MISO: Souping Up Big Data Query Processing with a Multistore System

Jeff LeFevre[+†] Jagan Sankaranarayanan[*] Hakan Hacıgümüş[*]
Junichi Tatemura[*] Neoklis Polyzotis[+] Michael J. Carey[%]
[*]NEC Labs America, Cupertino, CA [+]University of California, Santa Cruz [%]University of California, Irvine
{jlefevre,alkis}@cs.ucsc.edu, {jagan,hakan,tatemura}@nec-labs.com, mjcarey@ics.uci.edu

ABSTRACT

Multistore systems utilize multiple distinct data stores such as Hadoop's HDFS and an RDBMS for query processing by allowing a query to access data and computation in both stores. Current approaches to multistore query processing fail to achieve the full potential benefits of utilizing both systems due to the high cost of data movement and loading between the stores. Tuning the physical design of a multistore, i.e., deciding what data resides in which store, can reduce the amount of data movement during query processing, which is crucial for good multistore performance. In this work, we provide what we believe to be the first method to tune the physical design of a multistore system, by focusing on which store to place data. Our method, called MISO for MultISstore Online tuning, is adaptive, lightweight, and works in an online fashion utilizing only the by-products of query processing, which we term as *opportunistic views*. We show that MISO significantly improves the performance of ad-hoc big data query processing by leveraging the specific characteristics of the individual stores while incurring little additional overhead on the stores.

1. INTRODUCTION

Parallel relational database management systems (RDBMS) have long been the "work horse" storage and query processing engine of choice for data warehousing applications. Recently there has been a substantial move toward "Big Data" systems for massive data storage and analysis; HDFS with Hive is perhaps the most common example of such a system. The arrival of big data stores has set off a flurry of research and development, initially dealing with questions such as which store is best suited for which purpose; however, the growing consensus is that both stores have their place, and many organizations simultaneously deploy instances of each. Currently these stores have different roles – the big data store for exploratory queries to find business insights and the RDBMS for business reporting queries, as depicted in Figure 1(a).

Having both a parallel RDBMS and a big data store inevitably raises questions about combining the two into a "multistore" system, either for performance reasons or for analysis of data sets that span both stores. There have been a number of interesting proposals for this

[†]Work done when the author was at NEC Labs America

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2588568 .

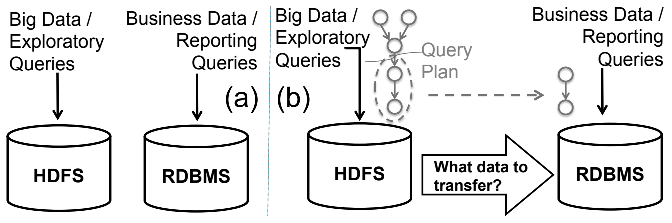

Big Data / Exploratory Queries
Business Data / Reporting Queries
(a)
HDFS
RDBMS
(b)
Big Data / Exploratory Queries
Query Plan
Business Data / Reporting Queries
HDFS
What data to transfer?
RDBMS

Figure 1: Typical setup in today's organizations showing (a) two independent stores and (b) multistore system utilizing both stores.

combination [2, 9, 11, 22]. In this paper we explore a simple yet powerful and ubiquitous opportunity: using the parallel RDBMS as an accelerator for big data queries, while preserving the above-mentioned roles for both stores.

Multistore systems represent a natural evolution for big data analytics, where query processing may span both stores, transferring data and computation as depicted in Figure 1(b). One approach to multistore processing is to transfer and load all of the big data into the RDBMS (i.e., up-front data loading) in order to take advantage of its superior query processing performance [17] relative to the big data store. However, the large size of big data and the high cost of an ETL process (Extract-Transform-Load) may make this approach impractical [17, 21]. Another approach is to utilize both stores during query processing by enabling a query to transfer data on-the-fly (i.e., on-demand data loading). However, this results in redundant work if the big data workload has some overlap across queries, as the same data may be repeatedly transferred between the stores.

A more effective strategy for multistore processing is to make a tradeoff between up-front and on-demand data loading. This is challenging since exploratory queries are ad-hoc in nature and the relevant data is changing over time. A crucial problem for a multistore system is determining *what* data to materialize in *which* store at *what* time. We refer to this problem as tuning the physical design of a multistore system.

In this paper we present the first method to tune the physical design of a multistore system. While physical design tuning is a well-known problem for a conventional RDBMS, it acquires a unique flavor in the context of multistore systems. Specifically, tuning the physical design of a multistore system has two decision components: *what* data to materialize (i.e., which materialized views) and *where* to materialize the data (i.e., which store). Our primary focus in this paper is on the *where* part since this is critical to good multistore query performance. Furthermore, since the different stores are coupled together by a multistore execution engine, the decision of where to place the data creates a technically interesting problem that has not been addressed in the existing literature. Beyond its technical merits, we argue that the problem actually lies at the core of multistore system design — regardless of how multistore systems are setup (i.e., tightly versus loosely coupled) or the way they are used (i.e., enabling data analysis spanning stores versus query acceleration). Without a well-tuned design, a multistore system will be forced to split query processing between the big data store and the RDBMS in a way that does not leverage the full capabilities of each system.

We introduce *MISO*, a MultIStore Online tuning algorithm, to "soup up" big data query processing by tuning the set of materialized views in each store. MISO maximizes the utilization of the existing high-performance RDBMS resources, employing some of its spare capacity for processing big data queries. Given that the RDBMS holds business data that is very important to an organization, DBAs are naturally protective of RDBMS resources. To be sensitive to this concern, it is important that our approach achieves this speedup with little impact on the RDBMS reporting queries. MISO is an adaptive and lightweight method to tune data placement in a multistore system, and it has the following unique combination of features:

- It works online. The considered workload is ad-hoc as the analysts continuously pose and revise their queries. The multistore design automatically adapts to the dynamically changing workload.

- It uses materialized views as the basic element of physical design. The benefits of utilizing materialized views are that views encapsulate prior computation, and the placement of views across the stores can obviate the need to do costly data movement on-the-fly during query processing. Having the right views in the right store at the right time creates beneficial physical designs for the stores.

- It relies solely on the by-products of query processing. Query processing using HDFS (e.g., Hadoop jobs) materializes intermediate results for fault-tolerance. Similarly, query processing in multistore systems moves intermediate results between the stores. In both cases we treat these intermediate results as *opportunistic* materialized views, which can be strategically placed in the underlying stores to optimize the evaluation of subsequent queries. These views represent the current relevant data and provide a way to tune the physical design almost for free, without the need to issue separate costly requests to build materialized views.

Online physical design tuning has been well-studied, but previous works have not considered the multistore setting. Other previous work has studied query optimization for multistore systems in the context of a single query accessing data in different stores. In contrast, our proposed technique allows the multistore system to tailor its physical design to the current query workload in a low-overhead, "organic" fashion, moving data between the stores depending on common patterns in the workload. A key metric for evaluating big data queries is the time-to-insight (TTI) [13], since this metric represents the total time to get answers from data — which includes more than query execution time. TTI includes data-arrival-to-query time (the time until new data is queryable), the time to tune the physical design, *and* the query execution time. TTI represents a holistic time metric for ad-hoc workloads over big data, and our approach results in significant improvements for TTI compared to other multistore approaches.

In the following sections we describe the MISO approach to speeding up big data queries by utilizing the spare capacity available in the RDBMS, performing lightweight online tuning of the physical design of the stores. In Section 2 we provide an overview of the related work, and we describe our multistore system architecture in Section 3. We describe the problem of multistore design and its unique challenges in Section 4. In the remainder of the section we present MISO, which solves the multistore design problem by tuning both stores. Since we take an online approach, the design adapts to changing workloads. Furthermore, as the design is opportunistic, MISO imposes little additional overhead. In Section 5 we provide experimental results showing the benefits of MISO in improving TTI, up to $3.1\times$, and further analyze where the improvements come from. Then in Section 5.4, we show that MISO has very little impact on an RDBMS that is already executing a workload of reporting queries. Finally, we summarize our findings in Section 6.

2. RELATED WORK

Multistore query processing. Earlier work from Teradata [25] and HadoopDB [2] tightly integrates Hadoop and a parallel data warehouse, allowing a query to access data in both stores by moving (or storing) data (i.e., the working set of a query) between each store as needed. These approaches are based on having corresponding data partitions in each store co-located on the same physical node. The purpose of co-locating data partitions is to reduce network transfers and improve locality for data access and loading between the stores. However, this requires a mechanism whereby each system is aware of the other system's partitioning strategy; the partitioning is fixed, and determined up-front. More recent work [9,22] presents multistore query optimizers for Hadoop and a parallel RDBMS store. These systems "split" a query execution plan across multiple stores. Data is accessed either in its native store, or via an external table declaration. A single query is executed across both systems by moving the working set between stores during query processing. These systems represent "connector approaches" between Hadoop and an RDBMS, using a data transfer mechanism such as Sqoop [4] (or their own in-house equivalent).

Recent work on Invisible Loading [1] addresses the high up-front cost of loading raw files (Hadoop data) into a DBMS (column-store). In the spirit of database cracking [12], data is incrementally loaded and reorganized in the DBMS by leveraging the parsing and tuple extraction done by MR jobs, and a small fraction of the extracted data is silently loaded by each MapReduce query. The data loaded represents horizontal and vertical sub-partitions of the Hadoop base data. This approach also represents opportunistic design tuning, but the views belong to a very specific class, i.e., horizontal and vertical fragments of base data. Furthermore, this approach requires the corresponding processing nodes of each store to be co-located on the same physical node (similar to the Teradata work). In this approach, the query acceleration obtained is primarily due to the improved data access times for HadoopDB since data is stored as structured relations in an RDBMS rather than flat files in HDFS. In contrast, our approach supports a more general class of materialized views, which essentially allows us to share computation across queries instead of only improving access to base data. Moreover, we allow the big data system and the RDBMS to remain separate without modifying either system or requiring them to be tightly integrated.

