HT) + c_{update}(HT)$$

The cost of an RHA consists of three components: (1) the resize cost c_{resize}, (2) the cost c_{insert} to insert the initial tuple for each missing group-by key, and (3) the update costs c_{update} of the aggregated value for the other input tuples. For example, assume an RHA has 100 missing input tuples with 10 missing group-by keys. In that case, the RHA needs to pay 10 times the insert cost and 90 times the update cost to reuse the given hash table. Similar to the RHJ, the contribution-ratio *contr* and the overhead-ratio *overh* have an influence on the insert/update costs.

In the following, we take a closer look into defining different cost components for a given candidate hash table HT. The cost component c_{resize} represents the cost to resize the hash table for the distinct missing group-by keys. Again, these costs are dependent on the implementation details of the hash table.

For RHA, we use the same hash table implementation as for the RHJ operator (i.e., we use the same cost estimates for resizing). The nature of the aggregation operator defines the way we estimate the main two cost components. There is an insert cost when the input tuple represents a group-by key that doesn't exist in the hash table. Whereas an update occurs when a tuple corresponds to a group-by key that was already inserted into the hash table before. The functions to estimate the insertion cost c_{insert} and the update cost c_{update} are shown in the following equations.

$$c_{insert}(HT) = \underbrace{|NewKeys| \cdot (1 - contr(HT))}_{\text{\#tuples to insert}} \cdot \underbrace{c_i(htSize, tWidth)}_{\text{cost of a single insert}}$$

$$c_{update}(HT) = \underbrace{(|Input| - |NewKeys)|) \cdot (1 - contr(HT))}_{\text{\#tuples to update}} \cdot \underbrace{c_u(htSize, tWidth)}_{\text{cost of a single update}}$$

The equations above need an estimate for the insert/update cost (c_i, c_u) for a input single tuple. These costs again must be calibrated by a set of micro-benchmarks for a given hash table implementation and the underlying hardware. The results can be found in the Appendix A.2.

3.3 Matching and Rewriting

The goal of the matching procedure `getCandHTs` in Algorithm 1 is to find the candidate hash tables that can be reused for a given sub-plan; i.e., instead of computing a sub-plan, we reuse a hash table that was created before to (partially) avoid the computation of the sub-plan. The matching procedure `getCandHTs` enumerates all the different candidate hash tables that are stored in the cache of *HashStash* and checks if one of them can be reused.

In the rest of this section, we use the following notation: C (the cached plan) represents the plan that produced a cached hash table in *HashStash* and R (the requesting plan) represents the input plan of the matching procedure `getCandHTs` rooted by the operator r. If r is a hash-join, R represents the sub-plan below the join that builds the hash table (including the join itself but excluding the probing branch). For a hash-aggregate r, the sub-plan R represents the operator tree below the aggregation operator including the aggregation itself.

For finding a matching hash table that can be reused for R, our hash table manager stores lineage information in a similar way as described in [22] using a so-called recycle graph G_{Rec}. The idea is that G_{Rec} merges the lineage of all cached hash tables in one graph and represents a union over all cached plans C. The graph G_{Rec} in *HashStash* thus consists of nodes that represent operators and edges that represent the data flow between operators. Moreover, for each node (i.e., operator) in G_{Rec}, we store a flag to represent if there exists a cached hash table or not; i.e., only joins and aggregations materialize a hash table and some hash tables might actually be evicted from the cache by our garbage collector (see Section 5). Figure 3 shows an example of a recycler graph that resulted from two cached plans; the first plan that produces $HT1$ contains an aggregation operator over the `Customer` table and the second query plan contains a join operator that builds the hash table $HT2$ over the `Customer` table on the join key `cid`.

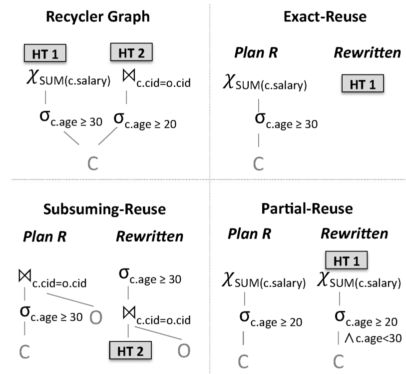


Figure 3: Match and Rewrite

In the following, we intuitively explain how the matching procedure `getCandHTs` works and which reuse cases are supported using the examples in Figure 3. In all reuse cases, the matching procedure checks if the requesting plan R and the cached plan C are equivalent (i.e., they do have the same join graphs) and the cached hash table provides all required attributes for R. The pseudo code and a more formal description of our matching procedure can be found in Appendix A.1. Moreover, for each of the reuse cases we discuss how to rewrite R to make use of a candidate hash table. To simplify the presentation in this section, we assume that the root operator r of R is a join operator and discuss the aggregation operator in the appendix as well.

Exact-reuse: In this case, R can be directly replaced by the cached hash table that was created by C. For example, Figure 3 (top right) shows the case where the requesting plan R matches the left path of the recycle graph G_{Rec}. The rewrite rule simply replaces R directly by $HT1$.

Subsuming-reuse: We also allow that R reuses a hash table of a cached plan C, if the cached hash table contains more data than required by R. This is the case if the cached hash table created by C contains a superset of the tuples required by R. When reusing this hash table, we have to post filter after probing to avoid false positives (i.e., tuples are returned that do not qualify for R). Figure 3 (bottom left) shows this case. Since the hash table $HT2$ contains customers for $age \geq 20$ and the requesting plan R requires only customer with $age \geq 30$, all false positives must be post filtered after probing using the filter predicate $\sigma_{age \geq 30}$.

Partial-reuse: We detect a partial-reuse case if a cached hash table does not contain all required tuples. Therefore, the rewrite must add the missing tuples to the hash table.

Figure 3 (bottom right) shows an example where $HT2$ can be partially reused; i.e., customers $20 \geq age < 30$ must be added to $HT2$ from the base table `Customer` before the hash table can be reused.

Overlapping-reuse: For this case, we test if the tuples selected by both plans R and C overlap. In this case, we apply both rewrites that we have discussed for the partial-reuse and the subsuming-reuse cases.

3.4 Benefit-oriented Optimizations

HashStash additionally implements the following benefit-oriented optimizations. The main intuition behind these optimizations is that one plan P is preferred over another plan P' if the plan P creates hash tables that promise higher benefits for future reuse.

Additional Attributes: For cached join hash tables, attributes used in selection operations in the sub-plan of the input which build the hash table might be added to the cache. This enables post-filtering of false positives without going back to the base tables. At the moment, we use a greedy heuristic that adds a selection attribute to the cached hash tables if the extended tuple still fits in the same number of cache lines. The reason is that based on our micro-benchmarks in Figure 12 in Appendix A.2, we can see that adding an additional value to a tuple does not have a negative impact on the insert, update, and probe cost.

Aggregate Rewrite: AVG is rewritten to SUM and COUNT to support the partial- and overlapping-reuse at the cost of initially creating a slightly bigger hash table. Here, we use the same heuristic as before to decide whether to apply this rewrite or not.

Join Order: Typically hash tables are always built over the smaller join input. However, if the hash table is reused in future it might be also beneficial to build the hash table over the bigger input. We therefore integrated a simple heuristic approach into our optimizer that is similar to the techniques presented in [22] to determine which intermediate result will provide more benefit for future queries based on the history of queries executed.

4. MULTI-QUERY REUSE

In this section, we describe the techniques in *HashStash* that enable shared plans to reuse cached hash tables. We call these plans *reuse-aware shared plans*. In the following, we first discuss the details of reuse-aware shared plans. Afterwards, we present how we extend our optimizer in *HashStash* to find an optimal reuse-aware shared plan for a given query-batch and a set of cached hash tables.

4.1 Reuse-Aware Shared Plans

The basic idea of shared plans is shown in Figure 5. Instead of compiling each query into a separate plan, multiple queries are compiled into one shared plan that reuses hash tables. The idea of shared plans has been presented in [11] already. In *HashStash*, we extend shared plans to allow them to reuse cached hash tables. In the following, we first reiterate over the idea of shared plans and then discuss the relevant modifications for our reuse-aware operators to work correctly in shared plans.

