Towards an Adaptive Multidimensional Partitioning for Accelerating Spark SQL

Soumia Benkrid[1], Ladjel Bellatreche[2], Yacine Mestoui[1], and Carlos Ordonez[3]

[1] Ecole nationale Supérieure d'Informatique (ESI)
s_benkrid@esi.dz,fy_mestoui@esi.dz
[2] LIAS/ISAE-ENSMA, Poitiers, France
bellatreche@ensma.fr
[3] University of Houston, Texas, USA
carlos@central.uh.edu

Abstract. Nowadays Parallel DBMSs and Spark SQL compete with each other to query Big Data. Parallel DBMSs feature extensive experience embodied by powerful data partitioning and data allocation algorithms, but they suffer when handling dynamic changes in query workload. On the other hand, Spark SQL has become a solution to process query workloads on big data, outside the DBMS realm. Unfortunately, Spark SQL incurs into significant random disk I/O cost, because there is no correlation detected between Spark jobs and data blocks read from the disk. In consequence, Spark fails at providing high performance in a dynamic analytic environment. To solve such limitation, we propose an adaptive query-aware framework for partitioning big data tables for query processing, based on a genetic optimization problem formulation. Our approach intensively rewrites queries by exploiting different dimension hierarchies that may exist among dimension attributes, skipping irrelevant data to improve I/O performance. We present an experimental validation on a Spark SQL parallel cluster, showing promising results.

Keywords: Multidimensional Partitioning, Spark-SQL, Utility maximization, Adaptive Query Processing, Big data, dimensional hierarchies.

1 Introduction

In the last decades, the digital revolution is evolving at an extreme rate, enabling faster changes and processes, and producing a vast amount of data. In this context, companies are turning urgently to data science to explore, integrate, model, evaluate and automate processes that generate value from data. Beyond actual tools and data details, massive parallel processing is required for providing the scalability that is necessary for any big data application. Indeed, to result in linear speed-up, the DBA may often increase the resource in the cluster by adding more machines to the computing cluster. Furthermore, this requires additional costs which may be not available in practice.

A promising parallel processing approach requires three main phases: (1) data partitioning and allocating partitions, (2) running processing code on each partition, and (3) gathering or assembling partial results. However, the efficiency of the database parallel processing is significantly sensitive to how the data is partitioned (phase 1). Basically, the partitioning scheme is chosen in a static (offline) environment where the design is done only once and that design can persist. Most commonly, when the change is detected, the DBA must often intervene by taking the entire DW offline for repair. The problem is getting bigger and more difficult because the data system has become large and dynamic. Thus, carrying out a redesign at each change is not entirely realistic since the overload of the redesign is likely to be very high. Figure 1 illustrates the time needed to adapt a fragmentation schema among the size of the database. Accordingly, making an adaptive partitioning schema with minimal overload is a crucial issue.

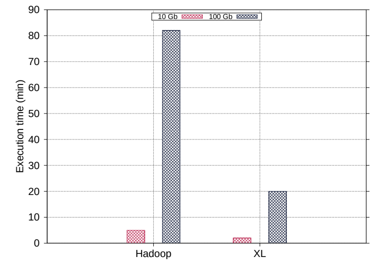

Fig. 1: Fragmentation Schema Adaptation according the database size

The adaptive issue has been extensively studied by the database community. The first line of works [22, 20, 4, 8, 10] focuses on changes in the workload, periodically or when a change occurs, an incremental design is performed. These workload-driven solutions yield promising results, but they require significant time to adapt a fragmentation schema. The second category of works uses Query Adaptive Processing [1, 16, 14, 24, 12] that use runtime statistics to choose the most efficient query execution plan during query execution. However, these Data-driven approaches can incur in non-negligible overhead.

As though HDFS stores the data by creating a folder with partition column values, altering the partition schema means the change of the whole directory structure and data. To overcome the problem, Spark SQL a big data processing tool for structured data query and analysis tailored towards using a Data-driven approach that uses runtime statistics to choose the most efficient query execution plan during query execution. Hence, the Spark SQL system is currently leading to serious random disk I/O costs since there is no correlation between the spark jobs and the data to be read from the disk.

