DeepSea: Progressive Workload-Aware Partitioning of Materialized Views in Scalable Data Analytics

Jiang Du
University of Toronto
jdu@cs.toronto.edu

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

Wei Tan
IBM T. J. Watson Research Center
wtan@us.ibm.com

Renée J. Miller
University of Toronto
miller@cs.toronto.edu

ABSTRACT

Selective materialization of intermediate query results as views is an effective method for improving query performance. In this paper, we extend this technique to adaptively partition views based on the access patterns of a workload. That is, we collect information about the selection conditions of queries at runtime and utilize this information to determine fragment boundaries for the initial partitioning when materializing a view. Furthermore, we refine view partitions over time based on the selection conditions of incoming queries. We present a novel cost-benefit model for partitioned views, as well as a candidate view and fragment selection approach - both of which exploit the nature of partitioned views by taking the correlation among view fragments into account. Furthermore, we present DeepSea, an implementation of these techniques built on top of Hive. Our experimental evaluation demonstrates the effectiveness of partitioned views, improving performance by up to an order of magnitude compared to state-of-the-art approaches.

1. INTRODUCTION

The use of materialized views is a common technique to improve the performance of query workloads [21]. The questions of what to materialize, when to materialize, and when to use a view have been well studied. The same is true for other automated physical design techniques such as index and partition selection. Proper physical design for base tables, e.g., horizontal partitioning, often significantly improves the performance of queries [23]. In modern SQL systems built on-top of distributed dataflow engines (e.g., Hadoop [1]), issues of physical design, including partitioning of large files, are paramount to the performance of the system. Furthermore, intermediate results are often materialized for fault tolerance purposes and these results can be utilized as materialized views to answer future queries [12]. While each of these techniques has been studied intensively, we are the first to study the combination of materialized view selection and horizontal partitioning.

The major advantage of creating a partitioned view from an intermediate query result is that future queries with selection conditions over the partition attribute can be answered efficiently by accessing a subset of the view's fragments. However, partitioning a view increases the cost of view creation. Furthermore, new challenges arise because we have to decide when to partition a view, how to select fragment boundaries (within a partitioning), when to repartition, and what fragments to evict to save space. We address these challenges in this work.

Online View Selection. A view selection algorithm that is based on a query workload is called *adaptive*. Adaptive (or workload-aware) materialization and partitioning of views may be done at design-time or at runtime. That is, either a complete workload is given and the *view selection* algorithm determines which views to materialize and how to partition them offline, or the algorithm works in an *online* fashion making decisions based on the history of queries that have been processed so far. While online materialized view selection has been studied [24], we are the first to consider the online adaptation of partitioning choices for views. Our partitioning strategy is motivated by two important characteristics of real-life data analytic workloads: 1) data access is often not distributed uniformly over the domain of a selection attribute and 2) access patterns evolve as the interests of users change over time.

Non-Uniform Distribution of Access. Figure 1 shows the access distribution for a real analytic workload over the Sloan Digital Sky Survey dataset (SDSS) [2]. The figure shows the selection ranges on attribute *ra* of table *PhotoPrimary* for queries submitted to SDSS between March 8, 2010 and March 8, 2011. Note that there are ranges that are rarely queried and others that are very frequently queried. Clearly, *adaptive partitioning* can improve query performance. We use the range conditions of queries to adjust fragment (partition) boundaries with the effect that *hot spots* are covered by relatively small fragments and less frequently accessed data are covered by fewer and larger fragments. This has the advantage of focusing the effort of partitioning on the parts of the data which will give us the most benefit. Queries accessing hot spots can be answered using small fragments without touching unwanted ranges of the view. Furthermore, using this approach we avoid paying the cost of partitioning data that is accessed infrequently.

Evolving Access Patterns. In addition to being non-uniform, real workloads are not static, but rather access patterns may shift over time. Figure 2 shows how the selection ranges of SDSS queries over attribute *ra* of table *PhotoPrimary* evolve over the sequence of the first 10,000 queries containing such a selection, starting from March 8, 2010. The vertical line near query 1,000 means that one or more queries have selected the whole domain of attribute *ra*. The figure shows that the first 3,000 queries focus mainly on the range between 200 and 300 degrees. Later in the workload, a large number of queries focus on values around 100 degrees.

To accommodate evolving access patterns, we make decisions on how to partition a view online as queries arrive. We create a mate-

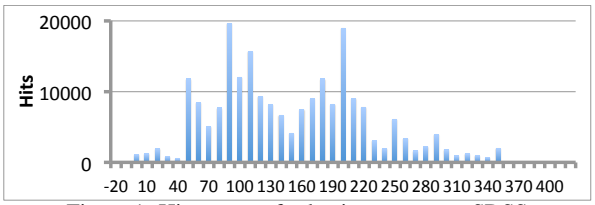


Figure 1: Histogram of selection ranges on SDSS

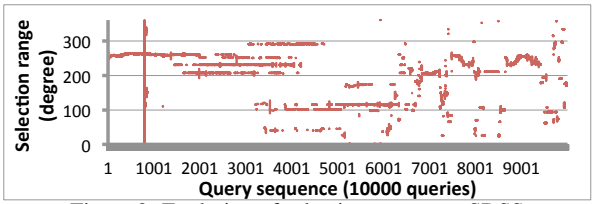


Figure 2: Evolution of selection ranges on SDSS

rialized view with an initial partitioning once we have determined that there is enough evidence that the creation of the view will benefit the current workload. We *progressively* refine fragment boundaries based on the selection conditions of incoming queries. Our progressive partitioning, coupled with a cost-based view and fragment eviction policy, allows us to adapt to evolving workloads. A view's competitiveness according to this cost model is based on its observed *benefits* (an estimate of the runtime that would have been saved if the view were to be materialized), its *creation cost* (the runtime overhead of materializing and partitioning the view), and its *storage size*. Importantly, we apply a decay function to timeout view benefits over time. This ensures that after a shift in the workload, views that are no longer useful for the current access pattern will eventually be replaced with views that fit the new pattern.

Partitioned Materialized View Pool Size. Typically, the storage space allocated for materialized views is not unlimited. We analyzed a BigBench workload [13] and found that if we materialize all intermediate join results as views, the total storage required is four times the size of the BigBench base tables. Of course, for evolving workloads, the number of materialized views and fragments would continue to increase and not all views will continue to provide a benefit to queries. Jain et al. [20] show the importance of a good *view selection strategy* for real-life applications: the savings that can be achieved with a small materialized view pool are similar to the savings that can be achieved with a large pool size as long as a good view selection strategy is applied. An important benefit of partitioned views is the finer granularity of control on view and partition selection: we can individually evict the fragments of a partitioned view that are unlikely to be used in the future.

Correlated Fragments. Given a finite amount of space for storing views, we present a novel strategy for selecting what fragments of a view to keep. Typically, decisions on whether to keep or evict a view are made independently for each view [15]. However, the benefits that different fragments of a partitioned view provide to a workload are not independent of each other. Returning to Figure 1, observe that ranges which are accessed often (ranges with many *hits*) tend to have neighbors with many hits (which are also accessed often). We find similar patterns for other attributes of different SDSS tables: parts of the domain of an attribute that are close to hot spots have a higher chance of being hit in the future than parts that are further away from hot spots. We present a new probabilistic model based on this correlation to determine when a fragment of view should be evicted. Our model treats a hit to a fragment as a sample from a probability distribution. We determine the normal distribution that has the maximum likelihood to have produced the sample and use this distribution for fragment selection.

Overlapping Fragments. Figure 2 also hints at a common pattern for selection ranges. A partition containing a few large fragments (for cold spots) and several small fragments (located at hot spots) may work well for some time, but as the workload evolves, there is a need to split a large fragment as additional queries begin to access it. This split incurs high write cost for repartitioning because if a fragment is split, its whole content needs to be read and written to disk. We present a solution that permits overlapping fragments. Rather than reading and writing the large fragment, we create a small fragment that overlaps the large fragment.

Contributions. Our main contributions are as follows.
- *Progressive, adaptive partitioning of materialized views.* We propose the first algorithm for progressively partitioning materialized views that adapts online to changes in a query workload.
- *Exploitation of fragment correlations.* Based on our study of real-life workloads, we present a novel cost-benefit model for view fragments and candidate selection that takes the correlation among fragments of a partition into account.
- *Overlapping fragments.* We allow overlapping fragments and show that they can reduce the cost of view creation especially over evolving workloads.
- *DeepSea.* We present DeepSea, an implementation of our techniques in Hive [27].
- *Evaluation.* We demonstrate DeepSea's effectiveness using a query workload modelled after a real workload from SDSS [2] and workloads from BigBench [13].

The remainder of the paper is organized as follows. We discuss related work in Section 2, introduce preliminaries in Section 3, and formally state the problem addressed in this work in Section 4. We give an overview of our approach in Section 5. We then present how to select view candidates in Section 6, how to select what to materialize and how to partition in Section 7, and how to answer queries using partitioned materialized views in Section 8. Afterward, we discuss the implementation of DeepSea in Section 9 and present our experimental evaluation in Section 10.

2. RELATED WORK

There are several lines of work related to our approach: answering queries using views; reusing intermediate query results; (online) self-tuning techniques for physical database design; database cracking; and semantic caching.