Finally, multistore systems share some similarities with federated databases in that both use multiple databases working together to answer a single query. However, with a federated system there is typically a strong sense of data and system ownership, and queries are sent to the location that owns the data to answer the query. With our multistore scenario, the roles of the stores are different — queries are posed on the base data in HDFS, while the RDBMS is only used as a query accelerator.

Reusing MapReduce computations. Previous work has shown that intermediate results materialized during query processing in Hadoop can be cached and reused to improve query performance [10, 15]. The work in [10] reuses these results at the syntactic level. This means that a new query's execution plan must have an identical subplan (operations and data files) to a previous query in order to reuse the previous results. Some recent work [15] utilizes Hadoop intermediate results as opportunistic materialized views at the semantic level and reuses them by a query rewriting method. These methods all focus on a single system and it is not straightforward to extend these approaches to a multistore setup. In MISO we treat both the intermediate materializations in Hadoop and the working sets transferred during multistore query processing as opportunistic materialized views, and we reuse

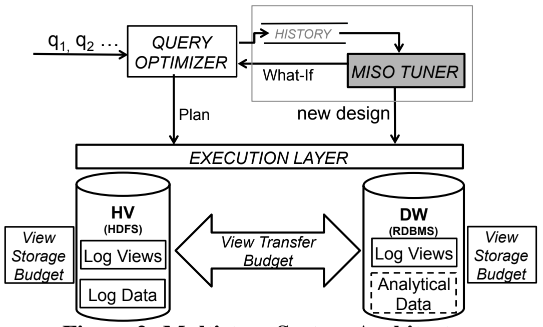


Figure 2: Multistore System Architecture

them at a semantic level. We then use these views to tune the physical design of both stores together.

Physical design tuning. Much previous work [5, 18, 19] has been done on online physical design tuning for centralized RDBMS systems. The objective of online tuning is to minimize the execution cost of a dynamically changing workload, including the cost to tune (reconfigure) the physical design. New designs may be materialized in parallel with query execution or during a reconfiguration phase. A reconfiguration phase may be time-constrained. For example, a new design may be materialized during a period of low system activity. Each of these prior works considers a single database system and uses indexes as the elements of the design. Our work considers a multistore system and utilizes opportunistic materialized views as the design elements. Tuning a multistore system adds interesting dimensions to the online tuning problem, particularly since data is being transferred between stores (similar to reconfiguration) during query processing. Obtaining good performance hinges upon a well-tuned design that leverages the unique characteristics of the two systems.

Other recent work on tuning has addressed physical design for replicated databases [8] where each RDBMS is identical and the workload is provided offline. In contrast, our work addresses online workloads and the data management systems in a multistore scenario are not identical. Moreover, with multistore processing the systems are not standalone replicas but rather data (i.e., views) and computation may move fluidly between the stores during query processing.

Recent work [16] has shown how to apply multi-query optimization (MQO) methods to identify common sub-expressions in order to produce a beneficial scheduling policy for a batch of concurrently executing queries. Due to the materialization behavior of MapReduce, these approaches can be thought of as a form of view selection, since the common sub-expressions identified are effectively beneficial views for a set of in-flight queries. However, MQO approaches require that the queries are provided up-front as a batch, which is very different from our online case with ad-hoc big data queries; hence, MQO approaches are not directly applicable to our scenario.

To summarize, all previous work has focused on either multistore processing for a single query or tuning a single type of data store. By dynamically tuning the multistore physical design, our approach utilizes the combination of these two stores to speed up a big data query workload.

3. SYSTEM ARCHITECTURE AND MULTISTORE TUNING

In a multistore scenario there are two data stores, the big data store and the RDBMS. In our system, the big data store is Hive (HV) and the RDBMS is a commercial parallel data warehouse (DW) as depicted in Figure 2. HV contains the big data log files, and DW contains analytical business data. While some multistore systems enable a single query to access data that spans both stores, MISO accelerates big data exploratory queries in HV by utilizing some limited spare capacity in DW and tuning the physical design of the stores for a workload of queries. In this way, MISO makes use of existing resources to benefit big data queries. With this approach, DW acts as an *accelerator* in order to improve the performance of queries in HV; later we show that MISO achieves this speedup with very little impact on DW.

Figure 2 depicts our system architecture, containing the *MISO Tuner*, the *Query Optimizer*, the *Execution Layer*, and the two stores HV and DW. In our system, each store resides on an independent cluster. HV contains log data while DW contains analytical data, but in this work our focus is on accelerating big data queries posed only on the log data. As indicated in the figure, each store has a set of materialized *views* of the base data (i.e., logs) stored in HV, and together these views comprise the multistore physical design.

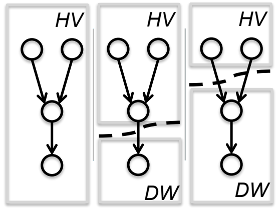
For fault-tolerance, Hadoop-based systems (e.g., HV) materialize intermediate results during query execution, and we retain these by-products as *opportunistic* materialized views. On the other hand, DW products generally do not provide access to their intermediate materialized results. However, if those results become available, then they can also be used for tuning. In our system, we consider the class of opportunistic materialized views when tuning the physical design. It is MISO Tuner's job to determine the placement of these views across the stores to improve workload performance. When placing these views, each store has a view storage budget as indicated in the figure, and there is also a transfer budget when moving views across the stores. These budgets limit the total size of views that can be stored and transferred.

3.1 System Components

Data sets. The primary data source in our system is large log files. Here we expect social media data drawn from sites such as Twitter, Foursquare, Instagram, Yelp, etc. This type of data is largely text-based with little structure. Logs are stored as flat HDFS files in HV in a text-based format such as JSON or XML.

Queries. The input to the system is a stream of queries $q_1, q_2, \cdots$ as indicated in Figure 2. The query language is HiveQL, which represents a subset of SQL implemented by Hive (HV). Queries are declarative and posed directly over the log data, such that the log schema of interest is specified within the query itself and is extracted during query execution. In HV, extracting flat data from text files is accomplished by a SerDe (serialize/deserialize) function that understands how to extract data fields from a particular flat file format (e.g., JSON). A query q may contain both relational operations and arbitrary code denoting user-defined functions (UDFs). UDFs are arbitrary user code, which may be provided in several languages (e.g., Perl, Python); the UDFs are executed as Hadoop jobs in HV. Queries directly reference the logs and are written in HiveQL, the system automatically translates query sub-expressions to a DW-compliant query when executing query sub-expressions in DW. The processing location (DW or HV) is hidden from the end user, who has the impression of querying a single store.

Execution Layer. This component is responsible for forwarding each component of the execution plan generated by the *Query Optimizer* to the appropriate store. A multistore execution plan may contain "split points", denoting a cut in the plan graph whereby data and computation is migrated from one store to the other.



Since DW is used as a accelerator for HV queries, the splits in a plan move data and computation in one direction: from HV to DW. It is the multistore query optimizer's job (described next) to select the split points for the plan. As an example, the figure alongside has three panels, showing an execution plan (represented as a DAG) and then two possible split points indicated by the cuts. At each split point, the execution layer migrates the intermediate data (i.e., the working set corresponding to the output of the operators above the cut) and resumes executing the plan on the new store. Note that in the second panel, one intermediate data set needs to be migrated by the execution layer, while in the third panel two intermediate data sets need to be migrated. When intermediate data sets are migrated during query execution, they are stored in temporary DW table space (i.e., not catalogued) and discarded at the end of the query. The execution layer is also responsible for orchestrating the view movements when changes to the physical design are requested by the tuner. When views are migrated during tuning, they are stored in permanent DW table space and become part of the physical design until the next tuning phase.

Multistore Query Optimizer. This component takes a query q as input, and computes a multistore execution plan for query q. The plan may span multiple stores, moving data and computation as needed, and utilizes the physical design of each store. Multistore query optimizers are presented in [9, 22], and our implementation is similar to that in [22]. As noted in [22,23], the design of an optimizer that spans multiple stores must be based on common cost units (expected execution time) between stores, thus some unit normalization is required for each specific store. Our multistore cost function considers three components: the cost in HV, the cost to transfer the working set across the stores, and the cost in DW, expressed in normalized units. For HV we use the cost model given in [16] and for DW we use its own cost optimizer units obtained from its "what-if" interface. We performed experiments to calibrate each system's cost units to query execution time, similar to the empirical method used in [22]. As it turns out, when the necessary data for q was present in DW, it was always faster to execute q in DW by a very wide margin. Because the stores have very asymmetric performance, the HV units completely dominate the DW units, making a rough calibration/normalization of units sufficient for our purposes.

Because the choice of a split point affects query execution time, it is important to choose the right split points. The multistore query optimizer chooses the split points based on the logical execution plan and then delegates the resulting sub-plans to the store-specific optimizers. The store in which query sub-expressions are executed depends on the materialized views present in each store. Furthermore when determining split points the optimizer must also consider valid operations for each store, such as a UDF that can only be executed in HV. Moving a query sub-expression from one store to another is immediately beneficial only when the cost to transfer and load data from one store to another plus the cost of executing the sub-expression in the other store is *less than* continuing execution in the current store. Furthermore, due to the asymmetric performance between the stores we observed that when the data (views) required by the sub-expression was already present in DW, the execution cost of the sub-expression was always lower in DW. The primary challenge for the multistore query optimizer is determining the point in an execution plan at which the data size of a query's working set is "small enough" to transfer and load it into DW rather than continue executing in HV.