In this paper, we introduced a new dynamic SQL framework for Apache Spark. Our novel approach relies on the intuition that multi-dimensional partitioning could accelerate the spark SQL since thereby it supported fewer I/O operations and effective prefetching. Our SQL framework is based on our Genetic Optimization Physical Planner [6] and relies on the intuition that a query should leverage knowledge available in the reference partitioning schema to Spark as much as possible. Specifically, our SQL optimizer (1) uses reference partitioning to enable efficient data processing by avoiding unnecessary read; (2) leverages dimensional hierarchies' information to maximize the benefit of the partitioning schema. The goal of our work is to show that by combining efficient workload-driven approaches and adaptive execution processing, the performance of Spark SQL can be significantly increased.

The remainder of this paper is organized as follows. Related works are discussed in Section 2. Section 3 provides the formulation of our studied problem. Section 4 introduces a novel planner for data partitioning based on genetic optimization. Section 5 presents experiments evaluating the quality of our solution. Finally, Section 6 summarizes our main findings.

2 Literature Review

Data partitioning has attracted plenty of research attention. Before 2013, researchers typically assume that the query workload is provided upfront [21, 1, 25, 7, 18, 8] and try to choose the best partitioning schema in offline (static) mode using mainly some optimization search techniques such as branch-and-bound search [25, 18], genetic algorithms [18], and relaxation/approximation [7]. Nevertheless, it is difficult to maintain the usefulness of the offline data partitioning schema in a dynamic context. In fact, there have been two main directions of research, depending on how to select data to be migrated. We can further distinguish between two different categories. 1) Workload driven techniques, which focus on the workload changes and 2) data driven techniques, which work with the optimizer.
Workload-driven adaptive approaches. The most recent works focus on studying how to migrate data based on the workload change [19], and the data items that are affected [22, 20, 4]. E-Store [22] is a dynamic partitioning manager for automatically identifying when a reconfiguration is needed using system and database statistics. It explores the idea of managing hot tuples separately from cold tuples by designing a two-tier partitioning method that first distributes hot tuples across the cluster and then allocates cold tuples to fill the remaining space. Simultaneously, the research community has sought to develop additional design techniques based on machine learning, such as view recommendation [11], database cracking [3], workload forecasting [17], cardinality estimation [13] and Autonomous DBMS [2, 23, 5], by which multiple aspects of self-adaptive can be improved. However, only a few works have recently focused to tune data-partitioning by using reinforcement learning (RL) [8] and deep RL [10]. Hilprecht et al. [10] propose an approach, based on Deep Reinforcement Learning (DRL), in which several DRL agents are trained offline to learn the trade-offs of using different partitioning for a given database schema. For that, they use a cost model to bootstrap the DRL model. If new queries are added to the workload or if the database schema changes, the partitioning agent is adapted by progressive learning. However, to support a completely new database schema, a new set of

DRL agents must be trained.
Data-driven adaptive approaches. The data driven adaptive approaches consist of improve the execution performance by generating a new execution plan at runtime if data distributions do not match the optimizer's estimations. Kocsis et al. [14] proposed HYLAS, a tool for optimizing Spark queries using semantic preserving transformation rules to eliminate intermediate superfluous data structures. Zhang et al. [24] propose a system that dynamically generates execution plans at query runtime, and runs those plans on chunks of data. Based on feedback from earlier chunks, alternative plans might be used for later chunks. DynO [12] and Rios [15] focus on adaptive implement by updating the query plan at runtime.
Recently, we have proposed a two-step approach based on genetic algorithms to improve the performance of dynamical analytical queries by optimizing data partitioning in a cluster environment[6]. Although this effort put on the query optimizer, it is still in the infant stage for Spark SQL optimizer. This is mainly due to the Parquet storage format, the HDFS block size is much larger, indexes and buffer pool. In this paper, we characterized the effectiveness of the query optimization in the aim of enhancing Spark SQL optimizer with detailed partitioning information. Indeed, our solution aims to filter out most of the records in advance which can reduce the amount of data in the shuffle stage and improve the performance of equivalent connections.