Answering Queries Using Views. Answering queries using materialized views has been studied intensively [3, 21]. Given a set of views and a query, computing the least expensive plan for the query using the views is computationally hard, because query containment checks are required to determine whether a query can be computed from a view. Query containment for bag semantics (SQL) is undecidable, even for restricted query classes (union of conjunctive queries). As a consequence, practical approaches for *logical matching* (i.e., determining whether a view can be used to answer a query independent of the query syntax) usually apply sufficient conditions for matching that are decidable or even in PTIME [14, 29]. Goldstein and Larson [14] present a lightweight algorithm that is integrated with a transformation-based optimizer and uses a cost model to determine the best rewriting. We have extended this approach to support matching fragments of partitioned views.

Reusing Intermediate Results. Although materialization has been studied extensively for relational databases [3, 16, 21], distributed

systems such as Hadoop have different characteristics that need to be explored and exploited. ReStore [12] materializes intermediate results of MapReduce jobs for reuse in future queries. Perez and Jermaine [24] exploit salient features of SQL-on-Hadoop systems including immutable data, abundant storage to accommodate materialized views, and excessive materialization of intermediate results that enables generating materialized views as a by-product of answering queries. The approach optimizes queries to produce intermediate results that, if materialized as views, would improve performance for past queries. If past queries are indicative of future queries then this would result in a speed up for future queries. We also gather knowledge about a workload to guide materialization, but in addition investigate partitioning for materialized views. The Nectar system [15] caches and reuses results of DryadLINQ/Dryad computations. ReStore and Nectar only perform physical matching, i.e., a view matches a sub-query if they are computed using the same expression. As explained previously, we decide to use logical matching which greatly improves the potential for reuse. Reuse of intermediate query results has also been studied for main memory DBMS such as MonetDB [19, 22]. This approach uses physical matching of operators except for selections where subsumption of range restrictions is considered, e.g., the result of a selection on $A < 5$ is a superset of the result of a selection on $A < 3$, and thus a query with selection $A < 3$ can be rewritten by using a materialized view whose selection is $A < 5$. Similar to ReStore and in contrast to automated materialized view and index selection approaches for relational databases, our approach significantly reduces view creation cost by considering intermediate query results as candidates for materialized view creation. In addition, whenever possible we use intermediate results that are materialized anyways by the MapReduce engine (e.g., at the end of a reduce phase).

Automated Physical Design. Automated tuning [10] is a rich field including: partitioning [23, 25], index selection [6, 26], and materialized view selection [4, 5, 8]. Adaptive index selection creates and drops indexes on-the-fly [6, 26]. Given a constraint on storage space, the idea is to monitor incoming queries and profile the performance gain for each index and then create the most promising ones. Adaptive materialized view selection [4, 5, 8] shares the same philosophy. Both index and materialized view selection use the DBMS optimizer's cost model to evaluate the benefits of an index or view without actually creating it. Bruno and Chaudhuri [9] have explored online index selection that is 3-competitive. However, this bound only holds for single index candidates. In contrast to these approaches we do not assume a sophisticated optimizer. Our solution also repartitions data on-the-fly, as a by-product of query answering. The H_2O system [7] supports multiple storage layouts, i.e., *columnar*, *row* and *group of columns*. At run-time, the system decides which layout to use for which part of the data, and continuously evolves the storage layout and data access strategy. In constrast to H_2O, we focus on horizontal partitioning of data in a distributed environment, and address the size requirement of materialized view pool.

Database cracking. Database cracking [18], i.e., adaptive and progressive indexing, incrementally builds an index structure over a table based on access patterns of queries. There is a rich body of work on enhancements of cracking such as the study of robustness and adaptiveness to dynamic workloads [17]. A similarity between cracking and our approach is that they both incrementally refine physical designs based on selection conditions in queries. In contrast to cracking, DeepSea focuses on horizontal partitioning of materialized views and makes cost-based decisions on whether to refine a partition.

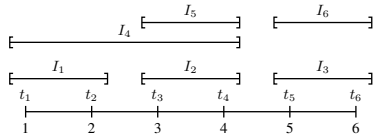
Semantic caching. Semantic caching [11] studies how to reuse subsets of input tables that are stored in a client-side cache. Each entry in the cache is described as a logical constraint (selection condition) providing a semantic description of the content of a cache entry. When a query is submitted to the client and can be answered (partially) using the cache, only a "remainder query" will be sent to the server to fetch the results that do not exist in the client's cache. Similar to DeepSea, intermediate results are reused and are reorganized based on access patterns. However, semantic caching only considers caching of the results of selections over base tables (we consider caching of a view that is partitioned on a selection attribute) and does not allow cached regions to overlap.

3. PRELIMINARIES

We now review the concept of horizontal partitioning. We use R, $S, \ldots$ to denote relations, $A, B, \ldots$ to denote attributes, and $\mathcal{D}(A)$ to denote the domain of attribute A. We call an attribute A *ordered* if there exists a total order $\leq_A$ over $\mathcal{D}(A)$. Only ordered attributes are considered as keys for horizontal partitioning.

Horizontal Partitioning. Horizontal partitioning splits the tuples of a relation into a set of disjoint fragments - each fragment holds the data for a range of values of the partition key (the attribute on which we partition). The union of these fragments equals the original relation.

DEFINITION 1 (HORIZONTAL PARTITIONING). *Let R be a relation and A an ordered attribute from R's schema. Consider a set $\mathcal{I} = \{I_1, \ldots, I_n\}$ of intervals where $I_i \subseteq \mathcal{D}(A)$. The* fragmentation $\mathbb{P}_{\mathcal{I}}(R.A)$ *of R on A according to $\mathcal{I}$ is the set of fragments $F_i \subseteq R$ defined as $F_i = \{t \mid t \in R \wedge t.A \in I_i\}$. If $\bigcup_{I \in \mathcal{I}} I = \mathcal{D}(A)$ and $\forall i, j : I_i \cap I_j = \emptyset$ then $\mathbb{P}_{\mathcal{I}}(R.A)$ is called a horizontal partition.*



EXAMPLE 1. *Assume a relation R has 6 tuples $\{t_1, t_2, t_3, t_4, t_5, t_6\}$ where the value of attribute A for tuple t_i is i. The domain $\mathcal{D}(A)$ of A is $\{1, \ldots, 6\}$. Consider a set $\mathcal{I}$ of three intervals $I_1 = [1, 2]$, $I_2 = [3, 4]$, and $I_3 = [5, 6]$ as shown above. A partitioning based on these intervals would result in fragments $F_1 = \{t_1, t_2\}$, $F_2 = \{t_3, t_4\}$, and $F_3 = \{t_5, t_6\}$. The fragmentation $\mathbb{P}_{\mathcal{I}}(R.A)$ is a horizontal partition of R according to A. Consider a second set of intervals $\mathcal{I}'$ containing $I_4 = [1, 4]$, $I_5 = [3, 4]$, and $I_6 = [5, 6]$. The fragmentation according to $\mathcal{I}'$ results in fragments $F_4 = \{t_1, t_2, t_3, t_4\}$, $F_5 = \{t_3, t_4\}$, and $F_6 = \{t_5, t_6\}$. This fragmentation $\mathbb{P}_{\mathcal{I}'}(R.A)$ is not a horizontal partition of R, because of the overlap between I_4 and I_5. Finally, $\mathcal{I}'' = \{I_4, I_6\}$ is again a horizontal partition of R.*

Overlapping Partitioning. It is sometimes beneficial to relax the disjointness requirement by allowing fragments to overlap. We call such a fragmentation an *overlapping partitioning*.

DEFINITION 2 (OVERLAPPING PARTITIONING). *Let R be a relation and A one of its attributes. We call a fragmentation $\mathbb{P}_{\mathcal{I}}(R.A)$ an overlapping partitioning iff $\bigcup_{I \in \mathcal{I}} I = \mathcal{D}(A)$.*

EXAMPLE 2. *Figure 3 illustrates why it may be beneficial to allow fragments to overlap. Assume that a query* Q1 *acccesses a range $[a, b]$ and that based on this access pattern we have decided*

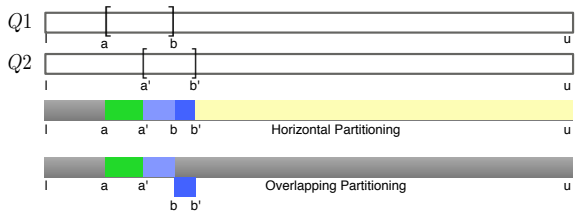


Figure 3: Overlapping fragments in progressive partitioning

to create a partition with three fragments $[l, a)$, $[a, b]$, *and* $(b, u]$. *A subsequent query* Q2 *accesses data in the range* $[a', b']$. *Note that* b *and* b' *are close to each other. Adaptive horizontal partitioning may create four new fragments based on* Q2 *by splitting the previously created fragments into* $[a, a')$, $[a', b]$, $(b, b']$ *and* $(b', u]$. *If we allow fragments to* overlap, *then we can avoid creating the fragment* $(b', u]$ *because no query has accessed data from this fragment yet. Instead, we create a fragment* $(b, b']$ *and keep the fragment* $(b, u]$ *that was created based on* Q1. *This avoids writing a large fragment that may not be accessed by future queries at the cost of additional storage for* $(b, b']$.

4. PROBLEM STATEMENT

We now state the problem addressed in this work: how to maintain a set of partitioned views (the *materialized view pool*) in an online fashion in order to maximize query performance.