In our implementation, we have added a "what-if" mode to the optimizer, which can evaluate the cost of a multistore plan given a hypothetical physical design for each store. The optimizer uses the rewriting algorithm from [15] in order to evaluate hypothetical physical designs being considered by the MISO Tuner.

MISO Tuner. This componentis invoked periodically to reorganize the materialized views in each store based on the "latest traits" of the workload. The tuner examines several candidate designs and analyzes their benefit on a sliding window of recent queries (*History* in Figure 2) using the what-if optimizer. The selected design is then forwarded to the execution layer, which then moves views from HV to DW and from DW to HV (indicated in Figure 2) as per the newly computed design. The invocation of the MISO tuner, which we term a *reorganization phase*, can be time-based (e.g., every 1h), query-based (e.g., every n queries), activity-based (e.g., when the system is idle), or a combination of the above. In our system, reorganizations are query-based.

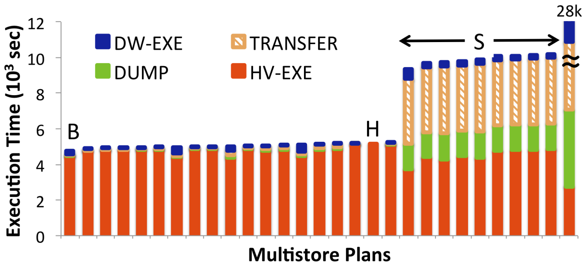


Figure 3: Execution time profile of all multistore plans for a single query. Plans are ordered by increasing execution time.

The *View Transfer Budget* is indicated in Figure 2 by the arrow between the stores. This represents the total size of views in GB transferred between the stores during a reorganization phase and is provided as a constraint to the tuner. The HV *View Storage Budget* and the DW *View Storage Budget* are also indicated in the figure and similarly provided as constraints to the tuner. These represent the total storage allotted in GB for the views in each store. While DW represents a tightly-managed store, HV deployments are typically less tightly-managed and may have more spare capacity than DW. For these reasons, new opportunistic views created in HV between reconfigurations are retained until the next time the MISO tuner is invoked, while the set of views in DW is not altered except during reorganization phases. In any Hadoop configuration, there must be enough temporary storage space to retain these materializations during normal operation, we only propose to keep them a little while longer – until the next reorganization phase. Since the HV view storage budget is only enforced at tuning time, a few simple policies can be considered if the system has limited temporary space: Reorganizations could be done more frequently, a simple LRU policy could be applied to the temporary space, or a small percentage of the HV view storage budget could be reserved as additional temporary space.

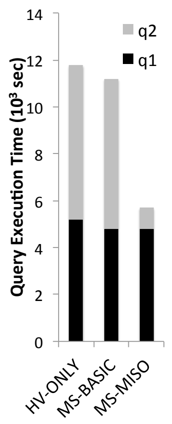
3.2 Overview of Multistore Tuning

Tuning the physical design of a multistore system is important to obtaining good query performance. In a multistore system, the query optimizer can choose many different split points at which to migrate computation from one store to the other during query evaluation. The overall execution time of a query is dependent on the existing physical design of each store, and varies depending on which portion of the query is evaluated in which store. A well-tuned physical design enables query execution plans to migrate to the DW sooner rather than later, thus making more extensive use of the DW's superior query-processing capabilities. In contrast, without a good physical design, the high cost of data movement will cause a query to spend more time in the slower HV store, which in turn provides limited opportunities for optimization. We demonstrate this point with the following simple experiment.

Figure 3 provides the execution time profiles (ordered from lowest to highest) for all possible plans of a single query, where each plan represents a unique split. The query is a complex query from [14] (query A_1v_1), that accesses social media data (e.g., Twitter, Foursquare), has multiple joins, computes several aggregates, and contains UDFs. These queries relate to marketing scenarios for a new product, and they try to identify potential customers based on their interests as identified in social media data. While the figure only shows one query, the trends observed were typical of all queries that we tested. Each plan begins execution in HV, and for each split point, the intermediate data is migrated to DW where the remainder of the plan is executed. For each plan, the stacked bar indicates the time spent executing in HV (red), the time spent to DUMP (green) and then TRANSFER/LOAD (yellow) the intermediate data over the network into DW, and finally the time spent executing in DW (blue).

The plan with the lowest execution time is on the far left of the figure, marked B. This plan is only 10% faster than the HV-only execution plan marked H. The worst plan in the figure is on the far right, at 28,000 seconds, and the bar is truncated here for display purposes. This plan represents the earliest possible split point, using HV only as an ETL engine to load all data into DW and then execute the query. From these observations it is clear that multistore execution offers little benefit for a single query (10% in the best case), and many multistore plans (those marked S) are *far more* expensive than the HV-only plan. This is due to the very high cost to transfer and load data from HV to DW, as can be seen in the yellow and green portions of the plans marked S. Additionally, there is a clear delineation between "good" plans (those to the left of S) and "bad" plans (those marked S).

We have found that the good plans all dump, transfer and then load much smaller data sets into DW than the bad plans, as shown in Figure 3. Good plans represent splits where the size of the intermediate data is small enough to offset the transfer and load costs. Unfortunately, we typically observed that the working set size does not shrink significantly until near the end of the query. This effect was also observed in Bing MapReduce workflows [3] (i.e., "little data" case). Another recent paper [22] examines a related experimental setup with two stores, and their optimizer considers ping-ponging a query (multiple sync/merges) back and forth across both stores. An examination of their experimental results reveals that the queries with the best performance are those that remain in Hadoop until the working set is small before spilling their remaining computation to the DW; effectively degenerating into the single-split multistore plans that we consider here. Hence, a query is only able to utilize DW for a small amount of processing, not benefiting much from DW's superior performance.



Tuning the physical design of HV and DW can facilitate earlier split points for multistore query plans. Better still, when the right views are present a query can bypass HV and execute directly in DW, thus taking advantage of DW's superior performance. The role of the tuner is to move the "right views" into the "right store". Again, we illustrate this point with another experiment on the same multistore system. In the figure alongside, we consider a workload of two exploratory queries, q_1 and q_2, that correspond to A_1v_2 and A_1v_3 from the workload in [14]. These are subsequent queries issued by the same data analyst, and thus have some overlap. Query q_1 is executed first, followed by q_2, and each query produces some opportunistic views when executed in HV.

The figure reports the total execution time for q_1 followed by q_2 using three different system variants. HV-ONLY represents an HV system that executes both queries in their entirety and does not utilize DW at all. MS-BASIC represents a basic multistore system without tuning, and is the same as the setting in Figure 3. This system does not make use of opportunistic views to speed up query processing. MS-MISO represents our multistore system using the MISO Tuner. To produce the MS-MISO result in the figure, we triggered a reorganization phase after q_1 but before q_2 executed. During reorganization, the Tuner transfers views between the stores in order to create beneficial designs for q_2. In the figure, MS-BASIC is only about 8% faster than HV-ONLY; while MS-MISO is about 2$\times$ faster than both HV-ONLY and MS-BASIC. While this result is for 2 related queries, for a larger workload MS-MISO results in up to 3.1$\times$ improvement later in Section 5. MS-MISO's improvement over MS-BASIC is because the tuner created a new multistore physical design after q_1, which enabled q_2 to take advantage of DW performance since the "right" views were present in DW.

4. MISO TUNER

In this section we describe the details of the MISO Tuner component shown in Figure 2. Because we address the online case where new multistore designs will be computed periodically, we desire a solution that is lightweight with low-overhead. This is achieved in the following ways. Since we utilize opportunistic views, the elements of our design are almost free, thus the tuner does not incur an initial materialization cost for a view. The reorganization time is controlled by the view transfer budget and view storage budgets for HV and DW. The values for these constraints can be kept small, in order to restrict the total tuning time. At the beginning of each reorganization phase, the tuner considers migrating views between the stores based on the recent workload history (indicated in Figure 2) and computes the new physical designs for HV and DW, subject to the view storage budget constraints and the view transfer budget.

Multistore systems notwithstanding, solving even a single physical design problem is computationally hard [6] because the elements of the design can interact with one another. Commercial tools (e.g., IBM DB2 Design Advisor [24]) take a heuristic approach to the physical design problem for a single store by treating it as a variant of the knapsack problem with a storage budget constraint. We similarly treat the multistore design problem as a variant of a knapsack problem. Our scenario is different in that we have two physical designs to solve, where each is dependent on the other, and each design has multiple dimensions — a view storage budget constraint and a view transfer budget constraint — that are consumed at different rates depending on the current design of each store. Due to the hardness of the problem, an optimal solution is impractical thus we take a heuristic approach. In order to keep our approach lightweight, we develop an efficient dynamic programming based knapsack solution.

4.1 Basic Definitions and Problem Statement

The elements of our design are materialized views and the universe of views is denoted as V. The physical design of each store is denoted as V_h for HV, and V_d for DW, where $V_h, V_d \subseteq V$. The values B_h, B_d denote the view storage budget constraints for HV and DW respectively, and the view transfer budget for a reorganization phase is denoted as B_t. As mentioned previously, reorganization phases occur periodically (e.g., after j queries are observed by the system), at which point views may be transferred between the stores. The constraints B_h, B_d, and B_t are specified in GB.