3 Definitions and Problem Formulation

In this section, we present all ingredients that facilitate the formalization of our target problem.

3.1 Preliminaries

Range Referencing Partitioning. In this paper, we reproduce the traditional methodology to partition relational DW to HDFS DW. More concretely, we *partition some/all dimension tables using the predicates of the workload defined on their attributes, and then partition the fact table based on the partitioning schemes of dimension tables.* To illustrate this partitioning, let us suppose a relational warehouse modelled by a star schema with d dimension tables $(D_1, D_2, ..., D_d)$ and a fact table F. F is the largest table, used on every BI query. Among these dimension tables, g tables are fragmented $(g \leq d)$. Each dimension table D_i $(1 \leq i \leq g)$ is partitioned into m_i fragments: $\{D_{i1}, D_{i2}, ..., D_{im_i}\}$, where each fragment D_{ij} is defined as: $D_{ij} = \sigma_{cl^i_j}(D_i)$, where cl^i_j and σ $(1 \leq i \leq g, 1 \leq j \leq m_i)$ represent a conjunction of simple predicates and the selection operator, respectively. Thus, the fragmentation schema of the fact table F is defined as follows: $F_i = F \ltimes D_{1j} \ltimes D_{2k} \ltimes .. \ltimes D_{gl}$, $(1 \leq i \leq m_i)$, where $\ltimes$ represents the semi join operation.

Benefit of a partitioning schema for a query. Using a partitioning schema to answer a query has a significant benefit since it can reduce overall disk performance. Data partitioning decomposes very large tables into smaller partitions, and each partition is an independent object with its own name and its own storage characteristics. The benefit can be calculated by the difference of query cost with/without using the partitioning schema. Which is defined as below.

DEFINITION 1 *Benefit : Given a query q and a partitioning schema SF, the cost of executing* q *is* $cost(q)$*, the cost of executing* q *using* SF *is* $cost(q|SF)$*, and the benefit is* $B_{q,SF} = cost(q) - cost(q|SF)$*. A fragemtation schema should ensure a positive benefit (*$B_{q,SF} > 0$*)*

Utility of Fragmentation Schema. Given a query q and a partitioning schema SF, we need to compute the utility of using SF for q. The utility of a partitioning schema SF seeks to maximize the benefit of a query, generally, over two periods t and $t + 1$. The utility of a partitioning schema SF over a query q is a metric that measures the reduction threshold in the cost of q by SF. Thus, we define $U(SF, q)$ as follows.

DEFINITION 2. *We suppose that if the period* $t + 1$ *transmits a new query* q*, its utility is given by:*

$$U(SF, q) = B_{q,SF'}(q, t + 1)/B_{a,SF}(q, t) \tag{1}$$

where $B_{q,SF'}(q, t + 1)$ $(\forall i \in \{0, 1\})$ *denotes the benefit of* q *under the partitioning schema* SF *define at time* $t + i$ *and* SF' *is the partitioning schema generated after the consideration of* q *in the predefined workload.*

3.2 Problem Statement

We now describe the formulation of our problem: Given:

- A SPARK-SQL cluster $\mathcal{DBC}$ with M nodes $\mathcal{N} = \{N_1, N_2, \ldots, N_M\}$;
- A relational data warehouse $\mathcal{RDW}$ modeled according to a star schema and composed by one fact table $\mathcal{F}$ and d dimensional tables $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$.
- a set of star join queries $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_L\}$ to be executed over $\mathcal{DBC}$, being each query Q_l characterized by an access frequency f_l;
- A fragmentation maintenance constraint W that the designer considers relevant for his/her target partitioning process
- A target profit u which represents the minimum desired utility.