Configuration. A configuration C models the current content of the materialized view pool. It consists of the set of views $\mathcal{V}$ that are currently in the pool and a mapping $\mathcal{P}$ that associates each view V and one of its attributes A with a set of intervals describing the current partitioning of the view on this particular attribute. Note that we permit multiple partitions of a view to be stored in the pool as long as these partitions are on different attributes. We define $\mathcal{P}(V, A) = \emptyset$ if view V has not been partitioned on attribute A yet.

DEFINITION 3. *A configuration* C *is a pair* $(\mathcal{V}, \mathcal{P})$ *where* $\mathcal{V}$ *is the set of views materialized in the pool and* $\mathcal{P}$ *is a mapping that associates with each view* $V \in \mathcal{V}$ *and an attribute* A *in the schema of* V *a set of intervals* $\mathcal{I}$ *over the domain* $\mathcal{D}(A)$ *of* A. *We use* $S(C)$ *to denote the total storage size of the views in configuration* C.

Problem Definition. In this work, we assume a query-only workload, i.e., no updates. We address the following problem: given a *workload* $\mathcal{Q} = Q_1, \ldots, Q_n$ of queries to be executed that is unveiled one query at a time and a *pool size limit* S_{max} (maximal storage to be used for views), choose a sequence of configurations $\mathcal{C} = C_1, \ldots, C_n$ in order to minimize the total execution time of the workload plus the time spent on view creation $\textsc{Cost}(\mathcal{Q}, \mathcal{C}) = \sum_{i=1}^{n} \textsc{Cost}(Q_i, C_i) + \sum_{i=1}^{n-1} \textsc{Cost}(C_i, C_{i+1})$. Here $\textsc{Cost}(Q, C)$ denotes the cost of executing query Q given the set of views C and $\textsc{Cost}(C_i, C_{i+1})$ denotes the cost of creating configuration C_{i+1} from configuration C_i. We require $C_1 = \emptyset$, i.e., no views have been created before the workload execution. We are interested in a restricted version of this problem where new views and refinements of partitions have to be based on the currently executed query Q_i, i.e., only views and fragments corresponding to intermediate results of this query ($\mathcal{V}_{cand}(Q_i)$ and $\mathcal{P}_{cand}$, defined in Section 6) are considered as candidates to be added to C_{i+1}. Given these preliminaries we can state the online partitioned view selection problem as follows.

DEFINITION 4 (ONLINE PARTITIONED VIEW SELECTION). *Given a pool size limit* S_{max} *and workload* $\mathcal{Q} = Q_1, \ldots, Q_n$ *that is unveiled one query at a time, incrementally determine the sequence of configurations* $\mathcal{C} = C_1, \ldots, C_n$ *that minimizes*

$$\textsc{Cost}(\mathcal{Q}, \mathcal{C}) = \sum_{i=1}^{n} \textsc{Cost}(Q_i, C_i) + \sum_{i=1}^{n-1} \textsc{Cost}(C_i, C_{i+1})$$

subject to

1. $C_1 = \emptyset$
2. $C_{i+1} - C_i \subseteq \mathcal{V}_{cand}(Q_i) \cup \mathcal{P}_{cand}$ *for all* $i \in \{1, \ldots, n-1\}$
3. $S(C_i) \leq S_{max}$ *for all* $i \in \{1, \ldots, n\}$

The online partitioned view selection problem is difficult for several reasons. First, this is an *online* problem: for each incoming query Q_i, we must decide which partitioned views or fragments to create and which to evict from the pool without knowing the remaining sequence of queries from the workload. There is abundant literature for online algorithms that provide worst-case guarantees. An online algorithm is said to be *k-competitive* if its result is at most of a factor k worse than the solution computed by an optimal offline algorithm (an algorithm which has access to the whole input). However, the competitiveness factor of such algorithms for search space sizes encountered in our problem are too high to be of any practical relevance. Even if we were to consider the offline version of the problem, we cannot hope for an optimal solution because of the undecidability of query answering with views.

Given these constraints we strive for a principled yet scalable solution that applies a carefully selected set of heuristics for each of the sub-problems of *determining view and partition candidates*, *view and partition selection* (determine the next configuration), and *view and partition matching* (determining whether a partitioned view can be used to answer a query). The main idea underlying our approach is that a solution should take hints provided by queries in the workload into account when deciding which intermediate query results to materialize and how to partition them.

5. SOLUTION OVERVIEW

Algorithm 1 gives a high-level view of the approach we use to process a query. The input to the algorithm is a query Q, view configuration C, and view statistics STAT. In the first step we determine which views and fragments (in the pool or not) can be used to answer the query (Section 8). The result of this step is a set $Rewr(Q)$ of possible rewritings of the input query which use the views. We then update the statistics kept for partitioned views to record that some views/fragments can be used to answer the query. Afterwards, among the rewritings that only use queries which are currently in the pool (C) we determine the rewriting Q_{best} with the lowest expected cost. Now that we have chosen a "plan" for the query (Section 6), we determine which of the intermediate results of the query are viable candidates to be stored as materialized views ($\mathcal{V}_{cand}$) and how to partition them ($\mathcal{P}_{cand}$). Note that even if a view $V \in \mathcal{V}_{cand}$ already exists and is partitioned, we may still produce fragment candidates for it (e.g., splitting an existing fragment to create a refined partition). Given such sets of candidates we add them to the set of partitioned views for which we want to keep statistics (using an initial rough estimate of their costs and benefits). The next step, described in more detail in Section 7, is to determine which of these candidates should be materialized during the execution of Q_{best} and, if necessary, which views to evict from the current configuration C to make space for these new views (recall that we limit the pool size by S_{max}). Once we have selected the views $\mathcal{V}_{sel}$ and fragments $\mathcal{P}_{sel}$ to create, we instrument the query Q_{best} to materialize intermediate results (and partition them if need

Algorithm 1 `ProcessQuery` (Q, C, STAT)

Input : Query Q, View Configuration C, View Statistics STAT
Output : Updated configuration C and statistics STAT

1: $Rewr(Q)$ = COMPUTEREWRITINGS(Q, C, STAT)
2: UPDATESTATS$(Rewr(Q), \text{STAT})$
3: Q_{best} = SELECTREWRITING$(Rewr(Q))$
4: $(\mathcal{V}_{cand}, \mathcal{P}_{cand})$ = COMPUTEVIEWCAND$(Q_{best}, C, \text{STAT})$
5: ADDCANDIDATES$(\mathcal{V}_{cand}, \mathcal{P}_{cand}, \text{STAT})$
6: $(\mathcal{V}_{sel}, \mathcal{P}_{sel})$ = VIEWSELECTION$(\mathcal{V}_{cand}, \mathcal{P}_{cand}, C)$
7: Q_{best}^{instr} = INSTRUMENTQUERY$(Q_{best}, \mathcal{V}_{sel}, \mathcal{P}_{sel})$
8: EXECUTEQUERY(Q_{best}^{instr})
9: UPDATESTATS$(\mathcal{V}_{cand}, \mathcal{P}_{cand}, \text{STAT})$

be). We then execute the instrumented query Q_{best}^{instr} and return its result to the user. Finally, we update the statistics for all candidates based on the information gained by executing Q_{best}, e.g., we now have precise measurements for the size of candidate views.

6. VIEW AND PARTITION CANDIDATES

We now discuss how our approach determines which views and fragments to create for a given query Q and configuration C. Our creation process operates in two steps: first we determine for which views and fragments we have gathered enough evidence to materialize them and then based on this subset of candidates we determine the next configuration based on the "value" of a view or a fragment using the statistics that we keep.

View and Fragment Statistics. For each view or fragment candidate, no matter whether materialized in the pool or not, we store statistics such as its size S, the estimated cost of creating it (COST), the set of timestamps when this view could have been used to answer a query (T), and a list of potential savings associated with each such timestamp (B). B and T together with a decay function that times out benefit as mentioned in Section 1, are used to compute the *benefit* of a view. For fragments we only record T and S since the benefit can be inferred based on its size and the saving of the view this fragment belongs to. Similarly, COST of a fragment is determined based on COST for its view.

DEFINITION 5. *The view statistics* STAT *is a triple* $(\mathcal{V}_{\text{STAT}}, \mathcal{P}_{\text{STAT}}, \Sigma)$ *where* $\mathcal{V}_{\text{STAT}}$ *is a set of views,* $\mathcal{P}_{\text{STAT}}$ *is a mapping as in* C *that associates each view and attribute in its schema with a set of fragment intervals, and* Σ *maps each view in* $\mathcal{V}_{\text{STAT}}$ *and fragment in* $\mathcal{P}_{\text{STAT}}$ *to a tuple* (S, COST, T, B) *respective* (S, T).

6.1 View Candidates

We first notice that certain relational operators are less likely to provide results that can be reused or the reuse of such an operator's result would not result in significant performance improvement. We consider the intermediate results of the following operators as candidates: join, aggregation, and projection. Joins are good candidates, because join computation is expensive and join results are likely to be reused. We consider aggregation operators, because the result size of an aggregation is typically small while its input size is large. Thus, we can save large computational cost by paying a small storage and creation cost. Likewise, projections can also reduce the size of their input considerably. We do not consider selections as view candidates, because materializing the input of the selection and partitioning it on the attribute used in the selection is usually more effective than using selections along.