Different from a normal plan, in a shared plan individual operators execute the logic of multiple queries. The most common shared operator is the shared scan operator that evaluates the filter predicates of multiple queries in one scan. In order to keep track of which tuples qualify for which query, shared operators in [11] use a Data-Query Model where each tuple is tagged by the IDs of those queries it qualifies for. For example, if a tuple produced by a shared scan satisfies the predicates of query Q_1 and Q_3 but not of query Q_2, this tuple will be tagged using Q_1 and Q_3 (or 101 if a bitlist is used to represent query IDs). Moreover, other operators such as joins and aggregation operators can be shared as well. Figure 5 shows an example of a shared plan where the selection operators and the hash-join are shared by three queries (Q_1 to Q_3) while the aggregation is not shared (i.e., there exists one separate operator for each query). For the hash-join, we see that tuples tagged with query IDs (qids) are stored in its hash table. The query IDs are used during probing to produce the output of the join.

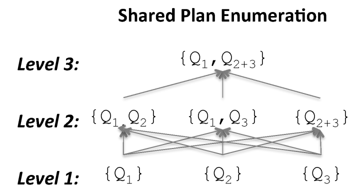


Figure 4: Dynamic Programming based Plan Merging

In the following, we describe our extensions for the reuse-aware hash-join and hash-aggregate such that they can execute multiple queries at a time.

Shared Reuse-Aware Hash-Joins: In general, the shared reuse-aware hash-join (SRHJ) operator works similarly to the non-shared reuse-aware hash-join (RHJ) presented in Section 3.2.1: Instead of recomputing the hash table in the build phase from scratch, a cached hash table is reused to avoid re-computation.

However, there are some important differences between an SRHJ that has to support query-batches and a non-shared RHJ that only supports a single query. First, the SRHJ can only reuse hash tables that include query IDs for tagging (as shown in Figure 5). A hash table that does not include query IDs can not be reused for a shared operator. Second, before the SRHJ operator starts to execute it has to re-tag all tuples stored in the reused hash table using the predicates of current query-batch. Otherwise, if it does not re-tag all tuples in the reused hash table, these tuples will be tagged with obsolete query IDs from a previous (non-active) query-batch, which might lead to wrong query results if query IDs are recycled. To that end, re-tagging represents an overhead that has to be considered in the cost model of an SRHJ.

Shared Reuse-Aware Hash-Aggregates: Shared aggregates are different from normal aggregation operators since they split the execution into two phases: a first phase that groups the input tuples by keys and a subsequent aggregation phase. While the grouping phase is shared for all queries, the subsequent aggregation phase is carried out for each query separately (i.e., the output of the grouping phase is split based on query IDs). In this paper, we focus on shared hash-aggregates that store the output of the grouping phase in a hash table before applying the aggregation functions on the individual tuples stored in the hash-table.

The goal of a shared reuse-aware hash-aggregate (SRHA) is to reuse hash tables to avoid the re-computation of the grouping phase. This is very different from reusing hash tables for a non-shared RHA operator since hash tables of an SRHA store individual tuples and not aggregates. Another

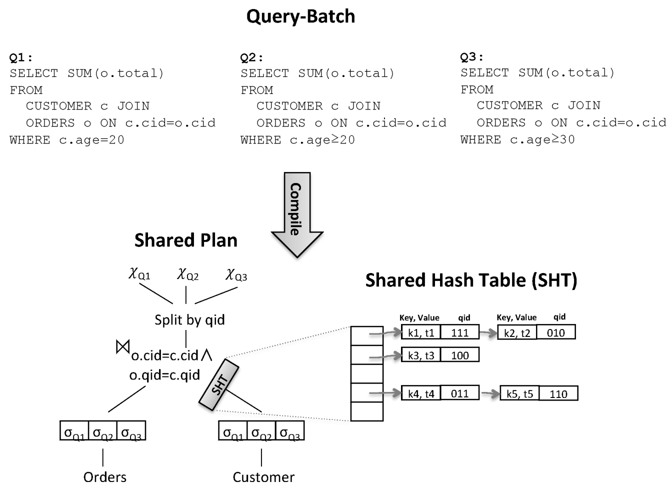


Figure 5: Shared Plans using a Data-Query Model

difference is that the SRHA operator also needs to re-tag all the tuples stored in the reused hash table (just as for the SRHJ operator) before the operator is executed. Both these aspects (i.e., storing individual tuples and the need for re-tagging) influence the cost of an SRHA and must be included in the cost model.

Finally, SRHA and the RHA operators also differ in how they select candidate hash tables from the cache. While an RHA must find hash tables with the same aggregation functions, an SRHA is more flexible since it can recompute any arbitrary aggregate function on the grouped data. For example, a hash table which was built for an SRHA operator that computes one aggregation function (e.g., SUM) can be reused by another SRHA operator, which computes a different aggregation function (e.g., MIN).

4.2 Plan Enumeration

In the following, we discuss the plan enumeration implemented in *HashStash* to support query-batches. The goal of the optimizer is to find a set of reuse-aware shared plans $\{S_1, S_2, \ldots, S_n\}$ for a given query-batch $\{Q_1, Q_2, \ldots, Q_m\}$ with $n \leq m$ that minimizes the total runtime to execute all queries in the given batch by reusing cached hash tables.

In order to find the optimal set of reuse-aware shared plans $\{S_1, S_2, \ldots, S_n\}$, *HashStash* uses a *dynamic programming approach* to merge query plans incrementally into reuse-aware shared plans. The pseudo-code for our procedure can be found in Appendix A.3. Figure 4 shows the application of the dynamic programming procedure to a batch of three queries (e.g., such as those in Figure 5). Each node in the dynamic programming graph in 4 represents a so-called merge configuration that describes which queries should be merged together into a shared reuse-aware plan and which should be executed using a separate non-shared reuse-aware plan. In terms of notation, $\{Q_1, Q_{2+3}\}$ represents a merge configuration, which defines that two separate plans should be generated: one non-shared reuse-aware plan for query Q_1 and one shared reuse-aware plan for queries Q_2 and Q_3.

In *HashStash*, queries may or may not be merged, depending on two aspects: First, two queries are merged if the total runtime of the shared plan is less than the sum of executing two individual plans. Second, not all queries are mergeable. In order to guarantee a correct plan execution, two queries Q_1 and Q_2 can only be merged if they have the same join graph. Otherwise, these queries cannot be merged and the plans must be kept separate. If two queries Q_1 and Q_2 are mergeable, the result of merging in *HashStash* is a shared reuse-aware plan where (1) all join operations are shared (i.e, SRHJ operators are used for joins) and (2) all aggregation operators that use the same group-by keys are shared (i.e, SRHA operators are used for aggregations).

In order to find the merge configuration that results in the minimal total runtime (i.e., the total sum over of plans), *HashStash* starts the dynamic programming process with merge configurations of size 1 (called level 1). On level 2, the optimizer then continues to find the merge configurations for all possible combinations of two queries which has the minimal total runtime by extending the merge configurations from the level below until the process reaches level m. For example, in order to compute the merge configuration on level 3 in Figure 4, the dynamic programming process merges query Q_3 into the merge configuration $\{Q_1, Q_2\}$ of level 2 amongst the other possible combinations (e.g., merging Q_2 into $\{Q_1, Q_3\}$ or merging Q_1 into $\{Q_{2+3}\}$. In order to merge Q_3 with the merge configuration $\{Q_1, Q_2\}$, the dynamic programming process enumerates all three possible merge configurations $\{Q_{1+3}, Q_2\}$, $\{Q_1, Q_{2+3}\}$, and $\{Q_1, Q_2, Q_3\}$ and keeps only the one with the minimal total runtime. Moreover, in order to avoid analyzing the same merge configuration twice, *HashStash* memoizes merge configurations and their estimated runtime.