Let $\mathcal{M} = \langle V_h, V_d \rangle$ be a *multistore design*. $\mathcal{M}$ is a pair where the first component represents the views in HV, and the second component represents the views in DW. In this work we restrict the definition to include only two stores, but the definition may be extended to include addition stores in the more general case.

We denote the most recent n queries of the input stream as workload W. A query q_i represents the i^{th} query in W. The cost of a query q given multistore design $\mathcal{M}$, denoted by $cost(q, \mathcal{M})$, is the sum of the cost in HV, the cost to transfer the working set of q to DW and the cost in DW under a hypothetical design $\mathcal{M}$. The evaluation metric we use for a multistore design $\mathcal{M}$ is the total workload cost, defined as:

$$TotalCost(W, \mathcal{M}) = \sum_{i=1}^{n} cost(q_i, \mathcal{M}). \quad (1)$$

Intuitively, this metric represents the sum of the total time to process all queries in the workload. This is a reasonable metric, although others are possible. Note that the metric does not include the reorganization constraint B_t since it is provided as an input to the problem.

V_h, V_d	Design of HV and DW before reorg
V_{cands}	Candidate views for M-KNAPSACK packing
V_h^{new}, V_d^{new}	Design of HV and DW after reorg
V_d^-	Views evicted from DW ($= V_d - V_d^{new}$)
B_h, B_d, B_t	View storage and transfer budgets
B_t^{rem}	Transfer budget remaining after DW design
$sz(v), bn(v)$	Size and benefit of view $v \in V_{cands}$

Table 1: MISO Tuner variables.

The *benefit* of a view $v \in V$ for a q query is loosely defined as the change in cost of q evaluated with and without view v present in the multistore design. Formally, $benefit(q, v) = cost(q, \mathcal{M} \cup v) - cost(q, \mathcal{M})$, where $\mathcal{M} \cup v$ means that v is added to both stores in $\mathcal{M}$.

For a query q, a pair of views (a, b) may interact with one another with respect to their benefit for q. The interaction occurs when the benefit of a for q changes when b is present. We employ the *degree of interaction* (doi) as defined in [20], which is a measure of the magnitude of the benefit change. Furthermore, the type of interaction for a view a with respect to b may be positive or negative. A positive interaction occurs when the benefit of a increases when b is present. Specifically, a and b are said to interact positively when the benefit of using both is higher than the sum of their individual benefits. In this case, we want to pack both a and b in the knapsack. A negative interaction occurs when the benefit of a decreases when b is present. As an example, suppose that either a or b can be used to answer q. Hence both are individually beneficial for q, but suppose that a offers slightly greater benefit. When both views are present, the optimizer will always prefer a and thus never use b, in which case the benefit of b becomes zero. In this case packing both a and b in the knapsack results in an inefficient use of the view storage budget. We will utilize both the doi and the type of interaction between views when formulating our solution later.

Given our definitions of multistore design, the constraints, cost metric, and workload, we can define the multistore design problem.

Multistore Design Problem. Given an observed query stream, a multistore design $\mathcal{M} = \langle V_h, V_d \rangle$, and a set of design constraints B_h, B_d, B_t, compute a new multistore design $\mathcal{M}^{new} = \langle V_h^{new}, V_d^{new} \rangle$, where $V_h^{new}, V_d^{new} \subseteq V_h \cup V_d$, that satisfies the constraints and minimizes future workload cost.

4.2 MISO Tuner Algorithm

Because we desire a practical solution to the multistore design problem, we adopt a heuristic approach in the way we handle view interactions and solve the physical design of both stores. In this section we motivate and develop our heuristics and then later in the experimental section we show that our approach results in significant benefits compared to simpler tuning approaches.

The workload we address is online in nature, hence reorganization is done periodically in order to tune the design to reflect the recent workload. Our approach is to obtain a good multistore design by periodically solving a static optimization problem where the workload is given. At each reorganization phase, the tuner computes a new multistore design $\mathcal{M}^{new} = \langle V_h^{new}, V_d^{new} \rangle$ that minimizes total workload cost, as computed by our cost metric $TotalCost(W, \mathcal{M}^{new})$. Note that minimizing the total workload cost here is equivalent to maximizing the benefit of $\mathcal{M}^{new}$ for the representative workload W. The MISO Tuner algorithm is given in Algorithm 1, and its variables are defined in Table 1. We next give a high-level overview of the algorithm's steps and then provide the details in the following sections.

The first two steps given in lines 3 and 4 handle interactions between views as a pre-processing step before computing the new designs. The tuner algorithm begins by grouping all views in the current designs V_h and V_d into interacting sets (line 3). The goal of this step is to identify views that have strong positive or negative interactions with other views in V. At the end of this step, there are multiple sets of views,

Algorithm 1 MISO Tuner algorithm

```
1: function MISO_TUNE($\langle V_h, V_d \rangle, W, B_h, B_d, B_t$)
2:     $V \leftarrow V_h \cup V_d$
3:     $P \leftarrow$ COMPUTEINTERACTINGSETS($V$)
4:     $V_{cands} \leftarrow$ SPARSIFYSETS($P$)
5:     $V_d^{new} \leftarrow$ M-KNAPSACK($V_{cands}, B_d, B_t$)
6:     $B_t^{rem} \leftarrow B_t - \sum_{v \in V_h \cap V_d^{new}} sz(v)$
7:     $V_h^{new} \leftarrow$ M-KNAPSACK($V_{cands} - V_d^{new}, B_h, B_t^{rem}$)
8:     $\mathcal{M}^{new} \leftarrow \langle V_h^{new}, V_d^{new} \rangle$
9:     return $\mathcal{M}^{new}$
10: end function
```

where views within a set strongly interact with each other, and views belonging to different sets do not. We sparsify each set by retaining some of the views and discarding the others (line 4). To sparsify a set, we consider if the nature of the interacting views within a set is positive or negative. If the nature of the interacting views is strongly positive, then as a heuristic, those views should always be considered together since they provide additional benefit when they are all present. Among those views, we treat them all as a *single* candidate. If the nature of the interacting views is strongly negative, those views should not be considered together for $\mathcal{M}^{new}$ since there is little additional benefit when all of them are present. Among those views, we choose a representative view as a candidate and discard the rest. We describe these steps in detail in Section 4.3.

After the pre-processing step is complete, the resulting set of candidate views V_{cands} contains views that may be considered independently when computing the new multistore design, which is done by solving two multidimensional knapsack problems in sequence (lines 5 and 7). The dimensions of each knapsack are the storage budget and the transfer budget constraints. First on line 5, we solve an instance of a knapsack for DW, using view storage budget B_d and view transfer budget B_t. The output of this step is the new DW design, V_d^{new}. Then on line 7, we solve an instance of a knapsack for HV, using view storage budget B_h and any view transfer budget remaining (B_t^{rem}) after solving the DW design. The output of this step is the new HV design, V_h^{new}. The reason we solve the DW design first is because it can offer superior execution performance when the right views are present. With a good DW design, query processing can migrate to DW sooner thus taking advantage of its query processing power. For this reason, we focus on DW design as the primary goal and solve this in the first phase. After the DW design is chosen, the HV design is solved in the second phase. In this two-phase approach, the design of HV and DW can be viewed as complimentary, yet formulated to give DW greater importance than HV as a heuristic. We describe the two knapsack packings in detail in Section 4.4.

4.3 Handling View Interactions

Here we describe our heuristic solution to consider view interactions since they can affect knapsack packing. For example if a view v_a is already packed, and a view v_b is added, then if an interaction exists between v_a and v_b, the benefit of v_a will change when v_b is added. Our heuristic approach only considers the strongest interactions among views in V, producing a set of candidate views whereby we can treat each view's benefit as independent when solving the knapsack problems. This is because a dynamic programming formulation requires that the benefit of items already present in the knapsack does not change when a new item is added. We use a two-step approach to produce the candidate views V_{cands} that will be used during knapsack packing, described below.

Before computing the interacting sets, we first compute the expected benefit of each view by utilizing the predicted future benefit function from [18]. The benefit function divides W into a series of non-overlapping epochs, each a fraction of the total length of W. This represents the recent query history. In this method, the predicted future benefit of each view is computed by applying a decay on the view's benefit per epoch — for each $q \in W$, the benefit of a view v for query q is weighted less as q appears farther in the past. The outcome of the computation is a smoothed averaging of v's benefit over multiple windows of the past. In this way, the benefit computation captures a longer workload history but prefers the recent history as more representative of the benefit for the immediate future workload.

Compute Interacting Sets. To find the interacting sets, we use the method from [19] to produce a *stable partition* $P = \{P_1, P_2, \ldots, P_m\}$ of the candidate views V, meaning views within a part P_i do not interact with views in another part P_j. In this method, the magnitude of the interaction (i.e., change in benefit) between views within each part P_i is captured by *doi* [20]. When computing *doi*, we slightly modify the computation to retain the sign of the benefit change, where the sign indicates if the interaction is positive or negative. For a pair of views that have both positive and negative interactions, the magnitude is the sum of the interactions, and the sign of the sum indicates if it is a net positive or negative interaction. The partitioning procedure preserves only the most significant view interactions, those with magnitude above some chosen threshold. The threshold has the effect of ignoring weak interactions. The value of the threshold is system and workload dependent, but should be sufficiently large enough to result in parts with a small number (e.g., 4 in our case) of views thus only retaining the strongest interactions.

Since the parts are non-overlapping (i.e., views do not interact across parts), each part may be considered independently when packing M-KNAPSACK. This is because the benefit of a view in part P_i is not affected by the benefit of any view in part P_j, where $P_i \neq P_j$. At this point however, some parts may have a cardinality greater than one, so we next describe a method to choose a representative view among views within the same part.