The optimization problem involves finding the best fragmentation schema SF^* such that

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{L} B_{Q_i, SF^\star} \\ \text{subject to} \quad & |SF^\star| \leq \mathcal{W}, \\ & \sum_{i \geq 1} \mathcal{U}(SF^\star, Q_i) \geq u. \end{aligned} \tag{2}$$

4 System Architecture

Figure 2 illustrates the global architecture of our framework. It contains two main parts. *Partitioning schema selection* and *controller*. First, given a query workload, we determine the best data partitioning schema. Next, the controller interacts with Spark SQL to rewrite queries, launch new configurations, and collects performance measurements. Precisely, the decision-adaptive activity can be regarded as an analysis form that it is trying to choose which adaptation strategy should be started at the current time to maximize the overall utility that the system will provide during the remainder of its execution. Later, we rewrite queries using some highly beneficial of the partitioning schema. As last step, we execute the rewritten workload. Next, we discuss the details these two parts.

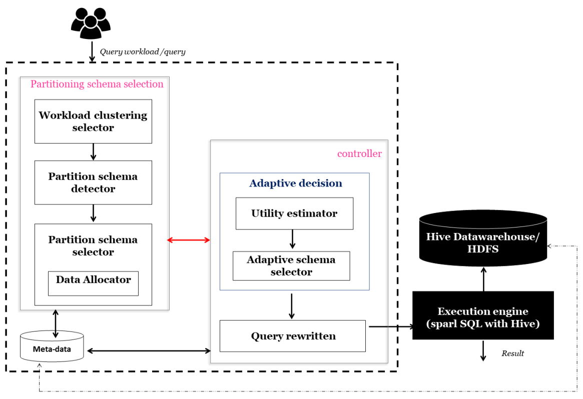

Fig. 2: The overview of system framework.

4.1 Partitioning schema selection

The main idea of our partitioning approach, is to build several data partitioning schemes and merge them together to get a more accurate (having high utility) and dynamic global data partitioning schema. Precisely, once the data partitioning schema for each sub-workload is selected, an algorithm is used to aggregate over the data partitioning schemes to form the most efficient data partitioning.

This part contains three components: *workload clustering selector*, *Partitioning Schema detector* and *partitioning schema selector*. At first, we use *workload clustering selector* to divide the given workload to a number of sub-workloads such that queries in the same groups are related to one another. Later, we use *partitioning schema builder* to build a fragmentation schema for each query group. Once the partitioning schemes of the workload clusters have been generated, their outputs must be combined into a single partitioning schema. The *Partitioning Schema selector* merges them together to get a more accurate and stable data partitioning schema. In particular, we use our genetic planner [6]. Finally, the data allocator place the so-generate fragments over Spark-SQL cluster using hash distribution.

4.2 Controller

This part contains two components: *utility estimator* and *adaptive schema selector*. Every query issued by users first goes to the *utility estimator* to compute the utility $U(SF,q)$. For that, we design a cost model to calculate the cost of executing a query on a given fragmentation schema. First, *utility estimator* calls *partitioning schema builder* in order to build un fragmentation schema by considering the new query. After that, we can compute the benefit $B_{q,SF}$ and the utility $U(SF,q)$.

When a violation of performance constraints occurs, the new queries causing the failure are integrated into the fragmentation scheme. Our genetic algorithm is called again to select the best partitioning plan. First, a starting population of adaptation strategies is created based on the last exploration phase population of the genetic algorithm, as well as adaptation strategies for the updated offline classes with ad-hoc queries. Then these adaptation strategies are iteratively improved by applying a combination of mutation and crossover operators, with the most efficient plans being more likely to pass to the next generation. It is expected that this iterative process increases utility over time, thereby reducing average query evaluation. We emphasize that by reusing exploration candidates, we reduce the number of evaluations of the fitness function to choose the best solution.

4.3 Query rewritten

The *query rewritten* component will extract the dimensions and predicates from the query, and then search the partitioning attribute for any predicate which can provide the data source needed by the query. Upon finding a qualified partitioning attribute, the engine will add/replace the original predicate with the new predicate and then use Spark SQL to execute the query.

In order to reduce the reading of the data, we not only use the fragmentation attributes, but we also use the hierarchy dimension structure [9, 21]. We can distinguish the following two main cases for which such an improvement is possible.