Finally, to estimate the total runtime of a merge configuration, the optimizer computes the optimal reuse-aware (shared) plan associated with each entry of the given merge configuration. In order to find the best reuse-aware (shared) plan associated with entry in a merge configuration, the optimizer applies a variant of the enumeration process presented in Section 3.1 that supports query graphs and not only query trees. For example, given the merge configuration $\{Q_{1+3}, Q_2\}$ the optimizer applies the enumeration procedure to find the best (shared) plan separately for Q_{1+3} and Q_2. In order to find the best reuse-aware shared plans (e.g., for Q_{1+3}), the the plan enumeration in Section 3.1 uses reuse-aware shared operators (i.e., SRHJ and SRHA) instead of using non-shared reuse-aware operators.

5. GARBAGE COLLECTION

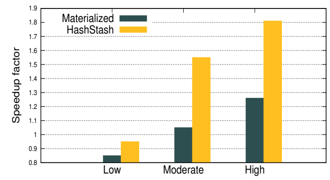
In this section, we provide the details of how garbage collection is implemented in *HashStash*. The main goal of garbage collection is to evict hash tables from the cache that are most likely not to be reused by other queries in future. In Section 2, we already described that the hash table manager monitors the hash table cache and will start an eviction process, whenever the total memory footprint of the cached hash tables exceeds a threshold (i.e., no more memory is available to store new hash tables). To decide on which hash tables to discard is the crucial part of the eviction process. Different from the eviction process used in database buffers, the garbage collection in *HashStash* does not work on the granularity of pages. Instead it can either work in a coarse-grained mode on the granularity of complete hash tables or in a more fine-grained mode on the granularity of individual hash table entries. While a coarse-grained mode needs less storage space for book keeping and requires less monitoring overhead than a fine-grained mode, it tends to keep “old” entries in a hash table even if other entries in the hash table are only used. Moreover, evicting individual entries from a hash table in a fine-grained mode requires a scan of individual buckets of the hash table. Finally, in a fine-grained mode, concurrent access of the eviction process and queries to the same hash table need to be synchronized. In *HashStash*, we have implemented this fine-grained mode. However, initial results showed that this mode results in a higher additional load that reduces the efficiency of *HashStash*. Therefore, we have decided to integrate only a *least recently used (LRU)* policy that evicts complete hash tables instead of evicting individual entries of hash tables (i.e., garbage collection is working in a coarse-grained mode) in *HashStash*. In order to implement the LRU policy, the Garbage Collector of *HashStash* keeps a *timestamp* of the last access for each hash table in its usage information. Based on this timestamp, the eviction process picks the hash table with the oldest timestamp and evicts it from the cache. The garbage collection process repeats the eviction until the memory footprint drops below the memory threshold. In our experiments we see that this policy is able to efficiently deal with different workloads where queries build on recent results. Moreover, the coarse-grained mode introduces only a minimal overhead for book keeping and for executing the eviction process.

6. EXPERIMENTAL EVALUATION

In this section, we report the results of our experimental evaluation of the techniques presented in this paper. The main goal of this evaluation is to: (1) compare the efficiency of reusing internal hash tables in *HashStash* to other existing reuse-strategies, (2) present the performance gains for both interfaces: the query-at-a-time and the query-batch interface, (3) show the efficiency and the accuracy of our optimizer and the cost models, (4) analyze the overhead of applying garbage collection in *HashStash*. In the following, we explain the details of the experimental setup that are used for all experiments.

Workload and Data: In order to analyze the efficiency of different re-use strategies we are using three different types of analytical workloads with (1) low-, (2) medium-, and (3) high-reuse potential. Each of the workload consists of 64 different queries over the TPC-H database schema. For the workload with the low-reuse potential, the average overlap of tuples read from base tables by two subsequent queries is 1%. This simulates the fact that users often look at different parts of a database. For the medium-reuse and high-reuse cases, the overlap is 10% and 50% respectively. The idea is that the spatial locality increases in workloads with higher reuse; i.e., in the high-reuse case users typically explore data in a common region using several queries before changing focus to other parts of the data. Details about all queries in the workload are given in Appendix A.4.

For the main dataset in all our experiments, we use a TPC-H database with a scale factor of $SF = 200$. Moreover, we created secondary indexes on all selection attributes used in our workloads. In order to enable parallel execution (see Implementation and Hardware), we split all tables in the TPC-H database into 20 partitions (i.e., one per core) using the reference-based partitioning scheme described in [33] to co-partition data. Co-partitioning tables in an analytical main-memory database is a common method to execute queries in parallel. This approach minimizes the number of re-partitioning operations during query processing and thus increases the degree of parallelism [24, 2]. We could have also used another classical partitioning scheme where we co-



(a) Speed-up of Runtime

Workload / Stats		**Mat.**	**HashStash**
Low	Mem Size	21.04 GB	23.2 GB
	Hit Ratio	0	0.01
	Time	287.04 (sec)	256.30 (sec)
Med	Mem Size	17.16 GB	12.86 GB
	Hit Ratio	1.27	1.45
	Time	291.84 (sec)	198.73 (sec)
High	Mem Size	16.34 GB	10.40 GB
	Hit Ratio	1.37	1.58
	Time	215.04 (sec)	149.41 (sec)

(b) Workload Statistics

Figure 6: Single-Query Reuse

partition only the two biggest tables on their join keys and share the other tables between the cores (similar to replication in the distributed setting). This partitioning scheme would favor reuse-based strategies (including our own execution strategy implemented in *HashStash*) over strategies that do not implement any form of reuse. The reason is that execution strategies without reuse would always need to scan the larger (shared) base tables instead of the smaller partitions.

We do not evaluate other scaling factors of the TPC-H database since the relative performance gains of *HashStash* compared to other executing strategies (e.g., materialization-based reuse) will be similar. For some experiments that contain micro benchmarks, we use synthetic data sets (e.g., to show the effects of our cost models). We describe these synthetic data sets further in the corresponding sections.

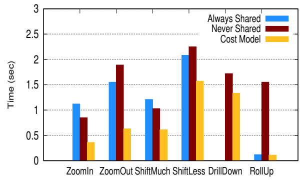
Implementation and Hardware: We implemented a system prototype using C++ and used GCC 4.9.2 as the compiler for a standalone system. For the execution of SQL queries, we generate C++ code and compile it into executable code. This execution model is used in modern main memory databases such as Hyper [17] and Tupleware [6] that both generate code in order to avoid unnecessary overhead during execution (i.e., iterator calls) and enable higher register and cache locality. For parallel execution, we use partition-based parallelism, which is a common execution model in modern parallel in-memory databases [17, 2]. In this model, individual threads first compute a sub-query on each partition and then the results are merged in order to compute the final result.

For running all experiments, we used one machine with an Intel Xeon E5-2660 v2 processor (20 cores) and 128GB RAM running Ubuntu Server 14.04.1. The cache sizes of the processor are: 32KB data + 32KB instruction L1 cache, 256KB L2 cache and 25MB L3 cache. Unless indicated otherwise, all experiments are executed using 20 threads (i.e., one thread pinned to each core).

6.1 Exp. 1: Single-Query Reuse

We now analyze the benefits of *HashStash* for the single-query interface. First, we vary the degree of reuse-potential using the workloads mentioned before and analyze the efficiency of different reuse strategies. We then study the effect of different degrees of parallelism on the reuse-efficiency.

Exp. 1a - Efficiency of Reuse: In order to show the efficiency of *HashStash*, we executed each workload on the partitioned $SF = 200$ TPC-H database using 20 threads with one thread per partition. We first executed the different workloads using the no-reuse strategy, which does not recycle any intermediates and has no cost for materialization. Afterwards, we executed the two reuse strategies: (1) materialization-based reuse where intermediate results are spilled out to a temporary table in memory, and (2) *HashStash*, which reuses internal hash tables.