DEFINITION 6 (VIEW CANDIDATES). *For a query* Q *and view configuration* C*, the set* $\mathcal{V}_{cand}(Q)$ *of view candidates for* Q *contains all subqueries* Q' *of* Q *that fulfill the following conditions:*

- Q' *is of the form* $\gamma(Q_1)$, $Q_1 \bowtie Q_2$, *or* $\pi(Q_1)$
- Q' *does not exist in* $\mathcal{V}$

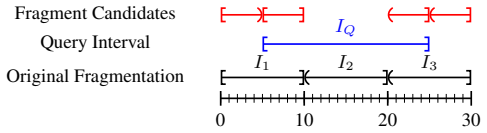
6.2 Partition Candidates

Similar to our view candidate generation approach, we want to use the characteristics of the current workload to guide the partition candidate generation. Note that we may maintain multiple partitions of the same view on different attributes. Given a current configuration of partitioned views C and statistics STAT kept for this configuration as well as for candidates, we consider new fragment candidates based on the selection conditions applied by a query. For every conjunction in the condition of a selection, i.e., a selection $\sigma_{l \leq A \leq u}(Q')$, which is a subquery of the current query Q, we consider new partition candidates for the view corresponding to Q', say V, based on the selection condition over attribute A. For the following discussions, without loss of generality, we assume $l \geq \underline{A}$ where $\underline{A}$ is the lowerbound of the domain of A, and $u \leq \bar{A}$ where $\bar{A}$ is the upperbound of the domain of A. It is trivial to replace l with $\underline{A}$ and similar for u when the above conditions do not hold. We have to distinguish several cases: 1) if we have not materialized Q' as a view V yet. In this case, we use l and u to split the potential fragments in $\mathcal{P}_{\text{STAT}}(V, A)$ which contain these points. If we have not yet gathered any intervals for this partition of V yet ($\mathcal{P}_{\text{STAT}}(V, A) = \emptyset$), then we initialize the partition with a single fragment: $\{\mathcal{D}(V, A)\}$ and then use l and u to split this fragment; 2) if a view V corresponding to Q' and a partition $\mathcal{P}(V, A)$ on attribute A already exists, then we again use the end points of the interval defined by the selection condition to consider splits of existing fragments that contain an end point as candidates. For each interval $I' = [l', u']$ of $\mathcal{P}(V, A)$ and the interval $I = [l, u]$, we create new candidates if either $l \in I'$ or $u \in I'$ using l respective u (or both) as split point(s).

DEFINITION 7 (PARTITION CANDIDATES). *Let* Q *be a query,* C *a view configuration, and* STAT *a view statistics. Consider a subquery* $\sigma_{l \leq A \leq u}(Q')$ *of* Q *where* Q' *corresponds to a view* V *in* $\mathcal{V}_{\text{STAT}}$ *and the intervals associated with partitioning* V *on attribute* A *(either* $\mathcal{P}(V, A)$ *if the view is in the pool or* $\mathcal{P}_{\text{STAT}}(V, A)$ *otherwise). We use* I *to denote* $[l, u]$*. For every interval* $I' = [l', u']$ *from* $\mathcal{P}(V, A)$ *respective* $\mathcal{P}_{\text{STAT}}(V, A)$ *we define the set of partition candidates* $\mathcal{P}_{cand}(V, A, Q')$ *according to* V*,* Q*, and* Q' *as the union of the sets of candidates for every such* I'*:*

1. *There is no overlap between these two intervals, i.e.,* $I' \cap I = \emptyset$*. In this case, no candidates are generated.*
2. *The query selection interval contains the partition interval, i.e.,* $I' \subseteq I$*. In this case, no candidates are generated.*
3. *The query selection interval overlaps the fragment interval from the left, i.e.,* $l < l' < u < u'$*. In this case, intervals* $[l', u]$ *and* $(u, u']$ *are considered as candidates.*
4. *The query selection interval overlaps the fragment interval from the right, i.e.,* $l' < l < u' < u$*. In this case, intervals* $[l', l)$ *and* $[l, u']$ *are considered as candidates.*
5. *The query selection interval is contained in the fragment interval, i.e.,* $I \subset I'$*. In this case, we consider three intervals as candidates:* $[l', l)$*,* $[l, u]$*, and* $(u, u']$.

EXAMPLE 3. *Consider a view* $V(A, B)$ *that is partitioned on attribute* A *using intervals* $I_1 = [0, 10]$*,* $I_2 = (10, 20]$ *and* $I_3 = (20, 30]$*. For an incoming query* $Q = \sigma_{5 \leq A \leq 25}(V)$ *we would consider the following candidates. Interval* $I = [5, 25]$ *overlaps with* I_1 *on the right (case 4). Thus, we create candidates* $[0, 5)$ *and* $[5 - 10]$*. No candidates are generated for* I_2 *(case 2). Finally,* I *overlaps with* I_3 *from the left (case 3) and we generate additional candidates* $(20, 25]$ *and* $(25, 30]$.



7. VIEWS AND PARTITION SELECTION

Our view and fragment selection method consists of two steps: 1) exclude candidates for which we have not gathered enough evidence of their effectiveness in improving the performance of the workload and 2) decide which candidates to materialize and which ones to evict to keep the pool size below the limit (S_{max}). The second step ranks views and fragments based on their *value* (Φ) as defined below. For each new fragment, we either create it by splitting existing fragments or create it as an overlapping fragment.

7.1 Cost and Benefit Model

We use a heuristic cost-benefit model to keep track of the "benefits" of view and fragment candidates. The benefits of a candidate are computed based on the potential savings in query execution time if this candidate would have been used to answer queries from the workload, the cost of creating it, its storage size, and other useful statistics for views and fragments. For candidates that have not been generated yet, we estimate their storage size and creation cost. We use this information to select which candidates to materialize and to decide which candidates to evict to make space for more competitive candidates.

View Statistics. For each view (candidate) V we keep the following statistics in $\mathcal{V}_{\text{STAT}}$: the **storage size** $S(V)$ occupied by the view, a set of **timestamps** $T(V)$ when the view was used to answer a query, and the **creation cost** of the view $\text{COST}(V)$ (which is initially estimated when we first see this view as a candidate). The creation cost is replaced with the actual cost once the first query containing the view as a subquery has been executed. The same applies to $S(V)$.

We compute the accumulated benefit $\mathcal{B}(V, t_{now})$ for a view at time t_{now} as follows. $\mathcal{B}(V, t_{now})$ is the cost we (could) have saved by using the view. The benefit is defined as

$$\mathcal{B}(V, t_{now}) = \sum_{Q \text{ used } V \text{ at } t} (\text{COST}(Q) - \text{COST}(Q/V)) \cdot \text{DEC}(t_{now}, t)$$

where $\text{COST}(Q)$ is the cost of query Q without using the view, $\text{COST}(Q/V)$ is the cost of running the query when using view V, and $\text{DEC}(t_{now}, t)$ is a monotonically decreasing function (in $t_{now} - t$) mapping the current time (t_{now}) and time when query Q was executed (t) to a value in $[0, 1]$. $\text{DEC}(t_{now}, t)$ is used to weight past cost savings by their age. This enables our approach to adapt to a changing workload. In our implementation we use the decay function as defined below which times out any benefit after a threshold t_{max} and otherwise counts it proportionally based on $\frac{t}{t_{now}}$.

$$\text{DEC}(t_{now}, t) = \begin{cases} 0 & \text{if } (t_{now} - t) > t_{max} \\ \frac{t}{t_{now}} & \text{otherwise} \end{cases}$$

View Value. Similar to Nectar [15], for each view V in the pool and candidate in $\mathcal{V}_{\text{STAT}}$ we compute its "value" at time t_{now} as a cost-benefit ratio $\Phi(V, t_{now})$. We use Φ during view selection to determine which views should be in the next configuration (views with a higher value are preferred over views of lower value). Using $\text{COST}(V)$, the accumulated benefit $\mathcal{B}(V, t_{now})$, and size $S(V)$, we define $\Phi(V, t_{now})$ as:

$$\Phi(V, t_{now}) = \frac{\text{COST}(V) \cdot \mathcal{B}(V, t_{now})}{S(V)}$$

The intuition behind the definition of $\Phi(V, t_{now})$ is that when a view is expensive to generate or the accumulated benefit of the view is large, its value is high and it is preferred over views with lower value. On the other hand, if the size of the materialized view is large, it is less competitive than other views of smaller size and similar benefits.

Fragment Statistics. Similar to view statistics we also keep separate statistics for every fragment in $\mathcal{P}(V, A)$ (a partition of view V on attribute A that is in the pool) as well as $\mathcal{P}_{\text{STAT}}(V, A)$ (a potential fragment candidate which is currently not materialized, but we have considered as a candidate before). For each such interval I (corresponding to a fragment F) we maintain the following information: the **storage size** of the fragment $S(I)$, its **creation cost** $\text{COST}(I)$, and a set of **timestamps** $T(I)$ when the fragment was hit (it was or could have been used to answer a query). These timestamps are used to compute the fragment value in a similar fashion as the view value explained above. We define the cost of creating the fragment to be the same as the cost of creating the partitioned view this fragment belongs to. This is because in order to recompute the fragment if it is not in the pool, we have to recompute the view's query and partition it.