Sparsify Sets. To sparsify the parts, we first take account of positively interacting views, then negatively interacting views. We exploit positive interactions by applying a heuristic to ensure that views that have a large positive interaction are packed together in the knapsack. Within each part, view pairs with positive interactions are "merged" into a single knapsack item, since they offer significant additional benefit when used together. The new item has a weight equal to the sum of the size of both views, and the benefit is the total benefit when both views are used together. This merge procedure is applied recursively to all positively interacting pairs within each part, in decreasing order of edge weight. Merging continues until there are no more positively interacting pairs of views within the part.

After applying the merge procedure, all parts with cardinality greater than 1 contain only strongly negative interacting views. Our next heuristic is to not pack these views together in the knapsack, but select one of them as a representative of the part. This is because packing these views together is an inefficient use of knapsack capacity. To choose the representative, we order items in the part by decreasing benefit per unit of weight and only retain the top item as a representative of the part. This greedy heuristic is commonly used for knapsack approaches to physical design [7, 24] and we use it here for choosing among those views with strong negative interactions within a part. The procedure is applied to each part until all parts have a single representative view. After these steps, the resultant partition P contains only singleton parts. These parts form V_{cands} and are treated as independent by M-KNAPSACK.

4.4 MISO Knapsack Packing

During each reorganization window, the MISO Tuner solves two instances of a 0-1 multidimensional knapsack problem (M-KNAPSACK henceforward); the first instance for DW and the second instance for HV. Each instance is solved using a dynamic programming formulation. At the beginning of each reorganization window, note that V_h includes all views added to the HV design by the MISO tuner during the previous reorganization window as well as any new opportunistic views in HV created since the last reorganization. During reorganization, the MISO tuner computes the new designs V_h^{new} and V_d^{new} for HV and DW respectively. Since the DW has better query performance, as a first heuristic MISO solves the DW instance first resulting in the best views being added to the DW design. Furthermore, we ensure $V_h \cap V_d = \emptyset$ to prevent duplicating views across the stores. Although this is also a heuristic, our rationale is it potentially results in a more "diverse" set of materialized views and hence better utilization of the limited storage budgets in preparation for an unknown future workload. If desired, this property could be relaxed by including all views in V_{cands} when packing both HV and DW.

As an additional heuristic we do not limit the fraction of B_t that can be consumed when solving the DW design in the first phase. Any remaining transfer budget B_t^{rem} can then be used to transfer views evicted from DW to HV. This means that all of the transfer budget could potentially be consumed during the first phase when transferring views to DW. Alternatively, we could reserve a fraction of B_t for transfers in each direction, although this too would be a heuristic.

4.4.1 *Packing DW M-Knapsack*

In this phase, the target design is V_d^{new}, and the M-KNAPSACK dimensions are B_d and B_t. There are two cases to consider when packing DW. Views in HV (V_h) will consume the transfer budget B_t (Case 1), while views in DW (V_d) will not (Case 2). The candidate views are $V_{cands} = V_h \cup V_d$. The variable k denotes the k^{th} element in V_{cands} (i.e.,view v_k), the order of elements is irrelevant. The recurrence relation C is given by the following two cases.

Case 1: $v_k \in V_h$ applies when view v_k is present in HV.

$$C(k, B_d, B_t) = \begin{cases} C(k-1, B_d, B_t) & ; sz(v_k) > B_t \\ \max \begin{cases} C(k-1, B_d, B_t), \\ C(k-1, B_d - sz(v_k), B_t - sz(v_k)) + bn(v_k) \end{cases} \\ & ; sz(v_k) \leqslant B_t, sz(v_k) \leqslant B_d \end{cases}$$

In this case, either element k does not fit in B_t (since $sz(v_k) > B_t$), and if so, skip element k and continue. Otherwise, k does fit in B_t and B_d (since $sz(v_k) \leqslant B_t$, and $sz(v_k) \leqslant B_d$), and if so take the max value of either (a) skip element k and continue, or (b) add element k to M-KNAPSACK, subtract its size from B_t and B_d, accumulate its benefit ($bn(v_k)$), and continue.

Case 2: $v_k \notin V_h$ applies when view v_k is not in HV.

$$C(k, B_d, B_t) = \begin{cases} C(k-1, B_d, B_t) & ; sz(v_k) > B_d \\ \max \begin{cases} C(k-1, B_d, B_t), \\ C(k-1, B_d - sz(v_k), B_t) + bn(v_k) \end{cases} \\ & ; sz(v_k) \leqslant B_d \end{cases}$$

In this case, either element k does not fit in B_d (since $sz(v_k) > B_d$), and if so, skip element k and continue. Otherwise, k does fit in B_d (since $sz(v_k) \leqslant B_d$), and if so take the max value of either (a) skip element k and continue, or (b) add element k to M-KNAPSACK, subtract its size from B_d, accumulate its benefit ($bn(v_k)$), and continue. At the end of this first phase the design of DW, V_d^{new}, is complete.

4.4.2 *Packing HV M-Knapsack*

In this phase, the target design is V_h^{new}, and the M-KNAPSACK dimensions are B_d and B_t^{rem}, and V_d^- represents the views evicted from the DW which are now available for transfer back to HV. The solution is symmetric to Phase 1, with modified inputs. The value B_t is initialized to B_t^{rem}. The candidate views are those remaining in V_h and those evicted from DW (V_d^-); therefore $V_{cands} = (V_h \cup V_d) - V_d^{new}$. We similarly have 2 cases, defined in Phase 2 as:

- Case 1: $v_k \in V_d^-$ applies when view v_k was evicted from DW.
- Case 2: $v_k \notin V_d^-$ applies when view v_k was not evicted from DW.

As before, these cases indicate whether a view needs to be moved or not. The recurrence relations from Phase 1 can now be used with these modified cases.

The complexity of our knapsack approach is $O(|V| \cdot \frac{B_t}{d} \cdot \frac{B_d}{d} + |V| \cdot \frac{B_t}{d} \cdot \frac{B_h}{d})$, where d represents the discretization factor (e.g., 1 GB). By discretizing the storage and transfer budgets, this approach can be made efficient in practice and suitable for our scenario that requires periodically recomputing new designs.

5. EXPERIMENTAL STUDY

In this section we present an experimental evaluation of the effectiveness of MISO for speeding up multistore queries. First we evaluate MISO in a multistore system in the absence of any other workload on DW in order to better understand the performance of our tuning techniques. Later we perform a more realistic evaluation of MISO by using a DW with limited spare capacity by running a background workload on DW. We then measure the impact of the multistore queries on the DW background workload and vice-versa. In Section 5.1 we describe the experimental methodology. In Section 5.2 we compare MS-MISO to several other system variants to highlight its benefit for multistore query processing. Then in Section 5.3 we compare the behavior MS-MISO to several possible multistore tuning algorithms. Lastly in Section 5.4, to evaluate MS-MISO in a realistic scenario we model a DW with an existing business workload and show the impact of executing multistore queries on the DW.

5.1 Methodology

Data sets and workloads. Our evaluation utilizes three data sets: a 1 TB Twitter data stream of user "tweets", a 1 TB Foursquare data stream of user "check-ins", and a 12 GB Landmarks data set represent static data containing geographical information about locations of popular interest. The identity of the user is common across Twitter and Foursquare logs, while the checkin location is common across the Foursquare and Landmarks logs. The data sets are stored in their native JSON format as logs in HDFS.

Our workload consists of 32 complex analytical queries given in [14] that access social media data and static historical data, for restaurant marketing scenarios. The queries model eight data analysts, each posing and iteratively refining a query multiple times during their data exploration. Each analyst (referred to as A_i for Analyst i) evolves a query through four versions denoted as $A_iv_1, A_iv_2, A_iv_3, A_iv_4$. An evolved query version A_iv_2 represents a mutation of the previous version A_iv_1, as described in [14], thus there is some overlap between queries. The queries contain relational operators as well as UDFs, and the queries are written in HiveQL with UDFs included via the Hive-CLI.

Stores. We utilize two distinct data stores: a Hive store (HV) and a parallel data warehouse (DW). Each store is a cluster of nodes, and the clusters are independent. The clusters are connected via 1GbE network and reside on adjacent racks. Each node has the same hardware: 2 Xeon 2.4GHz CPUs (8 cores), 16 GB of RAM, and exclusive access to its own disk (2TB SATA 2012 model). Additionally, the head nodes each have another directly attached 2 TB disk for data staging (used for data transfers and loading between the stores). HV runs Hive version 0.7.1 and Hadoop version 0.20.2, while DW runs the latest version of a widely-used, mature commercial parallel database (row-store) with horizontal data partitioning across all nodes. The HV cluster has 15 machines and the DW cluster has 9 machines. The HV cluster is 1.5× the size of the DW cluster since HV is less expensive than DW and thus in a realistic setting would typically be larger.

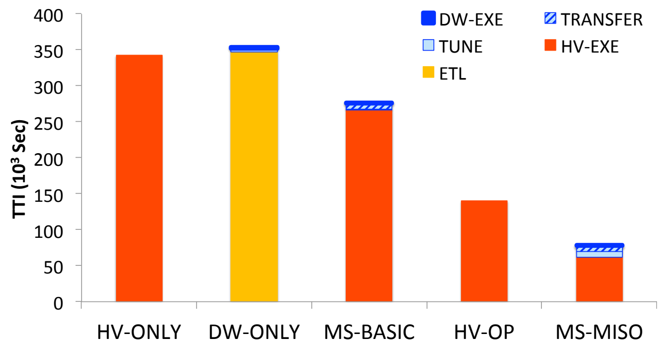


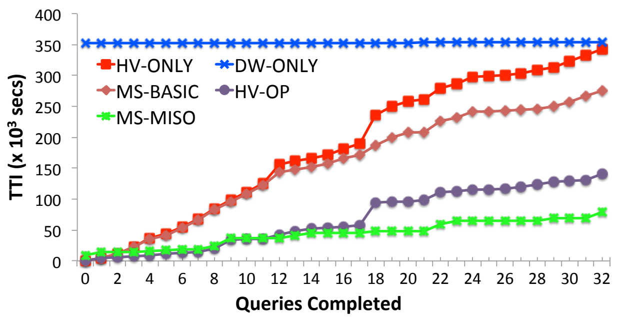
Figure 4: Performance results for 5 system variants.