(a) Runtime

Mutation	Update	Selected Reuse Scheme (O, P, C, S, Agg)
Zoom In	1996-06-01 1996-09-01	NSSSN
Zoom Out	1992-01-01 1998-01-01	NSSSS
Shift Much	1996-09-01 1998-01-01	NSSSN
Shift Less	1994-01-01 1998-01-01	NSSSS
Drill-down	add p_brand	SNSSN
Roll-up	remove p_mfgr	XXXXS

(b) User Interactions

Figure 7: Reuse on the Query-Level

The reuse strategy (1) is designed to simulate the reuse models of existing systems such as MonetDB [16] or Vectorwise [22]. In order to ensure a fair comparison of (1) and our reuse strategy (2), we cache the same intermediates in both strategies. For example, a hash table that is created by a join in (2) can be seen as a materialization of one of its inputs. Thus for (1), we introduce materialization operators in the plan that materializes the inputs for all join operations on which the hash table is built, as well as for the output of the aggregation operators. Furthermore, as another difference, (1) supports only exact and subsuming-reuse but not partial or overlapping-reuse as described in [16, 22]. In order to compare the two reuse strategies and exclude other effects, we implemented both approaches, (1) and (2), and the no-reuse strategy in *HashStash*.

The results of this experiment are shown in Figure 6. In this experiment, we turned the garbage collection (GC) off. The effects of GC are analyzed in Appendix A.5. Figure 6a shows the overall speed-up of both reuse strategies over the no-reuse strategy when running under different workloads. We see that our strategy in *HashStash* shows the highest speed-up for all workloads (low-, medium-, and high-reuse). For the workload with high reuse potential *HashStash* achieves a speed-up of 1.8× over the no-reuse strategy, while the materialization-based reuse strategy only achieves 1.25×. For the workload with low-reuse potential, which simulates a user randomly browsing the data, *HashStash* performs comparable to the no-reuse strategy; i.e., it does not introduce additional significant overhead even if there is (almost) no reuse potential. This is different from the materialized-reuse strategy, which incurs a penalty (i.e., a negative speed-up of 0.85×) caused by the additional materialization costs. Figure 6b shows additional statistics for each workload: memory footprint, hit ratio (i.e., number of times a cached hash table is reused), and the total runtime. For the materialized-reuse strategy, we report the memory footprint for all temporary tables as well as the hit ratio per temporary table (i.e., how often a temporary table was reused). For *HashStash*, we report the footprint for all cached intermediate hash tables tables as well as the hit ratio per hash table. The hit ratio is given as the average ratio of how often each element in the cache was re-used by a query. For the medium- and high-reuse case, we see that *HashStash* requires less memory in total while providing a higher hit ratio per cached element than the materialized-reuse strategy. The main reason for this is that the materialized-reuse strategy only supports two out of four reuse-cases supported by *HashStash*. To that end, less intermediates are reused and more new ones are added to the cache. For the low-reuse case, we see that the hit ratio of the cache is almost 0 for both strategies. In this case, the memory footprint is the highest since queries just register new elements to the cache without actually reusing them. Moreover, the memory footprint of *HashStash* is slightly higher than the materialized-reuse case. The reason is that hash tables have an additional overhead (e.g., pointers for linked lists of extends) when compared to a temporary table which is essentially an array in memory without any overhead. However, it is interesting to note that this does not have an effect on the runtime of *HashStash* since caching the internal hash tables does not cause any additional memory I/O compared to the no-reuse strategy. This is different from the materialized-reuse strategy, which requires additional I/O to persist the output of operators to the memory in order to support reuse.

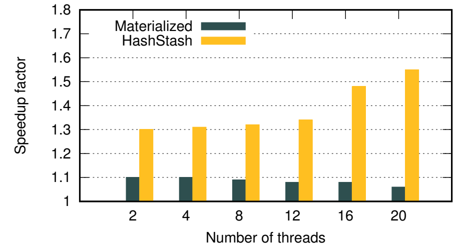


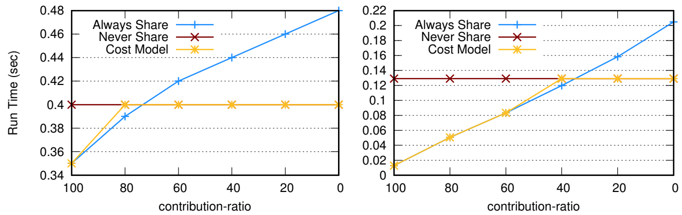
Figure 8: Effect of Multi-threading

Exp. 1b - Effect of Parallelism: In this experiment we executed the workload with medium-reuse potential using different numbers of threads (equally distributed to our two sockets) to see the effect of different degrees of parallelism and thus the effect of a varying load on the memory bus. As a dataset we again used the partitioned $SF = 200$ TPC-H database. We hypothesized that our reuse approach implemented in *HashStash* would reduce the load on the memory bus since no additional I/O for materializing intermediates is required. Figure 8 shows the results of comparing the speed-up of the two reuse-based approaches (our approach and the materialization-based reuse approach) over the no-reuse baseline. The results show that as the number of threads increase, our reuse-approach improves in terms of speed-up, which supports our claim. While our reuse approach creates much less load on the memory bus, the no-reuse and the materialized-reuse approaches both become memory-bound when using more than 12 threads. Therefore, the relative speed-up of our approach over the no-reuse based approach increases, while the speed-up of the materialization-based reuse approach decreases slightly.

6.2 Exp. 2: Efficiency of Query Optimizer

In this experiment, we show the benefits of our reuse-aware optimizer. We study the runtime of (a) reuse on the query-level as well as (b) reuse on the operator-level (i.e., for reuse-aware joins and aggregations). The main goal is to compare the performance of our cost-model based strategy with two baselines: the first baseline is *never-share*, where we turn reuse in our system completely off. The second baseline is *always-share*, where all operators use a greedy-heuristic to reuse the matching hash table in the cache with the highest reuse ratio. We include this strategy to show that greedily reusing hash tables can result in a performance that is worse than the performance of the *never-share* strategy and to emphasize the need for a cost model that decides whether to reuse a hash table or not.

Exp. 2a - Reuse on the Query-level: In this experiment, we selected a subset of seven queries from the workload with high-reuse potential. We selected these queries, since each query represents a different type of user interaction and thus provides different reuse potentials for join and aggregation operators. We selected the high-reuse case in order to show that the always-share baseline might re-



(a) Reuse-Aware Hash-Join (b) Reuse-Aware Hash-Agg.

Figure 9: Reuse on the Operator-Level

sult in non-optimal plans and showing that our cost-based approach finds better reuse-aware plans.

The first query of the sequence we picked is a 5-way SPJA query over the tables `Lineitem`, `Orders`, `Part`, `Customer`, and `Supplier`. The details of the six follow-up queries are summarized in Table 7b. The first column of this table lists the type of user interaction that was applied . The second column shows the difference of each query to its predecessor: The first four follow-up queries modify the selection predicate on the attribute `o_orderdate`. The last two queries modify the group-by keys.

For running this experiment, we executed all seven queries sequentially over the TPC-H database using our reuse strategy as well as using the two baselines (never-share and always-share). The first query populates the cache with five hash tables in total: four resulting from the joins and one from the aggregation. The results for the six follow-up queries (that are candidates for reuse) are shown in Figure 7a. In this figure, we see that the *Cost Model* strategy, which is based on our optimizer, outperforms the two other baselines since it always picks the optimal reuse strategy. In the best case (i.e., the *RollUp* follow-up query), the speed up factor is about two orders of magnitude better than *never-share*. The reason for this is because the cached aggregation hash table is sufficient to execute the *RollUp* query (i.e., no missing tuples need to be added and thus no joins need to be executed at all). For the *Drill Down* query, we could not execute the *Always Share* strategy since the *p_brand* attribute was never included in the corresponding hash table in previous executions and thus that hash table is not reusable.