Fragment Value. The value of a fragment is also modeled as a cost-benefit ratio in the same fashion as for views with the exception that benefits are computed as a ratio of the view creation cost and the relative size of the fragment compared to the total size of the view. The accumulated benefit for a fragment I is computed as

$$\mathcal{B}(I, t_{now}) = \sum_{Q \text{ used } I \text{ at } t} \left(\frac{S(I)}{S(V)} \cdot \text{COST}(V) \cdot \text{DEC}(t_{now}, t)\right)$$

and

$$\Phi(I, t_{now}) = \frac{\text{COST}(V) \cdot \mathcal{B}(I, t_{now})}{S(I)}$$

Probabilistic Fragment Benefit Model. The definition of the value of a fragment above ignores the fact that fragments in a partition of a view do not exist independent of each other, i.e., two fragments may be "neighbors" (e.g., $[0, 10]$ and $[11, 30]$) or may be quite dissimilar (e.g., $[0, 10]$ and $[1000, 1010]$). If we treat the hits on fragments we have observed so far in the workload as samples of a probability distribution, then when using these samples to determine the underlying distribution it would be natural to consider "distance" between fragments in the mechanism that determines the distribution. For instance, if we observe a large number of hits on a fragment $[0, 5]$ and no hits on fragments $[6, 10]$ as well as $[11, 15]$, then it is still reasonable to assume that fragment $[6, 10]$ which is close to a "hot spot" has a higher likelihood to be used in the future than fragment $[11, 15]$. Based on this observation, we present a mechanism for adjusting the number of hits per fragment. Define the number of hits $H(I)$ for a fragment I as $H(I) = \sum_{Q \text{ used } I \text{ at } t} \text{DEC}(t_{now}, t))$.We now define the adjusted number of hits $H_A(I)$ to compute a more realistic fragment value.

Consider a partition $\mathbb{P}_{\mathcal{I}}(V.A)$ for a view V. Note that we do not require that all intervals in $\mathcal{I}$ are currently in the pool. We keep statistics for each $I \in \mathcal{I}$ no matter whether materialized or not. Let H_{total} denote the total hits over all fragments of $\mathcal{I}$ adjusted by our decay function, i.e., $H_{total} = \sum_{I \in \mathcal{I}} H(I)$. H_{total} is the total number of queries that used at least one fragment from $\mathbb{P}_{\mathcal{I}}(V.A)$ weighted by $\text{DEC}(t_{now}, t)$.

Based on the analysis of the real-life workloads presented in Section 1, it is reasonable to assume that a normal distribution underlies accesses to values of an attribute's domain. Thus, given the observed hits for fragments we want to choose the mean μ and variance σ^2 of a normal distribution such that the resulting distribution best fits the observed hits. Here we apply well-known techniques from statistics for computing the maximum likelihood estimators (MLE) for the mean $\hat{\mu}$ and variance $\hat{\sigma}^2$ of normal distributions [28] to do the curve fitting.

We split the domain of attribute A into equi-size intervals p_1, ..., p_n which we call parts to distinguish them from fragments. We choose a quantification such that no part p_i is partially contained in an interval $I \in \mathcal{I}$. For instance, for a domain $[0, 20]$ if $\mathcal{I} = \{[0, 10], [11, 15], [16, 20]\}$ we may choose parts of size 5: $\{[0, 5], [6, 10], [11, 15], [16, 20]\}$. Based on the hits recorded for fragments $I \in \mathcal{I}$ we then determine the hits for each part p_i. For each fragment, we split the number of hits to this fragment evenly to the parts that are contained in the fragment. Let $\mathcal{I}' \subseteq \mathcal{I}$ be the intervals containing p_i and $\#I$ the number of parts contained in interval I. We define $H(p_i) = \sum_{I \in \mathcal{I}'} \frac{H(I)}{\#I}$, i.e., summing up the number of hits for each interval containing the part weighted based on the number of parts the interval contains. The likelihood function L for a standard distribution $N(\mu, \sigma)$ and set of observations $\{p_1, \ldots, p_n\}$ determines how likely it is that this particular distribution produced the given set of observations. It is defined as:

$$L(\mu, \sigma^2; p_1, p_2, ..., p_n) = (2\pi\sigma^2)^{-n/2} exp(-\frac{1}{2\sigma^2} \sum_{i=1}^{n} (p_i - \mu)^2)$$

By solving the log-likelihood function of the above function we have the maximum likelihood estimator mean and variance:

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} p_i \qquad \hat{\sigma}_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (p_i - \hat{\mu}_n)^2$$

The distribution $N(\hat{\mu}, \hat{\sigma})$ is the normal distribution which is most likely given the observations (it maximizes the likelihood function L). Note that we use the adjusted sample variance for the estimator $\hat{\sigma}_n^2$ because usually we do not expect a very large number of fragments for a view. This is a standard approach in statistics [28].

Note that since the MLE method is inexpensive we repeatedly adapt the estimation during the selection process for each incoming query. Based on the smoothed distribution of value accesses $N(\hat{\mu}, \hat{\sigma})$ we get from the maximum likelihood method and H_{total}, the total number of hits over all partitions, we compute the adjusted hits for a fragment $I = [l, u]$ as:

$$H_A(I) = H_{total} \cdot (P(x \leq u) - P(x \leq l))$$

Here $P(x \leq c)$ is an estimate (which ignores interval overlap) of how likely an access to a point in the interval $[-\infty, c]$ is computed over the normal distribution we have estimated using MLE. Note that this technique works for any probability distribution such as a Zipfian distribution or a mixture of distributions as long as it is feasible to compute the MLE of such a distribution given the observations. Here we choose the normal distribution, because it closely resembles the access patterns we have found in the real world workloads we have studied.

7.2 Filtering View and Partition Candidates

Our goal is to only save an intermediate result as a materialized view if this view is likely to be reused in the future and if the benefit of reuse $\mathcal{B}(V, t_{now})$ will offset the cost $\text{COST}(V)$ of materializing this view. Thus, the subset of candidates we consider for materialization is:

$$\mathcal{V}_{sel} = \{V \mid V \in \mathcal{V}_{cand} \wedge \text{COST}(V) \leq \mathcal{B}(V, t_{now})\}$$

We apply a similar filtering step for fragment candidates. This step is only applied for fragment candidates of existing partitions, i.e., when we decide whether to refine an existing partition based on selection, but not for candidates fragments for partitions which are not in the pool yet. Here we use the total benefits for a fragment computed based on its adjusted hits (using the estimated probability distribution of hits). Consider a candidate fragment I_{cand} for partition $\mathcal{P}(V, A)$ that is a candidate for the current query. The cost of creating I_{cand} depends on which fragments are currently in $\mathcal{P}(V, A)$. To materialize I_{cand} we have to read all fragments I such that $I \cap I_{cand} \neq \emptyset$, extract data that belongs to I_{cand} and then store I_{cand}. While we do not know upfront the actual size $S(I_{cand})$ for a fragment I_{cand}, we can estimate it based on the sizes of fragments currently in $\mathcal{P}(V, A)$ that overlap with I_{cand}. We assume that values are uniformly distributed within each fragment, and thus we can use the relative overlap between I_{cand} and an intervals in $\mathcal{P}(V, A)$ to estimate the size as:

$$S(I_{cand}) = \sum_{I \in \mathcal{P}(V,A): I \cap I_{cand} \neq \emptyset} \frac{\|I_{cand} \cap I\|}{\|I\|} \cdot S(I)$$

Based on this estimate of the size for a candidate fragment we have not materialized yet (otherwise we would know its size) we estimate the cost of creating the fragment as:

$$\text{COST}(I_{cand}) = w_{write} \cdot S(I_{cand}) + \sum_{I \in \mathcal{P}(V,A): I \cap I_{cand} \neq \emptyset} w_{read} \cdot S(I)$$

Here w_{read} (and w_{write}) denote implementation specific constants for reading (respectively, writing) data. In our implementation of Deepsea, w_{write} is typically much larger than w_{read} if we store a fragment in HDFS. Given the cost and estimated size, we only consider fragments for which the benefits are larger than the creation cost:

$$\mathcal{P}_{sel} = \{I \mid I \in \mathcal{P}_{cand} \wedge \text{COST}(I) \leq \mathcal{B}(I)\}$$

7.3 View and Fragment Selection

Given the prefiltered set of candidate views $\mathcal{V}_{sel}$, we now determine which of them to materialize (admit to the pool). In case this causes the total size of the views and fragments to exceed the limit S_{max}, we also have to decide which views or fragments to evict from the pool. Note that for selection we treat each fragment of a view independently. That is, the views in the pool do not partake in the selection process, only their fragments. However, candidate views and fragments are treated alike (candidate fragments are only created for partitioned views in the pool and view candidates are only created for views that do not currently exist in the pool). Thus, the set of views and fragments that are considered to be selected for the next configuration are:

$$\text{ALLCAND} = \mathcal{V}_{sel} \cup \mathcal{P}_{sel} \cup \bigcup_{V \in \mathcal{V}, A \in \text{SCHEMA}(V)} \mathcal{P}(V, A)$$

We rank the elements (views and fragments) in this set based on their value Φ (defined in Section 7.1). We then greedily add elements to the new configuration based on their rank.