Experimental parameters. For some experiments we vary the view storage budgets B_h, B_d and the view transfer budget B_t. B_h, B_d are varied as a fraction of the base data size in each store. Because the log data is stored in HV, we consider its base data size to be 2TB corresponding to the size of the logs. Because there is no log data stored in DW, we consider its "base data" size to be 200 GB corresponding to the relevant portion of the log data accessed by the queries. For example, $B_h = 2\times$ is 4TB, whereas $B_d = 2\times$ is 400GB. B_t is expressed in GB representing the size of the data transferred between the stores during each reorganization phase.

System variants. We evaluate the workload on several system variants described below. Note that all base data (logs) is stored in HV.

- HV-ONLY is the Hive store. Queries complete their entire execution in the 15 node HV store only.
- DW-ONLY is the data warehouse store. Queries complete their entire execution in the 9 node DW store only. Prior to query execution, we ETL the subset of the log data accessed by the queries using HV as an ETL engine. Note that using Hive or Pig for ETL is becoming a common industry practice (e.g., Netflix, Yahoo, Intel). [1] Any UDFs that do not execute in DW are applied as part of the ETL process in HV, before loading into DW.
- HV-OP is an HV store that retains opportunistic views and rewrites queries using these views, with the method from [15]. When views exceed the view storage budget, they are evicted using an LRU replacement policy. Queries complete their entire execution in the 15 node HV store only.
- MS-BASIC is a basic multistore system, that does not make use of opportunistic views. Queries may execute in part on both the 15 node HV store and the 9 node DW store as determined by the optimizer.
- MS-MISO is our multistore system that uses the MISO Tuner, which determines the placement of views in each store. Queries may execute in part on both the 15 node HV store and 9 node DW store, as determined by the optimizer which takes into account the current placement of views in each store. Reorganization phases (R) occur every 1/10 of the full workload, which in this case is after every 3 queries. When computing the expected future benefit (Section 4.3), we use a query history length of 6 and epoch length of 3 queries. During reorganizations, no queries are executed and MISO completes all view movements, after which time query execution begins again.

[1] http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html, http://developer.yahoo.com/blogs/hadoop/pig-hive-yahoo-464.html, http://hadoop.intel.com/pdfs/ETL-Using-Hadoop-on-Intel-Arch.pdf



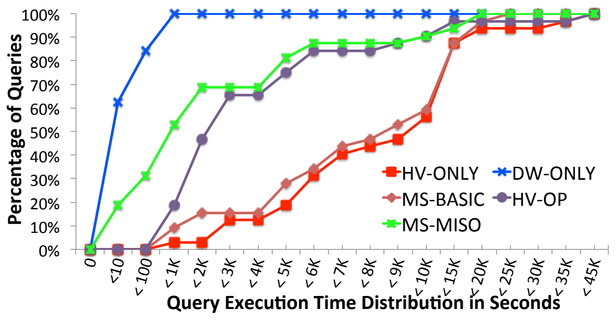


Figure 5: CDF plots as a function of percent workload complete for (a) TTI and (b) query execution time distribution.

Metrics. Our primary metric is TTI, representing the total elapsed time from workload submission to workload completion. TTI is defined as the cumulative time of loading data, transferring data during query execution, tuning the systems, and executing the queries. Wherever relevant, we break down TTI into the following components.

1. HV-EXE is the cumulative time spent executing queries in HV.
2. DW-EXE is the cumulative time spent executing queries in DW.
3. TRANSFER is the cumulative time for transferring (and loading) data between the stores during multistore query execution.
4. TUNE is the cumulative time spent computing new designs by a tuning algorithm, moving views between stores, and creating any recommended indexes for the views in DW.
5. ETL is the time spent loading the relevant data into DW using HV as an ETL engine before executing the workload. ETL time is only applicable to the DW-ONLY system variant.

5.2 Main Results

We first summarize the main results showing how MS-MISO outperforms other system variants. Figure 4 compares MS-MISO with the 4 other system variants – HV-ONLY, DW-ONLY, MS-BASIC, and HV-OP. In this experiment, HV-OP has a view storage budget of $2\times$ on HV, and views are retained in HV within the storage budget using a simple LRU caching policy. MS-MISO has storage budgets B_h and B_d of $2\times$ and B_t of 10GB. The systems variants are shown along the x-axis, and the y-axis reports TTI in seconds. For MS-MISO, TTI also includes time to compute new designs as well as the time spent moving views during reorganization phases. HV-ONLY represents the native system for our Hive queries, thus we use it as the baseline for comparing the other system variants.

Figure 4 shows MS-MISO has the best performance in terms of TTI, and DW-ONLY has the worst performance. DW-ONLY is actually 3% slower than HV-ONLY. This is because the vast majority of TTI for DW-ONLY is spent in the ETL phase, while the query execution time (DW-EXE) only constitutes a small fraction of the TTI. Although ETL is very expensive, it is a one-time process in our system and the cost is amortized by the benefit DW provides for all the queries in the workload. Clearly DW-ONLY would provide greater benefit for a longer workload. However, making this high up-front time investment to ETL all of the data is difficult to justify due to the exploratory nature of the queries. MS-BASIC offers limited performance improvement over HV-ONLY, only 19%, or a $1.2\times$ speedup. This is due to the high cost of repeatedly transferring data between the stores for each query. Since the data transferred and loaded is not retained after a query completes, subsequent queries do not benefit from this effort. HV-OP shows a 59% improvement over HV-ONLY, representing a $2.4\times$ speedup. This improvement is attributable to opportunistic views retained during query processing in HV. MS-MISO has a 77% improvement over HV-ONLY, representing a speedup of $4.3\times$. MS-MISO also results in a 68% improvement over MS-BASIC ($3.1\times$), and a 44% improvement over HV-OP ($1.8\times$). These results show that MS-MISO is able to utilize both the opportunistic views and DW effectively to speed up query processing.

The improvement of MS-MISO over HV-OP and MS-BASIC can be attributed to the following two reasons. First, it is able to load the right views into the right store during its reorganization phases. The reorganization phases allow MS-MISO to tune both stores periodically as the workload changes dynamically. This is especially important for exploratory queries on big data, since the queries and the data they access are not known up-front. Second, by performing lightweight periodic reorganization, MS-MISO is able to benefit from DW's superior query processing power. Furthermore, these two factors are relevant even with small storage budgets. For example, when we repeat this experiment using B_h and B_d equal to $0.125\times$, MS-MISO still results in 59% improvement over HV-ONLY ($2.4\times$ speedup). Next we provide a breakdown of the performance of MS-MISO to offer insights into how the combination of these two factors results in better performance.

5.2.1 Breakdown of TTI

To further breakdown the previous results from Figure 4, Figure 5 reports the cumulative distribution plots (CDF) for (a) TTI, and (b) query execution time for the five system variants. Figure 5(a) shows TTI along the y-axis, and the number of queries completed on the x-axis. A point (x, y) in this figure indicates that x queries had a total TTI of no more than y seconds. DW-ONLY has the worst TTI due to its significant delay before any query can begin execution – the first query can only start after the ETL completes at 348,000 seconds. In contrast, HV-ONLY, HV-OP, MS-BASIC, and MS-MISO allow users to start querying right away, although with varying performance. MS-BASIC and HV-ONLY have similar performance, again indicating that MS-BASIC is not able to make effective use of DW because it only considers a single query at a time and does not perform any tuning. HV-OP has the second best TTI due to its effective use of opportunistic views. MS-MISO has the fastest TTI among all system variants while still offering the benefits of immediate querying.

Figure 5(b) shows the percent of queries with execution time less than 10, 100, 1,000, 2,000 and so on until 45,000 seconds. To gain insight on how the queries perform, we report query execution time only, which does not include any ETL time or tuning time. Clearly DW-ONLY has the best query performance (top-most curve) with 84% of queries completing in less than 100 seconds and all the queries finishing within 1,000 seconds. Furthermore, the DW-ONLY curve shows that nearly 65% of queries complete within 10 seconds, and 90% within 100 seconds. In contrast, HV-ONLY performs worst (bottom-most curve) with less than 3% of queries completing within 1,000 seconds. The systems near the bottom of the graph, HV-ONLY, MS-BASIC, and HV-OP, have no queries that execute in less than 100 seconds. The systems near the top of the graph, DW-ONLY and MS-MISO, complete at least 30% of their queries in less than 100 seconds. While it is not surprising that DW-ONLY outperforms MS-

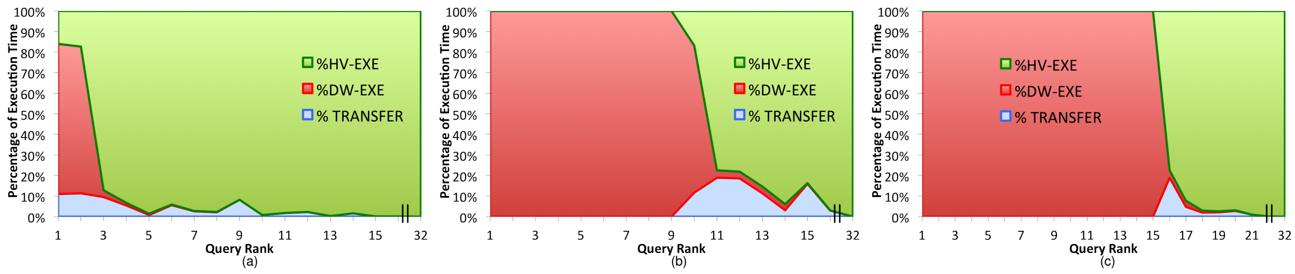


Figure 6: Breakdown of execution time into DW-EXE, HV-EXE and Transfer time components for queries from Figure 5(b); (a) MS-BASIC, (b) MS-MISO for $0.125\times$ storage budget, and (c) MS-MISO for $2\times$ storage budget.