The last column of Table 7b shows the detailed decisions of our optimizer (i.e., for the *Cost Model* strategy) for all operators of the six follow-up queries, which explain our performance results in Figure 7a. The string in this column uses one character to encode the decision for each operator (join and aggregation). The operators from left to right are shown in the header of the last column: For example, the O character represents the hash table created by the build phase of a join that scanned the `Orders` relation. The other characters represent the hash tables created by the build phase that scanned the `Part`, `Customer`, and `Supplier` tables. *Agg* represents the aggregation operator that is executed on top of all joins. The characters encode the following decision: N (*Not Shared*) states that a new hash table was created for the operator whereas S (*Shared*) states that the existing hash table was reused. Moreover, X defines that this operator was not need to be executed at all for the given query. For instance, this case occurs for the *Roll Up* operation, where the new query just needs to read the cached aggregation hash table.

Exp. 2b - Reuse on the Operator-level (RHJ): In this experiment, we show the efficiency of our optimizer for the reuse-aware hash-join (RHJ). For showing the efficiency, we directly execute the reuse-aware hash-join operators on two synthetic input tables. The table for the building phase was $16MB$ in size and the table for the probing phase was $10\times$ the size of the table for the build phase.

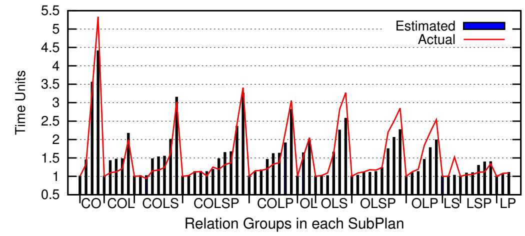


Figure 10: Accuracy of Cost Models

In order to show the efficiency of our optimizer for RHJ, after adding a candidate hash table of $16MB$ to the hash table cache, we executed multiple runs with different contribution ratios from 100% to 0%. 100% contribution-ratio means that the RHJ can reuse all tuples in the cached hash table and does not need to post-filter any tuples after probing; whereas 0% contribution-ratio means that the RHJ can not reuse any tuples in the cached hash table. Moreover, 0% contribution-ratio means there is 100% overhead in the reused hash table (i.e., all tuples must be post-filtered) due to the fact that for all contribution-ratios we keep the size of the cached hash table the same.

Similar to the previous experiment, we compare our *Cost Model* based strategy against the *Never Share* (i.e., a traditional hash-join) and the *Always Share* strategy which always picks the cached hash table for reuse. Figure 9a shows the results. We see that the *Never Share* strategy pays a constant price since it never reuses the hash table. Moreover, the costs for the Always Share strategy are constantly increasing since more and more missing tuples need to be added to the reused hash table (if the contribution-ratio decreases). At approx. 70% contribution-ratio, the Always Share gets more expensive than the *Never Share* strategy due to the overhead incurred in the cached hash table (i.e., tuples in the hash table that are not required by the RHJ). As an important result, we see that our Cost Model always picks the best strategy with the minimal cost: for a contribution-ratio from 100% to 70% it reuses the cached hash table and below 70% it decides to create a new hash table since the total runtime costs are cheaper when not reusing the candidate hash table in the cache.

Exp. 2c - Reuse on the Operator-level (RHA): In this experiment, we show the effect of reusing hash tables for reuse-aware hash-aggregates (RHAs). We again varied the contribution-ratio of the cached hash table as in the experiment before. Figure 9b shows that our cost model still picks the best strategy with the minimal cost.

6.3 Exp. 3: Accuracy of Query Optimizer

As described in Section 3.1, the plan enumeration algorithm is one of the core elements of *HashStash* that selects a reuse-aware plan with minimal runtime for a given set of cached hash tables. In this experiment, we validate the accuracy of the cost estimation component of our optimizer (i.e., the `cost` function used in Algorithm 1).

For this experiment, we execute the workload described in Section 6.1 with medium-reuse potential. In order to analyze the accuracy of our cost estimation, we select one of the 5-way join queries over the tables `Lineitem`, `Orders`, `Part`, `Customer`, and `Supplier` from this workload and analyze the estimated and actual cost of the optimizer. We selected this query since it is a complex query with multiple joins and the optimal reuse-aware plan contains both cases: operators that reuse a cached hash table and other operators that create a new hash table. In order to analyze the accuracy of our cost estimates, we compare the estimated and the actual cost for each enumerated sub-plan of this query. Figure 10 shows the results.

As a general observation, we can see that our cost models are accurate since the actual and estimated costs follow the same trend. To better understand the results, we clustered the costs into groups that represent equivalent sub-plans (i.e., one group represents sub-plans over the same partition of the join graph). For example, the group `CO` represents the enumerated join plans over the two tables `Customer` and `Orders` for all hash tables in the cache. Moreover, to better compare the actual and estimated costs, we are using normalized costs (called time units). For normalization, we divide the estimated costs of all plans in one group by the estimated cost of the plan with the lowest cost in that group. As a result, the plan with the lowest cost per group always has the cost of 1 time unit.

Figure 10 shows that, in the worst case, the deviation of the estimated from the actual cost of an enumerated plan is not higher than 25%. However, it is more important to determine if our optimizer finds the optimal plan. As discussed in Section 3.1, plan enumeration works incrementally and picks the cheapest sub-plan per group and composes the complete plan based on these optimal sub-plans. Thus, as long as the optimizer picks the cheapest plan per join group, it will make the correct decision. In order to see this, if the optimizer finds the cheapest plan per group, the normalized costs are sufficient (i.e., the absolute costs do not matter for this decision). Figure 10 orders the sub-plans per group by their actual costs. It is important to note that the first plan per group, which has the lowest actual cost resulting from the ordering, always has the lowest estimated cost as well. To that end, our optimizer is able the find the most optimal sub-plan per group for the query.

6.4 Exp. 4: Multi-Query Reuse

In this section, we present the evaluation results for the query-batch interface as explained in Section 4. In order to generate the batches of queries, we group the query trace of 64 queries of the workload with medium-reuse potential (i.e., with 10% overlap between the subsequent queries) of the experiment in Section 6.1 into smaller subsets of 4, 8, and 16 queries. Moreover, we expect that all 64 queries arrive at the same time and thus do not incur any additional overhead for batching. In order to populate the *HashStash* cache, we first executed one batch of the given size (e.g., 4 queries) and afterwards executed all 64 queries using different strategies (see below). In this section, we use a partitioned TPC-H database of $SF = 200$ and 20 threads.

In order to show the effect of reuse in shared plans, we executed the same sequence of batches using different modes: the first mode (*single-query plan, wo reuse*) is the baseline where all queries are executed sequentially and do not reuse any cached hash tables. The second mode (*single-query plan, w reuse*) executes all queries individually as well, but in a set up where reuse of intermediate hash tables using our cost-model is enabled. The third mode (*shared-query plan, wo reuse*) uses only pipeline-sharing without reusing any intermediates. In this case, we group all queries of a given batch into one shared plan. In the last mode (*shared plan, w reuse*), we use our reuse-aware shared plans using our optimizer as introduced in Section 4 to execute the batch.

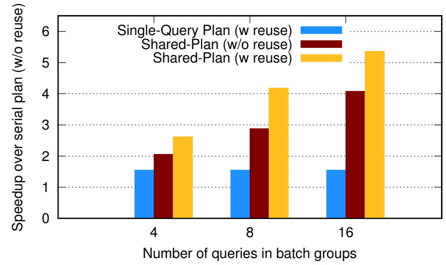


Figure 11: Batch Execution in *HashStash*

Figure 11 shows the speed-up of different re-use strategies over the baseline; i.e., the serial execution of all queries without any reuse. As expected, we see that *single-query plan, w reuse* has the same speed-up as in experiment 1a (see Section 6.1) and does not benefit from batching since it runs all queries individually. For *shared-query plan, wo reuse*, we create one shared plan for all queries in a batch. In this mode, we see the additional speed-up as an effect of batching. The speed-up of this approach is higher than that for the *single-query plan, w reuse* case. The reason is that the *single-query plan, w reuse* all queries are executed sequentially while *shared-query plan, wo reuse* only needs to run one shared query-plan. In this case, *shared-query plan, wo reuse* needs only one shared-scan whereas *single-query plan, w reuse* needs to execute multiple scans for the same table since the probing pipeline always needs to scan the base table; i.e., reuse of a hash table only avoids the build phase. Finally, for *shared-query plan, w reuse* the speed-up is the highest. The reason is that unlike *shared-query plan, wo reuse*, we create multiple shared reuse-aware plans for one batch using our optimizer and thus some of our shared plans do not need to go back to the base tables at all; e.g. if they can reuse a hash table that was produced by an aggregation operator. Moreover, *shared-query plan, wo reuse* creates only one shared plan that needs to join the tables for all queries in a batch. Thus, all queries in a batch are bound by the runtime of largest scan (even if they do not need this table), which is not the case for *shared-query plan, w reuse*.