Let $\text{ALLCAND}[i]$ be the ith element from ALLCAND according to Φ in decreasing order. We keep the first n elements from ALLCAND for the largest n such that $\sum_{i=0}^{n} S(\text{ALLCAND}[i]) \leq S_{max}$:

$$C_{i+1} = \{\text{ALLCAND}[i] \mid i \in \{0, \ldots, n\}\}$$

where

$$n = \underset{j \in \mathbb{N}}{\text{argmax}} (\sum_{i=0}^{j} S(\text{ALLCAND}[i]) \leq S_{max})$$

8. VIEW AND PARTITION MATCHING

The first important step when processing a query Q with our approach is to determine which views (whether in the pool or not) can be used to answer query Q. We call this process *matching*. The purpose of this step is to 1) update the statistics of views and fragments that could be used to answer query Q and 2) to determine the most efficient way of executing the query given the current configuration. The problem of finding all rewritings of a query Q given a set of views, i.e., queries that use the views and are equivalent to the input query, has often been called *query answering with views*. As mentioned earlier this problem in its full generality is undecidable for the class of queries we are interested in. We adopt a technique from Goldstein and Larson [14] that uses a sufficient condition to determine whether a view can be used to answer a query and indexes views such that this condition can be efficiently tested. We use a modified version of the index structure introduced in this work adapted for partitioned views to speed up matching.

8.1 A Sufficient Condition for Matching

The sufficient matching condition of Goldstein and Larson is checked over a representation of the query and the view (called *signature*) which is mostly independent of syntax, but can nonetheless be constructed from a concrete plan for the query. Signatures abstract away certain syntactic features such as join order. Our logical matching approach compares subqueries of a query with materialized views by computing the signatures for both the view and the subquery, and then checking the sufficient condition of Goldstein and Larson. Thus, we are able to also match parts of a query with a view. The signature of a query consists of the set of relations accessed by the query (*relation classes*), information on join and selection predicates (*attribute equivalence classes*, *selection predicate ranges*, and *remaining selection predicates*), projections, aggregation functions and group-by expressions. We refer the reader to Goldstein and Larson [14] for definitions of these abstractions.

8.2 Partition Matching

Once we have determined a rewriting using the views, the next step is to determine which partition of each view included in the rewriting to use and for each partition determine a subset of the fragments to be used. In order to match a fragment and a query, we must first find a match between the view represented by the fragment and the query. Note that a fragment of a view V corresponds to a view $\sigma_{l<A<u}(V)$ where A is the attribute on which V is partitioned on, and u and l are the boundaries of the fragment.

For every view V partitioned on A that is matched against a subquery Q' of the current query Q, we determine the restrictions Q' places on attribute A. This is done by using information about value ranges of selection conditions that are stored in the *Attribute Value Ranges* part of a query's signature (see [14] for a detailed explanation of the signature). Given our definitions of overlapping partitioning, the matching between a set of overlapped fragments and a query selection range is a set cover problem and thus is intractable. We use Algorithm 2 that greedily matches the fragments to a query selection range. Note that we use $\underline{I}$ to denote the lower and $\bar{I}$ to denote the upper bound of an interval I. We look for a set of fragments whose union covers the selection range. We maintain a variable $u_{covered}$ that stores the upper bound of the region covered so far. $u_{covered}$ is initialized to the lower bound of the selection range of the query u_θ. In each iteration of the loop, we greedily add the fragment that has the largest lower bound among the fragments that cover $u_{covered}$ from the left.

8.3 Indexing Partitioned Views

Algorithm 2 Partition Matching Algorithm

```
1: procedure PARTITIONMATCHING(θ, 𝓘)
2:     u_θ ← Upperbound of θ
3:     l_θ ← Lowerbound of θ
4:     F ← ∅
5:     u_covered ← l_θ
6:     while u_covered < u_θ do
7:         I_cand ← {I | I ∈ 𝓘 ∧ I ≤ u_covered ∧ Ī > u_covered}
8:         I_cur = argmax_{I∈I_cand} I
9:         u_covered ← Ī_cur
10:        F ← F ∪ {I_cur}
11:    end while
12:    return F
```

When computing matches between a query Q and a set of materialized views, it would be too slow to evaluate the sufficient matching condition over the signatures of all pairs of subqueries of Q and views in the pool. We adapt an in-memory index for view signatures called a filter tree [14] to be able to prune the search space early-on. A node in the tree is represented by a set of (key, pointer) pairs, where the key is a set of values, and the pointer points to a node on the next level. Each level represents one of the signature parts, e.g., the relations accessed by the view. The pointer of a leaf node points to a view. For each view, we store its partition information. For each partition of a view, we store the boundaries and statistics for each of its fragments. Note that we allow multiple partitions for the same view to exist as long as they are on different attributes. The search key for a query Q is its signature. We also use this index to keep the statistics for view and partition candidates (covered in Section 6).

8.4 Updating View and Partition Statistics

During view matching we update the statistics we keep for each view and its fragments, no matter whether the view or fragment is currently in the pool or not. For every rewriting $Q_{rewr} \in Rewr(Q)$ let V be a view that has been used in Q_{rewr} and for each such view $\mathcal{P}(Q_{rewr}, V, A)$ be the fragments of the partition of V on attribute A that are accessed by Q_{rewr}. We update the statistics for each such view and its fragments to reflect that it could be used to answer the query Q using the formulas presented in Section 7.1.

9. IMPLEMENTATION

DeepSea extends Hive [27], an SQL-on-Hadoop system [1]. While we have chosen Hive, because it is relatively mature, our techniques are applicable for any system that supports declarative querying ontop of shared-nothing dataflow systems.

Query Processing in DeepSea. Figure 4 shows how a query is processed by DeepSea. We use the parser and semantic analyzer of Hive to transform the input query into an abstract syntax tree (AST). The AST is translated into a directed acyclic graph (DAG) of operators (operator tree) and a task DAG (task tree) is generated from the operator DAG. Task DAGs assign operators to map and reduce phases. Our view matching module (see Section 8) rewrites the task DAG by replacing subqueries with references to materialized views or fragments. The rewritten DAG is then transformed into a DAG of MapReduce jobs to form a execution plan. We have implemented a *partition operator* that splits its input based on a list of fragment predicates which determine which input tuple belongs to which fragment. The output for each fragment is routed to a file sink operator that writes the fragment's content to a file.

Simulator. Testing view and fragment selection strategies requires extensive experiments over a large number of diverse workloads.

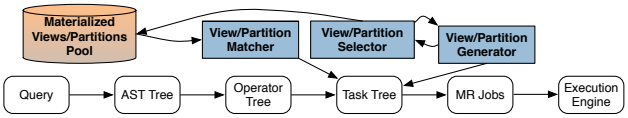


Figure 4: Query processing in DeepSea

Description	Values (default in bold)
Instance size	100GB, **500GB**
Pool size	50GB, 125GB, 250GB, 500GB, ∞
Query selectivity	**1% (Small)**, 5% (Medium), 25% (Big)
Query skew	Uniform (U), Light (L), **Heavy (H)**

Table 1: Parameters and their values

Since the benefits of partitioned views are more pronounced for large datasets, it is necessary to consider such datasets which results in large query runtimes. To be able to quickly test variations of a workload with different selection conditions ranges we have developed a simulator to study the efficiency of our selection algorithm and compare it to alternative approaches. We run a series of query templates with different selection patterns (introduced in Section 10) and gather statistics such as the storage size of views and fragments as well as the elapsed time. The simulator keeps track of the query template and the selection pattern that is running. It builds the necessary views and partitions based on the selection strategies and the size limit of the materialized view pool. Once sufficient statistics have been gathered for a query template, we estimate the runtime of future executions of a query template using linear regression.

Bounding Fragment Size. There are situations where bounding the size of a fragment (from above or below) may be beneficial. If the access patterns of queries are limited to a small subrange of the domain of an attribute, then our approach may create very large fragments for the parts of the domain that are accessed infrequently. In general it would be beneficial to split such large fragments, because the potential benefit of large fragments is small while the overhead of creating a few medium sized fragments instead is not very high. We approach this problem by limiting the maximal size of the fragments we create relative to the size of a view. We define a threshold ϕ for the relative size of a fragment. When we materialize and partition a view, we split every fragment that is larger than $\phi \times S(V)$ into smaller, equi-sized fragments. Big data systems are usually built on top of distributed file systems that favors large block sizes. For instance, HDFS has a default block size of 128 MB (or 64 MB depending on the version). We use the file system's block size as the lower bound for fragment size.

10. EVALUATION

We evaluate our system using the big data benchmark suite BigBench [13]. We demonstrate the overall performance of DeepSea using queries and data distributions that are modeled based on the SDSS workload [2]. This ensures that our evaluation considers important characteristics of real workloads. We also use BigBench to generate a set of synthetic workloads that are tailored to evaluate our major contributions: adaptive and progressive partitioning, exploitation of fragment correlations, and overlapping partitioning.

We generate instances of size 100GB and 500GB, both with uniform distribution, for the synthetic workloads. Table 1 shows parameters that we vary in the experiments as independent variables. The default value for each variable is shown in bold. We use the default value for the experiments unless otherwise mentioned. We consider three different query selectivities: *Small* (S) means that the selection condition returns 1% of the data; *Medium* (M) means that the selection condition returns 5%; and *Big* (B) means 25%. We use three different distributions for selection conditions of queries: uniform distributed (U), lightly skewed (L), and heavily skewed (H). *Uniform* means that for a fixed interval size, we pick a set of intervals such that the mid-point of the intervals is uniformly distributed. *Lightly skewed* means the mid-point of the selection intervals follows a normal distribution over the domain with a variance set to 7.5% of the domain. *Heavily skewed* also uses a normal distribution, but with the variance set to 0.25% of the domain.

Our evaluation is conducted on a cluster of 32 nodes. One node is a dedicated master node with 8 threads and 48GB memory. Each of the remaining 31 slave nodes has 6 threads, 12GB memory, and a 400GB disk. All results are based on the average of at least three runs, unless mentioned explicitly.