Case 1: Queries on "lower-level" attributes of the fragmentation dimension. In fact, each value of an attribute belongs to a low level in the hierarchy corresponds exactly to one value of the fragmentation attribute. If the query references all fragmented dimensions, it requires loading a single fragment to run, otherwise multiple fragments will be loaded. Thus, a rewrite is necessary by adding predicates on the partitioning attributes to load only the valid tuples needed.

Case 2: Queries on "higher-level" attributes of the fragmentation dimension.

Queries defined on attributes belonging to a high level in the fragmentation attribute hierarchy also benefit from the fragmentation scheme. This is because the number of fragments to load will be greater than the previous cases, since each value of the attribute has several associated values of the fragmentation attribute. Thus, the number of fragments increases if, and only if, certain dimensions of fragmentation are involved.

Obviously, all queries defined on fragmented dimensions will also benefit from the partitioning schema, as in the case of *Case 1* and *Case 2*. Thus, all queries referencing at least one attribute of a fragmented dimension table benefit from fragmentation by reducing the number of fragments to be processed.

5 Experimental Results

Our experiments were conducted on a Hadoop parallel cluster with 9 nodes, configured as 1 HDFS NameNode (master node) and 8 HDFS DataNodes (workers), the nodes were connected through 100 Mbps Ethernet. All cluster nodes had Linux Ubuntu 16.04.1 LTS as the operating system. On top of that, we installed Hadoop version 3.1.0, which provides HDFS, and Yarn, and Hive 2.0.0, with MySQL database (MySQL 5.5.53) for Hive metadata storage. Separately, we installed Spark 3.1.0 and we configured Spark SQL to work with Yarn and Hive catalog. The overall storage architecture is based on HDFS. First, we load the large amount of data on Hive tables in Parquet columnar format. Then, in the interrogation step, we use Spark SQL to read the Hive partitioned tables We use the well-known Star Schema Benchmark (SSB) to generate the test data set. The size of the data set used is $100G$.

In the first experiment, we focused the attention on the impact of fragmentation threshold. The aim of this experiment is to show the importance of choosing the best number of fragment to improve the workload performance. For that, we vary the fragmentation threshold in the interval $[100-600]$ and we calculate the throughput workload execution.

It clearly follows that since the value of W is smaller than 300, increasing of the fragmentation threshold improves the query performance because by releasing the fragmentation threshold, more attributes are used to fragment the warehouse. Although when the value of W is relatively large that decreases the query performance significantly, as HDFS is not appropriate for small data and lacks the ability to efficiently support the random reading of small files because

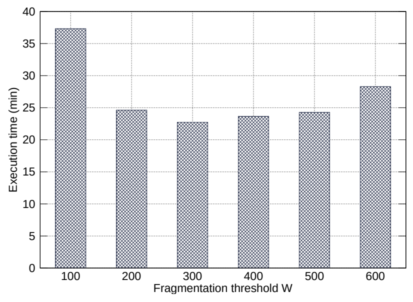

Fig. 3: Effect of the Fragmentation Threshold W on the Query Performance.

of its high capacity design. This experimental result confirms the importance of choosing the right number of final fragments to generate and online repartitioning may defer further degrade performance.

In the rest of the experiments, we fix the fragmentation threshold (W) to 200. We have chosen to fix W to 200 and not to 300 and this to have a flexibility for the adaptation/repartitionning step.

In the second experiment, we outlined the impact of the using of dimensional hierarchies on query processing. Figure 4 shows the results obtained and confirms that the use of the dimensional hierarchies improves significantly the performance as most of the effective characteristics of queries are taken into consideration. Precisely, our fine-grained rewritten allows many query to be confined too few fragments, thereby reducing I/O. Our Extensive rewritten outperforms Spark SQL optimizer by average 45% in terms of execution time.

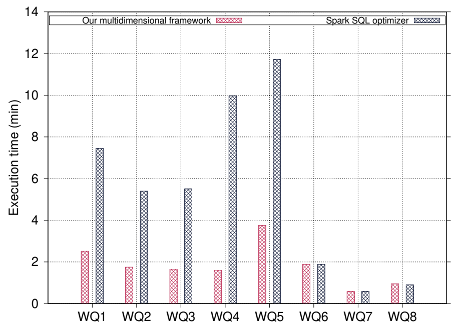

Fig. 4: Impact of using dimensional hierarchies.