7. RELATED WORK

Reuse of Intermediates: In order to better support user sessions in DBMSs, various techniques have been developed in the past to reuse intermediates [29, 16, 22]. All these techniques typically require that results of individual operators are materialized into temporary tables. This is very different from *HashStash*, which revisits "reuse" in the context of modern main memory DBMSs and seeks to leverage internal hash tables that are materialized by pipeline breakers and thus does not add any additional materialization cost to query execution.

In the following, we discuss further differences when comparing these techniques to the ideas of *HashStash*. [29] introduces an optimizer to select which intermediates should be reused. Different from *HashStash*, the cost models are rather coarse-grained and centered around the I/O benefits in disk-based DBMS. To that end, their cost models do not take the peculiarities of hash tables as well as hardware-dependent parameters such CPU caches into account. In [16], the authors integrate reuse techniques into MonetDB, a system that implements an operator-at-a-time execution model and relies on full materialization of all intermediate results. Therefore, MonetDB does not need to tackle the issues that result from additional materialization costs as in pipelined databases. [22] extends the work of [16] for pipelined databases and integrates the ideas into Vectorwise. In this paper, the authors introduce a cache with lineage which is similar to the ideas of the hash table manager in *HashStash*. A major difference is, however, that in both cases, intermediate results of operators are reused and not internal data structures of operators as we suggest in *HashStash*. Moreover, compared to all the approaches mentioned before [29, 16, 22], our work also supports reuse-cases for partial- and overlapping reuse and most importantly introduces a reuse-aware optimizer.

Another area where reuse of intermediates was analyzed is in the context of Hadoop. ReStore [18] is able to reuse the output of whole MapReduce jobs that are part of a workflow implemented in PigLatin. Moreover, it additionally materializes the output individual operators that are executed within a MapReduce job. Since ReStore is based on Hadoop and not tailored towards reuse in main memory systems, it makes their reuse techniques fundamentally different from those presented in *HashStash*.

Finally, buffer pools and query caches in database systems [5, 7] serve as a cache for frequently accessed data. However, the main purpose of buffer pools and query caches is to speed-up the access to base data (in case of the database buffer) or the final query result (in case of query caches) but not to reuse intermediates.

Cost Models for Hash Tables: We believe that the ideas presented in prior work [21, 20, 19] to model the cost of accessing hash tables are orthogonal to our cost model. The main contribution of our cost model is that it accurately models the reuse of hash tables, a question that was not covered by the other models. For example, in the case of a reuse-aware join operator, we account for the missing tuples that need to be added to build a hash-table. Moreover, during probing we model the cost of having additional tuples (that are not required by the query) in a hash table. To model the basic cost components for accessing a hash table (lookup, insert, update), we currently use ideas of micro-benchmark-based tuning as discussed in [19]. However, we could also leverage other cost models such as those discussed in [21, 20] to derive these costs.

Materialized Views: The reuse of results is a prime motivation of materialized views [12]. A key difference of our work is that we directly leverage internal data structures that are produced by query processing, as opposed to externalizing results as additional tables. As a consequence, when using a materialized view as input for a hash-join, the join operator still needs to build the hash table from the materialized view first, which is not needed in *HashStash*.

Moreover, our reuse-aware optimizer implements cost-models that target the reuse of internal data structures. As mentioned earlier, our cost model accounts for missing tuples, which is very different from the cost models for materialized views that are not "extended" during query execution. We also introduced benefit-oriented optimizations in *HashStash* (e.g., to store additional attributes in a hash table), which is another aspect not covered by traditional optimizers that rewrite queries for materialized views.

Automatic Physical Schema Design: Another line related to our work are techniques for online physical schema tuning [3]. The main goal of this work is to create additional database objects such as indexes or materialized views (discussed before) without involving a database administrator. Adaptive indexing [15, 28] also falls into this category and suggests creating indexes partially as a side effect of query processing. However, again these techniques do not consider internal data structures for reuse but externalize their decision by creating additional (partial) indexes, views, etc.

Multi-Query-Optimization: Another area of related work is Multi-Query-Optimization (MQO) [27]. The main idea of MQO is to identify common sub-expressions of a set of queries that are active in a DBMS at the same time. In order to save resources, common sub-expressions are only executed once. One problem of MQO is that in most workloads, common sub-expressions are a rather rare case. Therefore, MQO is typically used to optimize OLAP workloads over a star schema where the chance of common sub-expressions is higher since most queries join the dimension tables with the same fact table. All ideas in MQO are orthogonal to the reuse ideas presented in this paper; i.e., reuse of hash tables can be integrated into plans created by MQO techniques.

Work-Sharing: Work-sharing systems [25, 34, 31, 9, 4, 11] have similar goals as MQO since they also process multiple queries at a time by sharing work. However, different from MQO they do not require identifying the very same sub-expression to share work. One of the techniques for work-sharing is the shared (or cooperative) scan operator [25, 31, 34]. The idea of shared scans is that the scan operation can be shared by queries even if queries use different selection predicates. Other systems such as QPipe [9], CJoin [4], SharedDB [11] extend the idea of work-sharing to other operators such as joins and aggregations. All these ideas for work-sharing are again orthogonal to the reuse ideas presented in this paper. In this paper, we actually extended the ideas of [11] to integrate reuse into shared-plans.

8. CONCLUSIONS

The salient characteristics of modern main memory DBMSs and interactive analytical workloads require a critical rethinking of reuse in query processing. Our solution, called *HashStash*, focuses on the reuse of hash tables populated with intermediate query results. We avoid additional materialization costs by leveraging hash tables that are already materialized at pipeline breakers. We also do not incur the overhead of casting hash tables into relations and vice versa by treating hash tables as native units of reuse. Our reuse-aware optimizer can accurately model hash table usage and its impact on query performance, leading to highly profitable reuse choices that offer up to 5.3× performance improvement over the no-reuse baseline for realistic workloads.

9. ACKNOWLEDGMENTS

This research is supported in part by the Intel Science and Technology Center for Big Data, NSF IIS-1526639 and NSF IIS-1514491.

10. REFERENCES

[1] C. Binnig et al. SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. In *BTW*, 2013.

[2] C. Binnig et al. Sqlscript: Efficiently analyzing big enterprise data in SAP HANA. In *BTW*, 2013.

[3] N. Bruno et al. An online approach to physical design tuning. In *ICDE*, 2007.

[4] G. Candea et al. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *PVLDB*, 2009.

[5] C. Chen et al. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *EDBT*, 1994.

[6] A. Crotty et al. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*, 2015.

[7] S. Dar et al. Semantic Data Caching and Replacement. In *VLDB*, 1996.

[8] D. DeHaan et al. Optimal top-down join enumeration. In *ACM SIGMOD*, 2007.

[9] K. Gao et al. Simultaneous Pipelining in QPipe: Exploiting Work Sharing Opportunities Across Queries. In *ICDE*, 2006.

[10] G. Giannikis et al. SharedDB: Killing One Thousand Queries with One Stone. *PVLDB*, 2012.

[11] G. Giannikis et al. Shared Workload Optimization. *PVLDB*, 2014.

[12] J. Goldstein et al. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *ACM SIGMOD*, 2001.

[13] P. Hanrahan. Analytic database technologies for a new kind of user: the data enthusiast. In *Proc. of SIGMOD*, 2012.