MISO in terms of its *post-ETL* query performance, DW-ONLY is no longer competitive when ETL time is included. This shows that the lightweight adaptive tuning approach of MS-MISO is able to utilize the query processing power of DW without incurring the high up front cost of the ETL process.

5.2.2 *Breakdown of Store Utilization*

Next we highlight why MS-MISO performs better than MS-BASIC by examining the amount of time queries spend in each store. For each system in Figure 6, the y-axis indicates the fraction of time each query spends in HV, DW, or transferring data. For each system we rank the 32 queries by their DW utilization percent, and assign query rank 1 to the query that has the highest DW utilization, and rank 2 to the query with the next highest DW utilization; the x-axis indicates the query rank. Because the utilization trends stabilize after 15–20 queries, we truncate the x-axis in each figure.

Figure 6 shows this breakdown for three system variants – MS-BASIC in (a), MS-MISO with $0.125\times$ view storage budget in (b), and MS-MISO with $2\times$ view storage budget in (c). For MS-BASIC, only 2 queries spend the majority of their execution time in DW. In contrast, 9 of the queries in MS-MISO with the $0.125\times$ storage budget spend the majority of their total execution time in DW. This increases to 14 queries when the MS-MISO storage budget is increased to $2\times$.

Another way of quantifying DW utilization for each system is to consider the total execution time spent in HV versus DW for the first fastest queries in the workload, i.e., query ranks 1–16 in each figure. For queries ranked beyond 16, the HV-EXE component dominates most of a query's execution time, which does not illuminate DW utilization at all. With MS-BASIC in Figure 6(a), for every second spent in DW the queries spend 55 seconds in HV. With MS-MISO in Figure 6(b), for every 1 second spent in DW the queries only spend 1.6 seconds in HV. With MS-MISO in Figure 6(c), for every 1 second spent in DW the queries only spend 0.12 seconds in HV. This breakdown shows that MS-MISO is able to utilize DW more effectively than MS-BASIC, and increasing the storage budget for MS-MISO further increases DW utilization.

Finally, a different way to to examine store utilization is by reporting the ratio of plan operators executed by each store for each of the 32 queries. For MS-BASIC the best splits for all 32 queries were typically about 2/3 of the operators in HV and 1/3 in DW. For MS-MISO, an overview of the 32 splits is roughly as follows. For the fastest 9 queries, the split was 0/3 HV and 3/3 DW. For the next-fastest 20 queries, the average split was 1/3 HV and 2/3 DW. For the remaining 3 queries the average split was around 2/3 HV and 1/3 DW. Note that splits do not correlate exactly with plan execution times in Figure 6(b) due to the asymmetric performance of the stores.

5.3 Tuning Algorithm Comparisons

In this section we evaluate MS-MISO against four other possible tuning techniques for multistore physical design.

- MS-BASIC: Performs no tuning (i.e., does not use views).

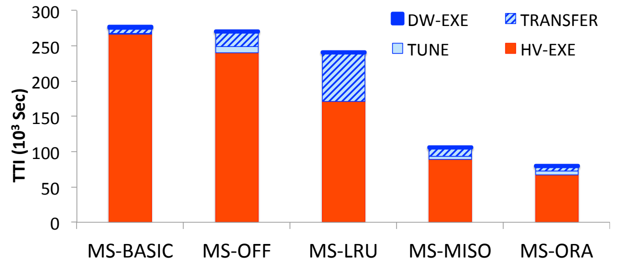


Figure 7: TTI Comparison of multistore tuning techniques.

- MS-OFF: Performs offline tuning, where it is given the entire workload up-front and tunes the stores one time by choosing the most useful set of views for the workload. This approach represents what is possible with current design tools and is provided only as a point of reference to show the performance trends.
- MS-LRU: Performs "passive" tuning by retaining working sets transferred between the stores during query execution as "views." LRU and its many variants are access-based approaches and provide a simple way of deciding which views to retain within a limited storage budget.
- MS-MISO: Performs online tuning using our MISO Tuner.
- MS-ORA: Performs online tuning using our MISO Tuner also, but the *actual* future workload is provided rather than predicted. This technique ("oracle") is just provided as a point of reference as the best performance possible for the MISO Tuner.

Tuning parameters for MS-OFF, MS-LRU, MS-MISO, and MS-ORA are set to $B_h = 0.125\times$, $B_d = 0.125\times$, and $B_t = 10$ GB since these budgets represent a more constrained environment.

Figure 7 compares the performance of the tuning techniques described above. Among all techniques, MS-BASIC performs the worst. This shows that multistore query processing without tuning does not perform as well as any of the other tuning techniques. Among the techniques that perform tuning, MS-OFF has the worst performance because the small storage budgets are insufficient to store all beneficial views for the entire workload. MS-MISO provides a 60% improvement over MS-OFF since it is able to adapt the physical design to the changing workload. Furthermore, MS-MISO results in a 56% improvement over MS-LRU because MS-LRU does not explicitly consider view benefits or interactions. This is because MS-LRU is an access-based approach whereas MS-MISO is a cost-benefit based approach that considers view benefits and interactions. Notice there is a large transfer time for MS-LRU, indicating that this method does not do a good job of retaining beneficial views. Finally, MS-MISO is 32% worse than MS-ORA, since the actual future workload is unknown to MS-MISO.

5.3.1 *Varying the Tuning Parameters*

Next, we analyze the performance of MS-MISO, MS-OFF, and MS-LRU under varying tuning constraints. Figure 8 shows the performance of MS-OFF, MS-LRU, and MS-MISO as we vary the storage

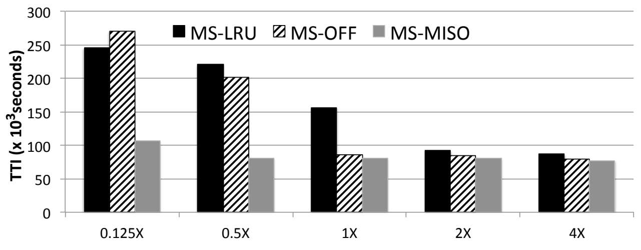


Figure 8: Varying the view storage budgets B_h and B_d, while transfer budget B_t is held constant at 10GB.

budget parameters B_h and B_d from $0.125\times$ to $4.0\times$. B_t was kept fixed at 10 GB. The y-axis reports TTI, and the x-axis shows the storage budgets on each store.

MS-MISO performs the best for all storage budgets, including the larger storage budgets. Among the smaller budgets of $0.125\times$ and $0.5\times$, both MS-OFF and MS-LRU perform significantly worse than MS-MISO. However, the performance of MS-OFF and MS-LRU improves with increasing storage budgets. For example, at $1\times$ budget, MS-MISO is 50% better than MS-LRU but only 7% better than MS-OFF, while with larger budgets the performance of all three methods begins to converge. The tuning techniques begin to have similar performance with larger storage budgets (2–4$\times$) since there is plenty of available storage space to retain many views in support of the recent workload.

5.4 Utilizing DW with Limited Spare Capacity

Now that we have shown the basic performance tradeoffs with MS-MISO, we evaluate MS-MISO in a realistic scenario with a DW that has limited spare capacity – that is, a DW that is executing a workload of reporting queries. As DW is a tightly-controlled resource, it is important to understand how our method affects the DW. To do this, we measure the impact of the multistore query workload on the DW reporting queries as well as the impact of the DW queries on the multistore query workload. These experiments are important because they model a realistic deployment scenario for a multistore System where an organization has two classes of stores and may want to utilize their existing DW resources to speed up the big data queries in HV.

To model a running DW, we consider varying amounts of spare IO and CPU capacity by executing a reporting query workload in DW. We examine four cases: A DW with 20% spare IO capacity, 20% spare CPU capacity, 40% spare IO capacity, and 40% spare CPU capacity. The spare capacities are measured as the amount of unused resources as reported by Linux IOstat; for example, 60% CPU consumption indicates a 40% spare CPU capacity. We measure these values for each of the 9 machines in the DW cluster every 10 seconds, and report the average value. In this scenario, we execute a background DW workload on the DW cluster that consumes a fixed proportion of IO or CPU resources. Below, we present the results for one case: a DW with 40% spare IO capacity. The remaining three cases showed very similar trends and hence are omitted. However, Table 2 below summarizes all of the results.

The multistore query workload is then executed as a single stream of 32 queries. As the multistore query workload is executed, we measure its impact on the background DW queries by the amount of slowdown of the background queries, representing the impact that the multistore workload has on the reporting queries running in DW. Conversely, we also measure the impact of the DW queries on the multistore workload.