10.1 Workload for a Real-Life Application

We demonstrate two key properties of our system on a real-life application: 1) we compare the performance of DeepSea when there is no size limit for the materialization pool to Hive that does not uses materialization and NP, a materialization strategy that stores each view without partitioning them; 2) we compare the performance of DeepSea when there is a size limit for the pool to state-of-the-art view selection strategies such as the one of Nectar [15].

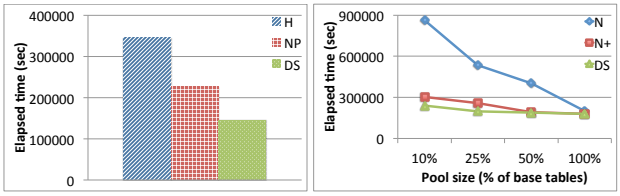
We create a histogram over the values of attribute `ra` for the table `PhotoPrimary` of SDSS. We then generate a BigBench dataset, and for all tables that contain attribute `item_sk` use the histogram that we obtained from SDSS attribute `ra` to sample values for `item_sk`. Furthermore, we generate a query workload: we pick ten query templates (Q1, Q5, Q7, Q9, Q12, Q16, Q20, Q26, Q29, Q30) from BigBench that contain joins, and we add a selection on attribute `item_sk` to these templates. We randomly pick 1000 selection ranges from the SDSS workload (selections on attribute `ra` of the table `PhotoPrimary`, kept in order of the query submission time). Next, we randomly picked a BigBench query template and mapped the selections of SDSS to selections on `item_sk` of the BigBench queries. Thus, we obtain a workload of 1000 BigBench queries simulating SDSS access patterns over an SDSS data distribution to evaluate the overall performance of DeepSea.

In this experiment, we compare DeepSea with two baselines. The first is the unmodified Hive system (*H* in the graphs). The second is a materialization strategy that does not use partitioning. We call this strategy non-partition (or *NP* in the graphs). This is akin to using a materialization strategy like ReStore [12]. However, in constrast to ReStore which only uses physical matching, *NP* applies our logical matching technique. Figure 5a shows performance results for the 500GB dataset without a pool size limit. Our approach requires only 64.2% of the time of non-partition materialization to execute the whole workload. Materialization without partitioning results in roughly 65.6% of the time of Vanilla Hive.

To evaluate the effectiveness of DeepSea's selection strategy, we compare it with the view selection strategy of Nectar [15]. Nectar does not consider accumulated benefit as a factor. To understand the performance gain due to the use of accumulated benefit in contrast with the other innovations in DeepSea, we extended Nectar's cost-benefit model to include the accumulated benefit of a view or fragment. The modified cost-benefit measure N^+ for views which we call *Nectar+* is: $N^+(V) = \frac{\text{Cost}(V) \times \mathcal{N}(V)}{S(V) \times \Delta T}$ where ΔT is the time elapsed since the last access to V and

$$\mathcal{N}(V) = \sum_{Q \text{ used } V \text{ at } t} (\text{Cost}(Q) - \text{Cost}(Q/V))$$

For fragments, we adapt our formula from Section 7.1 in a similar fashion by removing the application of the decay function. Fig-



(a) DS vs. NP vs. H (b) Selection strategies

Figure 5: Workload simulating SDSS (1000 queries), 500GB

ure 5b shows results for Nectar (N in the graph), Nectar+ (N+ in the graph), and DeepSea (DS in the graph) for different pool size limits. We observe that Nectar+ consistently outperforms Nectar, and DeepSea consistently outperforms Nectar+. When the size limit of the materialized view pool is relatively large (500GB, which is the total size of all base tables), the difference between Nectar, Nectar+ and DeepSea is marginal. When the size limit is shrunk to 10% of the total size of all base tables, DeepSea shows its strength requiring only $\sim$ 28% of the time of Nectar (20% faster than Nectar+) and being 30% faster than Vanilla Hive. DeepSea keeps fragments that can improve the overall performance in the pool, because they are neighbors of more frequently accessed fragments. Nectar and Nectar+, however, evict these fragments because of their low hit count. When the pool limit is decreased to 5% of the total database size, all three techniques perform poorly (worse than original Hive with no materialization (Figure 5a)). This is because with such a small materialized view pool, all three strategies evict fragments that are accessed earlier and admit fragments that are accessed more recently. Since evicted fragments may be accessed soon after eviction, there is an "oscillation" in the pool with extra working being done for the materialization and little or no gain seen from this extra work.

10.2 Adaptive and Progressive Partitioning

To understand the benefits of partitioning strategy, we compare DeepSea with equi-depth partitioning (E in the graphs or E followed by a number indicating the number of fragments). Equi-depth is a simple, non-adaptive and non-progressive alternative to DeepSea's partitioning approach. To evaluate the benefit of progressive partitioning standalone, we tease out the benefits of using DeepSea which is workload aware.

For this experiment, we do not bound the size of the largest fragment. We use instances of query template Q30 and vary the selection condition of this query to produce workload sequences where `Q30_i` denotes the ith query in a sequence.

First we generate a sequence of queries that has small selectivity and is heavily skewed as defined at the beginning of this section. Figure 6 shows the cost of partitioned view creation and the cost for queries that reuse fragments. Figure 6a shows that when the number of generated fragments increases, the cost for creating and partitioning the view increases as well. In Figure 6b, we notice that if the same number of fragments are generated by both approaches (6 fragments in this experiment), equi-depth performs worse than DeepSea because of the larger size of fragments that must be read during query evaluation. Increasing the number of generated fragments for equi-depth reduces the average runtime for the following queries. However, when we set the number of fragments to be relatively large (60 fragments), performance decreases. Small fragment size affects performance negatively, because a large number of files has to be read and data is unevenly distributed among tasks. Figure 6c shows the cumulative time for the query sequence.

In addition to better performance, DeepSea also differs from equi-depth partitioning in terms of the execution of MapReduce jobs on the cluster. We analyze cluster utilization for the queries that reuse the generated fragments by running the default query sequence on the default dataset. Besides noticing the time needed in DeepSea is about 20% less than equi-depth, the number of map tasks issued to the Hadoop engine is about 40% to 50% more for equi-depth. The reason is that the fragments used by equi-depth to answer the query are larger than the ones used by DeepSea. Thus, the Hadoop engine issues more map tasks to parallelize the read as much as possible. This indicates that equi-depth uses more resources than DeepSea to answer the same queries.

We now investigate how characteristics of the workload affect the performance of DeepSea compared to non-adaptive partitioning approaches such as equi-depth. In addition to measuring time for running a workload of 10 such queries, we also project the time (using linear regression) for 100 queries. Figure 7 shows the performance of materialization without partitioning (NP), materialization using an equi-depth partition of a fixed size (E), and our DeepSea approach using workload-aware partitioning (DS) compared to Hive on a 500GB dataset. The settings are indicated by concatenating the abbreviations for the selectivity and query-skew settings, for example, ML stands for medium selectivity and a lightly skewed distribution over the selection ranges.

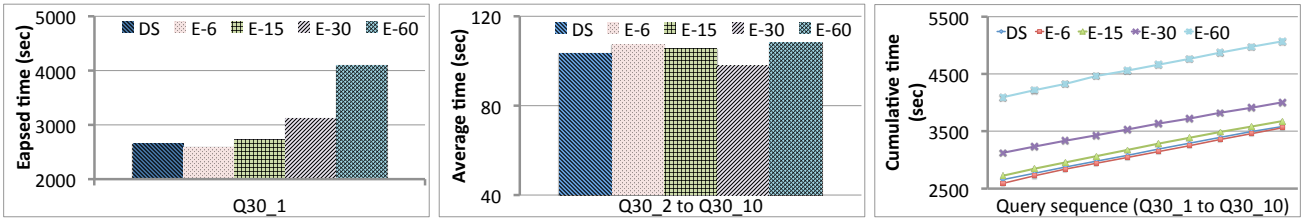
Figure 7a shows that both partitioning techniques (DeepSea and equi-depth) perform well compared to Hive and non-partition materialization in all experiments. When the selectivity is large, (indicated by B), our partition techniques can save 50 to 60% compared to Hive. For medium (M) selectivity, the partition techniques can save 60 to 70% and for small (S) selectivity the partition techniques can save 70 to 80%. Materialization alone without partitioning (NP) provides only a 15 to 25% improvement over Hive.

For uniformly distributed selections, DeepSea, as expected, does not provide a performance improvement over an equi-depth strategy (E). This is because equi-depth is tailored for such a distribution and the adaptive techniques of DeepSea do not pay off. However, for lightly skewed and heavily skewed selections, DeepSea has a noticeable advantage (up to 30%) over equi-depth partitioning. The performance of DeepSea increases and that of equi-depth decreases when introducing more skew (switching from uniformly distributed to lightly skewed and heavily skewed workloads). This is because we use the same number of fragments for DeepSea and equi-depth. When the workload is more and more skewed, there are fewer and smaller fragments needed by DeepSea to get the same benefit achieved by equi-depth.

Most optimizers will push down selections for reducing the size of intermediate results. Our materialization strategy requires that selections are not pushed down and hence we incur a performance hit initially. But even for small selectivities, this cost is quickly amortized over a workload. To understand when the additional work DeepSea does (by not pushing selections) is worth the cost, we plot the number of queries needed to recoup the cost of DeepSea in Figure 7b. Notice that for both DeepSea and equi-depth partitioning, the cost of not pushing a selection is recouped at almost the same point unless the workload is heavily skewed and includes queries with a large selectivity (requesting large portions of the data) in which case DeepSea has an advantage.