[14] Harizopoulos et al. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, 2005.

[15] S. Idreos et al. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 2011.

[16] M. Ivanova et al. An architecture for recycling intermediates in a column-store. In *ACM SIGMOD*, 2009.

[17] A. Kemper et al. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[18] C. Lei et al. Redoop: Supporting Recurring Queries in Hadoop. In *EDBT*, 2014.

[19] V. Leis et al. How good are query optimizers, really? *PVLDB*, 2015.

[20] F. Liu et al. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In *SoCC*, 2015.

[21] S. Manegold et al. Generic database cost models for hierarchical memory systems. In *VLDB*, 2002.

[22] F. Nagel et al. Recycling in pipelined query evaluation. In *ICDE*, 2013.

[23] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *VLDB*, 2011.

[24] O. Polychroniou et al. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, 2014.

[25] L. Qiao et al. Main-memory scan sharing for multi-core CPUs. *PVLDB*, 2008.

[26] K. A. Ross. Cache-conscious query processing. In *Encyclopedia of Database Systems*. Springer US, 2009.

[27] T. K. Sellis. Multiple-query Optimization. *ACM TODS*, 1988.

[28] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 1989.

[29] K. Tan et al. Cache-on-Demand: Recycling with Certainty. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, Heidelberg, Germany*, 2001.

[30] TIBCO Spotfire. Retrieved on July 21, 2016. http://spotfire.tibco.com.

[31] P. Unterbrunner et al. Predictable Performance for Unpredictable Workloads. *PVLDB*, 2009.

[32] G. Wang et al. Multi-query Optimization in MapReduce Framework. In *VLDB*, 2013.

[33] E. Zamanian et al. Locality-aware partitioning in parallel database systems. In *SIGMOD*, 2015.

[34] M. Zukowski et al. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *PVLDB*, 2007.

A. APPENDIX

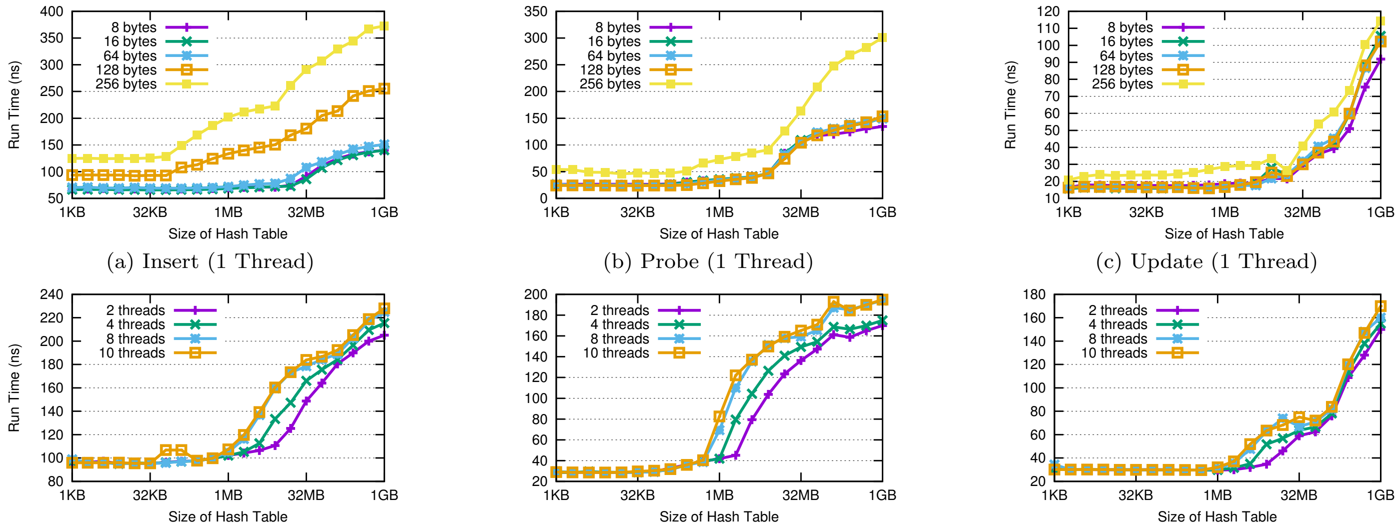
A.1 Matching and Rewriting Procedures

Algorithm 2: Matching Procedure in *HashStash*

```
Input  : Sub-Plan R
Output: Set of candidate hash tables candHTs
Algorithm getCandHT(Sub-Plan R)
candHTs ← ∅;
foreach candHT in G_Rec do
    C ← plan(candHT);
    p_C ← predicates(C);
    p_R ← predicates(R);
    if graph(C) ≡ graph(R) ∧ atts(R) ⊆ atts(C) then
        /* exact reuse                                  */
        if p_C = p_R then
            candHTs = candHTs ∪ candHT;
        /* subsuming reuse                              */
        else if p_R ⊂ p_C then
            candHTs = candHTs ∪ candHT;
        /* partial reuse                                */
        else if p_C ⊂ p_R then
            candHTs = candHTs ∪ candHT;
        /* overlapping reuse                            */
        else if p_C ∧ p_R ≠ ∅ then
            candHTs = candHTs ∪ candHT;
    end
end
return candHTs;
```

In the following, we explain how the matching procedure in *HashStash* works for each reuse case and then discuss which rewrites need to be applied.

Matching: In Algorithm 2, we show our matching procedure. The problem [is "problem" the word you want to use or perhaps rephrase] of matching is to find a subtree in our recycler graph G_{Rec} that matches the given sub-plan R and to check which of the four reuse-cases holds. Matching R to a cached sub-plan C in G_{Rec} builds on the notion of sub-graph isomorphism to test if R is contained in G_{Rec}. In our case, we simplify the problem and prune the search space since we only need to compare R to those sub-plans in G_{Rec} that actually rooted by an operator that refers to a cached hash table. In the following, we explain the algorithm of our matching procedure.



(a) Insert (1 Thread) (b) Probe (1 Thread) (c) Update (1 Thread)
(d) Insert (Multi-threaded) (e) Probe (Multi-threaded) (f) Update (Multi-threaded)

Figure 12: Reuse-aware Cost Parameters

The basic matching procedure shown here can only be used if the root r of R is a join (i.e., R is an SPJ query). We describe the extension of the procedure to also cover aggregation operators as root nodes at the end of this paragraph. The algorithm to find a match in the recycler graph G_{Rec} for a given sub-plan R first iterates over all candidate hash tables in the recycler graph (line 3-17). For each candidate hash table $candHT$, it then checks if the query graphs of the sub-plan C that creates the hash table is equivalent to R (i.e., if they join the same set of tables using the same join predicates) and if the attributes required by R are all in the cached hash table created by C (line 7). If this is true, the procedure then checks for the different reuse cases (line 8-15) as explained next.

- **Exact-reuse:** The matching procedure checks if the conjunction over all selection predicates used in R called p_R is equal to the conjunction over selection predicates used in C called p_C; i.e., if $p_C = p_R$ holds.
- **Subsuming-reuse:** We test if the conjunction of all selection predicates in R called p_R are subsumed by the conjunction of selection predicates of C called p_C; i.e., if $p_R \subset p_C$ holds.
- **Partial-reuse:** In order to support this reuse case, we test if the conjunction of all selection predicates in C called p_C are subsumed by the conjunction of selection predicates of R called p_R; i.e., $p_C \subset p_R$. This means, that the reused hash table does not contain all necessary tuples.
- **Overlapping-reuse:** If none of the before-mentioned cases holds, we test if the conjunction of all selection predicates in C called p_C overlaps with the conjunction of selection predicates of R called p_R; i.e., $p_C \wedge p_R \neq \emptyset$.