To examine the quality of our genetic based approach, we compared our genetic based approach with two others partitioning scheme aggregation approaches by running 100 queries.

- *Majority Voting*. Rank sub-domains by the number of times they appear in different partitioning schema
- *SUKP-Partionning*. Compute the utility of each sub-domains for each query, and rank sub-domains by their maximum utility for the workload. This is similar to the approach followed by [5].

As shown in 5 (a), the genetic approach outperforms the others two approaches. Through the genetic operators (crossover and mutation) the approach gives rise to better fragmentation patterns that promote the majority of queries.

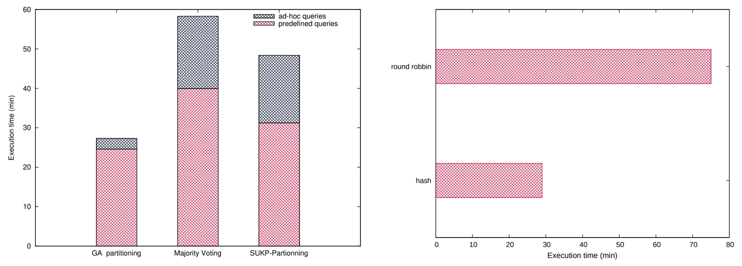

Fig. 5: Impact of the GA on the Query Performance.

Placing partitions on nodes is essential to achieve good scalability. In order to identify the best allocation mode, we compare the round robbin manner with the hash placement. As depicted in figure 5 (b) the hash placement is an efficient way to distribute data since the data of each partition can reside on all nodes. This ensures that data is distributed evenly across a sufficient number of data nodes to maximize I/O throughput.

Finally, in the fourth experiment, we focus our study on determining the minimum threshold of utility for selecting the best adaptive partitioning. To do this, we set the fragmentation threshold W at 200 ($W = 200$) and we varied the utility threshold u from 40 to 80. For each value, we select a fragmentation schema for each value of u, and we calculate the execution time of the 100 predefined workload and 10 ad-hoc queries. Then, we calculate the utility of the fragmentation schema according to the obtained execution time and the best fragmentation schema of each group of queries.

As depicted in Figure 6, too high or too low utility threshold can significantly impact the performance of the predefined and ad-hoc queries. Precisely, a high threshold may result in not obtaining a high utility fragmentation scheme for ad-hoc queries, but it is sufficiently useful for the predefined workload. Though a low threshold may result in a less meaningful schema for the predefined queries and the schema is perfectly adequate for ad-hoc queries. In addition to this, it should be noted that it is important of carefully choosing the utility desired rate (u). The administrator must carefully choose the best utility threshold according to the frequency of change in the workload.

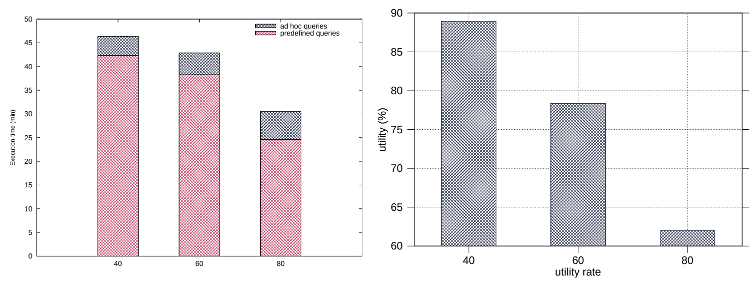

Fig. 6: Impact of the Utility Threshold on Query Performance.

6 Conclusions

In this article, we have presented an external adaptive and workload-sensitive partitioning framework for large-scale data. In this framework, we have addressed several performance issues; these include the limitations of static workload-aware partitioning, the overhead of rebuilding partitions in HDFS, and also irrelevant data spark-SQL reads. We have shown that utility-based formalization and the exploitation of partitioning attribute semantics (dimensional hierarchies) successfully address these challenges and provides an efficient Spark-SQL query processing framework. Extensive experiments evaluate partition quality, the impact of parameters, and dimensional hierarchies rewritten efficiency, with encouraging results. We believe that leverages multi-dimensional partitioning with Spark Adaptive Query processing shows promise to analyze dynamic workloads with ad-hoc queries in modern big data environments.