To create a DW environment with limited IO or CPU resources, we use a single TPC-DS query and run multiple parameterized versions of the query in parallel in order to consume a fixed amount of each resource. The queries are used to control the amount of spare IO or CPU capacity. To do this, we first load a 1TB scale TPC-DS dataset into DW. We then selected 2 queries, query templates q3 and q83, since query q3 is IO intensive and query q83 is CPU intensive. To obtain 40% spare IO capacity in DW, we continuously execute one instance of q3 (which consumes 60% of the IO resources). To obtain 20% spare IO capacity, we continuously execute three instances of q3. Similarly, to obtain 40% spare CPU capacity in DW, we continuously execute two instances of q83 (which consumes 60% of the CPU resources). To obtain 20% spare CPU capacity, we continuously execute three instances of q83. All views transferred during multistore query execution are kept in a temporary table space, whereas views transferred by MS-MISO during reorganization are stored in permanent table space.

Figure 9(a) reports the CPU and IO load for the DW cluster with 40% spare IO capacity. The x-axis corresponds to time steps during experiment, and the y-axis indicates CPU and IO resource consumption. Initially, the IO is stable at 60% and the CPU consumption is at about 20%, then the multistore queries begin executing. Multistore queries may spend a lot of their execution time in HV, during which time DW continues executing only its background workload.

The peaks in Figure 9(a) (e.g., marked R, T) correspond to periods of either view transfers during reorganization (R) by MS-MISO or working set transfers during query processing (T), while the flatter areas (e.g., marked Q) correspond to periods of multistore query execution. For both R and T cases, the data transfers put heavy demands on the IO resources, in some instances consuming 100% of the IO resources. In Figure 9(a), view transfers during (R) and (T) phases consume a lot of the available spare capacity of DW, while multistore query execution (Q) has little impact on the spare capacity. This results in brief instances of high resource impact, but long time periods with low impact. Next, we show how the resource consumption by the multistore queries impacts the execution of the running DW queries.

Figure 9(b) shows the average execution time of the DW background queries as Multistore queries are executed for the same experiment. The x-axis again reports the timesteps during the experiment and the y-axis reports the average execution time of q3 during the entire experiment. The average execution time of q3 when no Multistore queries are executing is 1.06 seconds. Over the course of the entire experiment, the average execution time of q3 increased to 1.09 seconds, representing an overall slowdown of 2.5% for q3. The peaks in this figure correspond similarly to working set transfers during query execution (T), or view transfers during reorganization (R). This figure shows that these events briefly impacting the average execution time of q3, which increases to over 5 seconds several times. The regions Q again correspond to Multistore query execution in DW. This figure shows that the impact on the average execution time of DW background queries is very small in spite of brief periods of high impact.

Table 2: Impact of multistore workload on DW queries and vice-versa.

DW Spare Capacity		Percent Slowdown of DW Queries	Percent Slowdown of Multistore Workload
IO	40%	1.1%	2.5%
	20%	1.7%	4.0%
CPU	40%	0.3%	4.2%
	20%	0.8%	5.0%

In order to quantify the impact of an active DW versus an empty DW, Table 2 reports the impact of the multistore workload and the DW queries on each other. For the DW queries, we report the slowdown in their average execution time with versus without the multistore queries running. For the multistore queries, we report the slowdown in the TTI with and without the DW queries executing. The slowdown of the DW queries due to the Multistore queries less than 2%. Similarly, the slowdown of the multistore workload is no more than 5% versus an empty DW. This shows that the impact of the workloads on each other is actually quite minimal, indicating MS-MISO is indeed beneficial when there is limited spare capacity in DW.

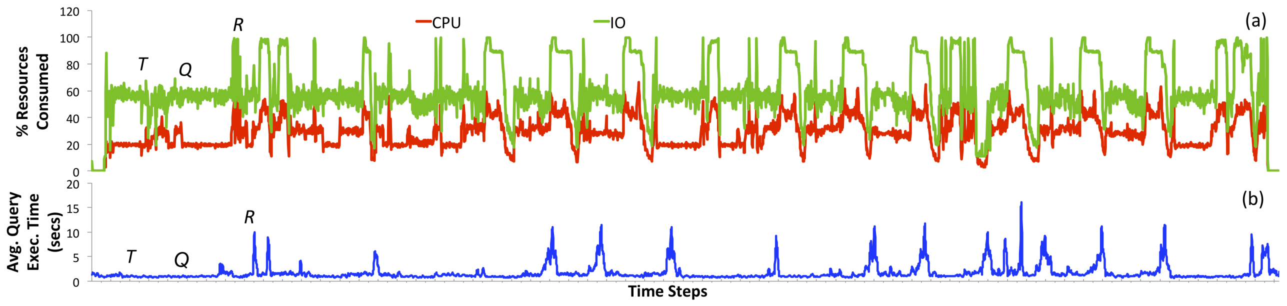


Figure 9: Impact of multistore workload on DW with 40% spare IO capacity showing (a) IO/CPU resources consumed and (b) average query execution time of the DW queries.

6. DISCUSSION AND CONCLUSION

The previous work on multistore systems has considered optimizing queries over data sets that span multiple stores. In this work we showed how a DW can be used as an accelerator for big data queries. Our main insight was that tuning the multistore system's physical design is key to achieving good performance. For a multistore system to be effective, it should use both stores toward their strengths: HV is best used for sifting through massive volumes of big data and extracting a small yet relevant subset of the data; DW is best used for processing this small subset of data efficiently since it can be loaded quickly with a short-term impact on the DW queries. In this paper, we argued that identifying this changing subset and determining where to store it is at the core of the concept of multistore systems. Our lightweight adaptive tuning method realizes the benefits of multistore processing by leveraging the strengths of each store to ensure that the current relevant subset of data (i.e., opportunistic views) is present in the "best" store based on the observed workload which is changing dynamically.

In this paper, we kept the transfer budget, B_t, small since a larger value has several undesirable consequences: a larger impact to DW during reorganization phases (slowing down DW queries), tuning consumes a large portion of TTI, and each reorganization moves more data across the network. However, there is a trade-off between how frequently one reorganizes the system versus the size of B_t. For instance, a small transfer budget value would allow more frequent reorganizations, each with a short-term impact on DW. An appropriate setting for B_t is dependent upon the nature of the workload and choosing a balance between the desired speed up for exploratory queries with the tolerable impact on DW reporting queries.

Our multistore approach combines the scalability of HV with the query performance of DW to leverage their unique strengths. There are several emerging big data systems such as Impala, Spark, Asterix, Stinger, and Hadoop 2.0 that make interesting design trade-offs to deliver good performance. An interesting area of future work is to explore ways of combining these emerging systems into multistores in a way that leverages their different strengths.

Although we do not address updates in this work, for future work there are several interesting considerations for maintaining opportunistic views; primarily due to the nature of the query domain (i.e., exploratory analysis), the way views are created (i.e., opportunistically), and the append-only nature of the updates in HDFS.

Acknowledgement: We thank Jeffrey Naughton and Hector Garcia-Molina for their insightful comments on the earlier versions of the paper. We also thank the anonymous SIGMOD reviewers for their useful feedback. J. LeFevre and N. Polyzotis were partially supported by NSF grant IIS-0447966 and a research grant from NEC labs.

7. REFERENCES

[1] A. Abouzeid, D. J. Abadi, and A. Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. In *EDBT*, 2013.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1), 2009.

[3] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *NSDI*, 2012.

[4] Apache Sqoop. http://sqoop.apache.org/, 2013.

[5] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.

[6] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *TKDE*, 16(11), 2004.

[7] S. Chaudhuri and V. Narasayya. An efficient, cost-driven index selection tool for Microsoft SQL Server. In *VLDB*, 1997.

[8] M. P. Consens, K. Ioannidou, J. LeFevre, and N. Polyzotis. Divergent physical design tuning for replicated databases. In *SIGMOD*, 2012.

[9] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in Polybase. In *SIGMOD*, 2013.

[10] I. Elghandour and A. Aboulnaga. ReStore: Reusing results of MapReduce jobs. *PVLDB*, 5(6), 2012.

[11] H. Hacigümüs, J. Sankaranarayanan, J. Tatemura, J. LeFevre, and N. Polyzotis. Odyssey: A multi-store system for evolutionary analytics. *PVLDB*, 6(11), 2013.

[12] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[13] S. LaValle, E. Lesser, R. Shockley, M. Hopkins, and N. Kruschwitz. Big data, analytics and the path from insights to value. *MIT Sloan Management Review*, 52(2), 2011.

[14] J. LeFevre, J. Sankaranarayanan, H. Hacıgümüş, J. Tatemura, and N. Polyzotis. Towards a workload for evolutionary analytics. In *SIGMOD Workshop on Data Analytics in the Cloud (DanaC)*, 2013. Extended version *CoRR abs/1304.1838*.

[15] J. LeFevre, J. Sankaranarayanan, H. Hacıgümüş, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic physical design for big data analytics. In *SIGMOD*, 2014.

[16] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: sharing across multiple queries in MapReduce. *PVLDB*, 3(1–2), 2010.

[17] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

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

[19] K. Schnaitter and N. Polyzotis. Semi-automatic index tuning: keeping DBAs in the loop. *PVLDB*, 5(5), 2012.

[20] K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: modeling, analysis, and applications. *PVLDB*, 2(1), 2009.

[21] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. QoX-driven ETL design: Reducing the cost of ETL consulting engagements. In *SIGMOD*, 2009.

[22] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *SIGMOD*, 2012.

[23] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *TODS*, 35(1), 2010.

[24] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

[25] Y. Xu, P. Kostamaa, and L. Gao. Integrating Hadoop and parallel DBMS. In *SIGMOD*, 2010.