In case that the root r of R represents an aggregation operator, we additionally need to check that the group-by attributes $group_r$ of r are a subset or equal to the group-by attributes $group_c$ of the cached hash table created by a cached plan C and all aggregation functions are compatible. Moreover, in case of subsuming- and overlapping-reuse, all selection predicates in C and R that are on attributes other than the group-by attributes must be equal. The reason is that we are then only able to post-filter additional tuples stored in a cached hash tables based on their group-by attributes. In the partial-reuse case, we can still add the missing tuples if the aggregation functions permit.

Rewrites: Depending on the reuse case detected by the matching procedure above different rewrites are required. In the following, we first discuss the rewrites in case that the root r of R is a join operator and then discuss the extensions for aggregation operators at the end of this paragraph.

- **Exact-reuse:** The rewrite rule for this case replaces the sub- plan R directly by the candidate hash table HT.
- **Subsuming-reuse:** In this case R can be replaced by a selection operator $\sigma_{post}(candHT)$ over the candidate hash table $candHT$ where the predicate $post$ is p_R; i.e., the selection is applied after probing.
- **Partial-reuse:** R is rewritten to a plan R' which adds the missing tuples from the base tables to the reused hash table by using the plan $\sigma_{p_R \wedge \neg p_C}(R)$.
- **Overlapping-reuse:** In this case, we apply both rewrites that we have discussed for the partial-reuse and the subsuming-reuse case before.

Again, if the root node r of R is an aggregation operator we need to apply additional rewrites on top of the rewrites discussed before. In that case that the group-by attributes $group_r$ of r are equal to $group_c$ we can directly reuse the cached hash table. However, if $group_r$ is only a subset of $group_c$, we need to add a post aggregation operator on top of R' that resulted from applying the before-mentioned rewrites. The post-aggregation uses $group_r$ as group-by attributes and rewrites the AVG aggregation functions to use $COUNT$ and AVG from the cached hash table.

A.2 Calibrating the Cost Models

In the following, we present the result of a micro-benchmark to determine the insert, probe, and update cost of a single tuple in a given hash table that are all required input parameters for our cost model described in Section 3. The micro-benchmark is implemented in C++. The results in this section are obtained using GCC 4.9.2. Figure 12 show the results of our micro-benchmark for the hash table implementation used in *HashStash* on a machine with an Intel Xeon E5-2660 v2 processor using 128GB RAM running Ubuntu Server 14.04.1. The cache sizes of the processor are: 32KB data + 32KB instruction L1 cache (private), 256KB L2 cache (private) and 25MB L3 cache (shared).

Figure 12a and Figure 12b show the results for the insert and probe operations using a single thread. For both the probe and insert operations, we can clearly see the effects of different hash table sizes ($1KB$ to $1GB$) and cache boundaries on the insertion/lookup costs. The effect of the tuple-width ($8B$ to $256B$) is also visible but needs some more explanation. For insertion, the cost does not change as long as a tuple fits into one cache line, which is 64B in our processor. Once the tuple-width exceeds the cache line size, the cost increases as shown for 128B and 256B in Figure 12a. For lookup, the behavior is slightly different: due to the prefetching of one cache line by the CPU, the cost to lookup one tuple increases only when the tuple-width exceeds 128B. Figure 12c shows the results for the update costs c_u (threaded) for the same setup. The costs for the update follow the same trend as the insert costs; i.e., the cost to update one tuple increases only when the tuple-width exceeds 128B.

We also executed all these micro-benchmarks using multiple threads to calibrate the cost model for using different number of threads where every thread accesses its own private hash table. The results of the multi-threaded benchmarks when using a tuple size of 128 bytes are shown in Figure 12d to 12f. We see that an increased number of threads has only an impact once the hash table spills out of the private (non-shared) $L2$ caches.

A.3 Multi-Query Optimization Procedures

The goal of the procedure introduced in Section 4.2 is to find a set of reuse-aware shared plans $\{S_1, S_2, \ldots, S_n\}$ for a given query-batch $\{Q_1, Q_2, \ldots, Q_m\}$ with $n \leq m$ that minimizes the total runtime to execute all queries in the given batch. In order to find the optimal set of reuse-aware shared plans $\{S_1, S_2, \ldots, S_n\}$, *HashStash* uses a *dynamic programming approach* to merge query plans incrementally into reuse-aware shared plans. The idea is that procedure incrementally merges plans in the given input set of queries $Q = Q_1, Q_2, ..., Q_n$ (i.e., the query batch) and prunes merged plans for a given subset if a more optimal merged plan for the same set of queries is found. Algorithm 3 shows the pseudo-code for the dynamic programming (DP) procedure.

Algorithm 3: DP for Multi-Query Reuse

Input : Set of Queries $Q = Q_1, Q_2, ..., Q_m$
Output: Set of Shared-Plans $S = S_1, S_2, ..., S_n$

Algorithm `getBestSharedPlans`($Set\ Q$)
for $i = 1$ *to* m **do**
  $sharedPlans[Q_i] \leftarrow$ `getBestReusePlan`(Q_i);
end
for $i = 2$ *to* m **do**
  forall $Q' \subseteq Q$ *such that* $|Q'| = i$ **do**
    $sharedPlans[Q'] \leftarrow \emptyset$; **forall** $Q_j \in Q'$ **do**
      $sharedPlans[Q'] \leftarrow sharedPlans[Q'] \cup$ `mergePlans`($sharedPlans[Q_j], sharedPlans[Q' - Q_j]$);
      $sharedPlans[Q'] \leftarrow$ `prunePlans`($sharedPlans[Q']$);
    end
  end
end
return $sharedPlans(Q)$;

A.4 Workload in Experiments

As mentioned in Section 6, we generated workloads for our experiments with different reuse potentials (low, medium, high). The queries in all these workloads have the following characteristics: The initial query in each workload is TPC-H query $Q3$ that joins the tables `Lineitem`, `Orders`, and `Customer` with an aggregation operator on top. We used this query as it represents a medium-complex query with three joins and one aggregation operator on top. The query creates three hash tables in total for potential reuse. All other queries in our workloads resulted from applying different modifications to simulate different user interactions that are commonly used in analytical frontends such as Tableau [13] or Spotfire [30].

The user interactions simulated by different queries are: zooming-in/-out, shifting as well as drill-down and roll-up operations. While zooming-in/-out and shifting only change the selection predicate of the previous query, drill-down and roll-up group-by attributes respectively. Both operations can also add/remove other tables to/from the query. By using *Drill-Down* operations, we thus might add a new join operation (and thus a new table) into the query. The resulting queries are all SPJA queries of the following form.

```
SELECT <group-atts>, <agg_functions>
FROM <joins>
WHERE <predicates>
GROUP BY <group-atts>
```

The different reuse cases (low, medium, high) are achieved by using different predicates with the given overlap mentioned before. The most complex query in all workloads is a join over 5 tables (similar to TPC-H query Q5); the simplest query has only 1 table (similar to TPC-H query Q1).

A.5 Efficiency of Garbage Collection

In this experiment we show the effect of garbage collection on the performance of *HashStash*. We again used the workload with medium-reuse potential and executed the complete trace using two modes: The first mode (*wo GC*) represents the case where we execute all queries using the query-at-a-time interface with reuse, however, no garbage collector was active; i.e., *HashStash* used as much memory as needed to cache all hash tables. For the second mode (*with GC*), we additionally activated the garbage collector (GC). For the cache, we used 20% of the memory that would be required to store all hash tables. As a result, we measured the additional runtime overhead that was caused by the effects of the garbage collector (i.e., monitoring the size of all caches hash tables, evicting and reloading evicted hash tables). Compared to *HashStash* without GC, our experiment shows that *HashStash* with GC introduces approximately only a 10% higher overhead for the medium- and high-reuse case. For the high-reuse case this is negligible when looking at the performance gains of *HashStash* over a DBMS without any reuse (as we have shown in the experiments before). For the medium-reuse case, *HashStash* can still achieve a performance speed-up of 10% over the no-reuse case. Note, however, that when increasing the cache size to 50% of the total memory required to cache all hash tables, the overhead of garbage collection drops down to 5%. Most interestingly, for the low-reuse workload, GC causes almost no overhead since intermediate hash tables are anyway almost never reused.