Although the obtained results are interesting and encouraging, this work opens several perspectives: (i) improvement of the adaption policy by using machine learning techniques, (ii) identifying automatically the best Spark SQL configuration (dynamic cluster).

References

1. F. Akal, K. Böhm, and H.-J. Schek. Olap query evaluation in a database cluster: A performance study on intra-query parallelism. In *ADBIS*, pages 218–231, 2002.
2. D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *ACM SIGMOD*, pages 1009–1024, 2017.
3. I. Alagiannis, S. Idreos, and A. Ailamaki. H2o: a hands-free adaptive store. In *ACM SIGMOD*, pages 1103–1114, 2014.
4. O. Asad and B. Kemme. Adaptcache: Adaptive data partitioning and migration for distributed object caches. In *Proceedings of the 17th International Middleware Conference*, pages 1–13, 2016.
5. S. Benkrid and L. Bellatreche. A framework for designing autonomous parallel data warehouses. In *ICA3PP*, pages 97–104, 2019.

6. S. Benkrid, Y. Mestoui, L. Bellatreche, and C. Ordonez. A genetic optimization physical planner for big data warehouses. In *IEEE Big Data*, pages 406–412, 2020.
7. N. Bruno and S. Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *ACM SIGMOD*, pages 227–238, 2005.
8. G. C. Durand, M. Pinnecke, R. Piriyev, M. Mohsen, D. Broneske, G. Saake, M. S. Sekeran, F. Rodriguez, and L. Balami. Gridformation: Towards self-driven online data partitioning using reinforcement learning. In *aiDM Workshop*, pages 1–7, 2018.
9. C. Garcia-Alvarado and C. Ordonez. Query processing on cubes mapped from ontologies to dimension hierarchies. In *Proceedings of the fifteenth international workshop on Data warehousing and OLAP*, pages 57–64, 2012.
10. B. Hilprecht, C. Binnig, and U. Röhm. Towards learning a partitioning advisor with deep reinforcement learning. In *aiDM Workshop*, pages 1–4, 2019.
11. A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *Proc. VLDB Endow.*, 11(7):800–812, Mar. 2018.
12. K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically optimizing queries over large scale data platforms. In *ACM SIGMOD*, pages 943–954, 2014.
13. A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.
14. Z. A. Kocsis, J. H. Drake, D. Carson, and J. Swan. Automatic improvement of apache spark queries using semantics-preserving program reduction. In *GECCO*, pages 1141–1146, 2016.
15. Y. Li, M. Li, L. Ding, and M. Interlandi. Rios: Runtime integrated optimizer for spark. In *ACM Symposium on Cloud Computing*, pages 275–287, 2018.
16. A. A. B. Lima, C. Furtado, P. Valduriez, and M. Mattoso. Parallel OLAP query processing in database clusters with data replication. *DaPD*, 25(1-2):97–123, 2009.
17. L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *ACM SIGMOD*, pages 631–645, 2018.
18. R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *ACM SIGMOD*, pages 1137–1148, 2011.
19. A. Quamar, K. A. Kumar, and A. Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *EDBT*, pages 430–441, 2013.
20. M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *VLDB Endow.*, 10(4):445–456, 2016.
21. T. Stöhr, H. Märtens, and E. Rahm. Multi-dimensional database allocation for parallel data warehouses. In *VLDB*, pages 273–284, 2000.
22. R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *VLDB Endow.*, 8(3):245–256, 2014.
23. T. Zhang, A. Tomasic, Y. Sheng, and A. Pavlo. Performance of OLTP via intelligent scheduling. In *ICDE*, pages 1288–1291, 2018.
24. W. Zhang, J. Kim, K. A. Ross, E. Sedlar, and L. Stadler. Adaptive code generation for data-intensive analytics. *Proceedings of the VLDB Endowment*, 14(6):929–942, 2021.
25. D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.