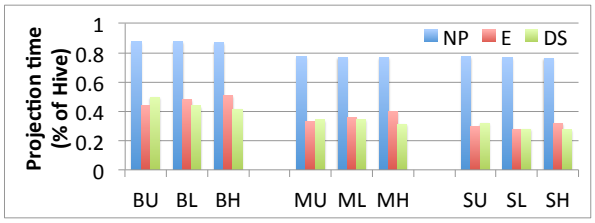
10.3 Exploitation of Fragment Correlations

We now compare our selection strategy that exploits fragment correlations against Nectar's strategy that is oblivious of such correlations. We use a workload that consists of ten queries (template Q30) that have big selectivity and are heavily skewed followed by another ten queries (also template Q30) that have small selectivity and are heavily skewed. We use a 500GB dataset with the pool

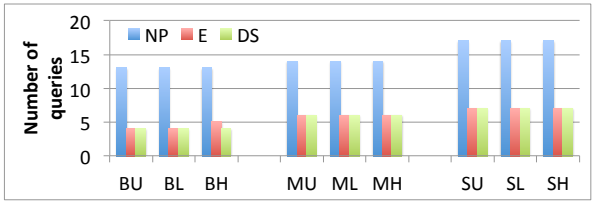


(a) Instrumented query materializing a view (b) Rewritten queries reusing a view (c) Cumulative time over the whole workload

Figure 6: Comparing equi-depth vs. adaptive partitioning (DeepSea) over workload using 10 instances of query template Q30, 100GB



(a) Expected elapsed time for 100 queries (% of Hive)

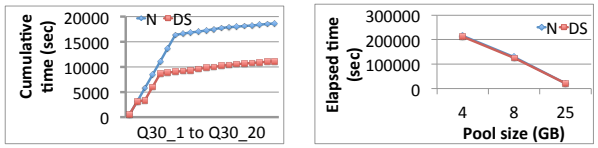


(b) # of queries needed to recoup materialization cost

Figure 7: Varying selectivity and skew, Q30, 500GB

size limit set to 7GB. Figure 8a shows that DeepSea benefits from smoothing the distribution of hits to fragments from the same partition and, thus, is more likely to keep fragments that are similar to frequently accessed fragments.

Recall that we smoothen the distribution of hits over an attribute's range by fitting it to a normal distribution. Figure 8 shows how the performance of our approach is affected by the distribution underlying the selections in a workload. DeepSea significantly outperforms Nectar's selection strategy if the real hits follow a normal distribution. Importantly, it does not perform worse than Nectar if the selection ranges follow a radically different distribution (Zipf).



(a) Normal (b) Zipf

Figure 8: Selection ranges following Normal resp. Zipf distribution

10.4 Overlapping Partitioning

A key benefit of overlapping partitioning is that it writes less data when repartitioning for certain patterns that we observe in real-life applications frequently. In order to compare overlapping partitioning with horizontal partitioning, we generate a workload sequence of 30 queries from template Q30 with small selectivity and heavy skew. The selections of Q30_1 to Q30_10 have a midpoint of 20,000, the selections of Q30_11 to Q30_20 have a midpoint of 40,000, and the selections of Q30_21 to Q30_30 have a midpoint of 60,000. The domain of the selection attribute is [0, 400,000]. We generate this workload to simulate the common query selection pattern that we have observed in SDSS.

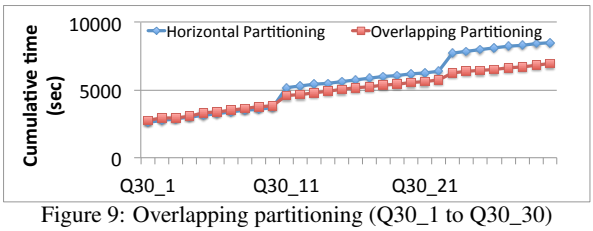
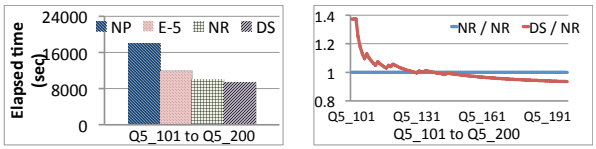


Figure 9: Overlapping partitioning (Q30_1 to Q30_30)

We are switching the pattern between Q30_10 and Q30_11 and between Q30_20 and Q30_21. Figure 9 shows that overlapping partitioning is more robust against changes in the workload, because it avoids writing a fragment that extends from the current upper bound of the selections to the upper bound of the domain that has not been queried yet.

We also generated a workload with 200 queries using query template Q5, all of which have big selectivity and are heavily skewed. The selection ranges for the first 100 queries were sampled from one distribution while the selection ranges for the next 100 queries follow a different distribution. Running this workload on the 100GB dataset, we compare against materialization without partitioning (NP in the graph), equi-depth partitioning with 5 fragments (E-5 in the graph) and DeepSea with no repartitioning (NR in the graph). Figure 10a shows for changing workloads, DeepSea outperforms the non-progressive approach that never repartitions by 7% and equi-depth partitioning by 27%. Figure 10b shows the cumulative time of DeepSea normalized to the cumulative time of the NR approach (no repartitioning), from query 101 (the first query following the new distribution) to query 200. DeepSea performs worse than NR for the first 30 queries because of the cost of repartitioning. This cost, however, is amortized by the subsequent queries.



(a) Cumulative time (b) Cumulative time ratio (DS/NR)

Figure 10: Adaptation to workload changes, Q5, 100GB

11. CONCLUSIONS

DeepSea is the first adaptive, progressive, workload-aware approach for automatic materialization and partitioning of views. Our

cost-benefit model for both views and fragments takes the correlations among fragments into account. Our progressive partitioning accommodates both dynamic analytic workloads and exploratory workloads where users explore multiple regions in the data before finding (and then focusing on) a region of interest. DeepSea is implemented in Hive and our experiments demonstrate that our approach is more effective than traditional materialization techniques that do not consider the physical design of materialized views or do not adapt online to the workload. We also demonstrate that for real workloads, our view/fragment selection strategy outperforms state-of-the-art selection techniques when the materialized view pool has a small size limit.

In the short term, there are several interesting ways in which we can improve DeepSea including considering how to merge consecutive fragments that are mostly accessed together and how to best partition views on multiple attributes. DeepSea contributes to a rich literature on adaptive, progressive physical design strategies. For a fixed *memory overhead*, DeepSea selects a set of partitioned views and fragments of views to optimize the query performance (or minimize the *read overhead*). In the future, we would like to consider updates and explore how our techniques could be used with different optimization goals (including minimizing update overhead). We also would like to integrate our approach with query optimization, this would allow us to explore strategies that potentially select a more expensive query plan if it allows the materialization of interesting views that could benefit the workload.

12. ACKNOWLEDGMENTS

We thank Tilmann Rabl and Michael Frank for providing BigBench for our experiments. Stan Zdonik and Nesime Tatbul gave advice to improve the work. We also thank the anonymous reviewers for their helpful suggestions. This research was funded in part by a Bell Graduate Scholarship and NSERC.

13. REFERENCES

[1] Hadoop. http://hadoop.apache.org/.

[2] Sloan Digital Sky Survey. http://cas.sdss.org/.

[3] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.

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

[5] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD*, pages 683–694, 2006.

[6] I. Alagiannis, D. Dash, K. Schnaitter, A. Ailamaki, and N. Polyzotis. An automated, yet interactive and portable DB designer. In *SIGMOD*, pages 1183–1186, 2010.

[7] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *SIGMOD*, pages 1103–1114, 2014.

[8] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, pages 156–165, 1997.

[9] N. Bruno and S. Chaudhuri. To tune or not to tune?: a lightweight physical design alerter. In *VLDB*, pages 499–510, 2006.

[10] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB*, pages 3–14, 2007.

[11] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan, et al. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

[12] I. Elghandour and A. Aboulnaga. ReStore: Reusing Results of MapReduce Jobs. *PVLDB*, 5(6):586–597, 2012.

[13] A. Ghazal, M. Hu, T. Rabl, F. Raab, M. Poess, A. Crolotte, and H. Jacobson. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *SIGMOD*, pages 1197–1208, 2013.

[14] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. *SIGMOD Rec.*, 30(2):331–342, 2001.

[15] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI*, pages 75–88, 2010.

[16] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[17] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.

[18] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[19] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. *TODS*, 35(4):309–320, 2010.

[20] S. Jain, D. Moritz, B. Howe, and E. Lazowska. SQLShare: Results from a multi-year sql-as-a-service experiment. In *SIGMOD*, pages 281–293, 2016.

[21] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views (extended abstract). In *PODS*, pages 95–104, 1995.

[22] F. Nagel, P. Boncz, and S. D. Viglas. Recycling in pipelined query evaluation. In *ICDE*, pages 338–349, 2013.

[23] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, pages 383–392, 2004.

[24] L. Perez and C. Jermaine. History-aware query optimization with materialized intermediate views. In *ICDE*, pages 520–531, 2014.

[25] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.

[26] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *ICDE*, pages 459–468, 2007.

[27] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[28] D. Wackerly, W. Mendenhall, and R. Scheaffer. *Mathematical Statistics with Applications*. Duxbury Press, 5th edition, 1996.

[29] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD*, pages 105–116, 2000.