Relax and Let the Database do the Partitioning Online

Alekh Jindal and Jens Dittrich

Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de

Abstract. Vertical and Horizontal partitions allow database administrators (DBAs) to considerably improve the performance of business intelligence applications. However, finding and defining suitable horizontal and vertical partitions is a daunting task even for experienced DBAs. This is because the DBA has to understand the physical query execution plans for each query in the workload very well to make appropriate design decisions. To facilitate this process several algorithms and advisory tools have been developed over the past years. These tools, however, still keep the DBA in the loop. This means, the physical design cannot be changed without human intervention. This is problematic in situations where a skilled DBA is either not available or the workload changes over time, e.g. due to new DB applications, changed hardware, an increasing dataset size, or bursts in the query workload. In this paper, we present AutoStore: a self-tuning data store which rather than keeping the DBA in the loop, monitors the current workload and partitions the data automatically at checkpoint time intervals — without human intervention. This allows AutoStore to gradually adapt the partitions to best fit the observed query workload. In contrast to previous work, we express partitioning as a One-Dimensional Partitioning Problem (1DPP), with Horizontal (HPP) and Vertical Partitioning Problem (VPP) being just two variants of it. We provide an efficient O^2P (One-dimensional Online Partitioning) algorithm to solve 1DPP. O^2P is faster than the specialized affinity-based VPP algorithm by more than two orders of magnitude, and yet it does not loose much on partitioning quality. AutoStore is a part of the OctopusDB vision of a One Size Fits All Database System [13]. Our experimental results on TPC-H datasets show that AutoStore outperforms row and column layouts by up to a factor of 2.

Key words: changing workload, online partitioning, self-tuning

1 Introduction

Physical database designs have been researched heavily in the past [23, 3, 2, 5, 6, 31, 7, 22, 26, 12]. As a consequence, nowadays, most DBMSs offer design advisory tools [1, 32, 30, 4]. These tools help DBAs in defining indexes, e.g. [8], as well as horizontal and/or vertical partitions, e.g. [3, 15]. The idea of these tools is to analyze the workload at a given point in time and suggest different

physical designs. These suggestions are computed by a what-if analysis. What-if analysis explores the possible physical designs. However, just finding the right set of partitions is NP-hard [29]. Therefore the search space must be pruned using suitable heuristics, i.e. typically some greedy-strategy [2]. The cost of each candidate configuration is then estimated using an existing cost-based optimizer, i.e. the optimizer is tricked into believing that the candidate configuration already exists. Eventually, a suitable partitioning strategy is proposed to the DBA who then has to re-partition the existing database accordingly.

1.1 Problems with Offline Partitioning

The biggest problem with this approach is that it is an *offline* process. The DBA will only reconsider the current physical design at certain points in time. This is problematic. Assume the workload changes over time, e.g. changes in the workload due to new database applications, an increasing dataset size, or an increasing number of queries. In these situations the existing partitioning strategies should be revisited to improve query times. In the current *offline approach* however, the partitioning strategies will only be changed if a human — the DBA — triggers an advisory tool with the most recent query logs and eventually decides to repartition the data. This means, the vertical and horizontal partitioning strategies are carved in stone until the DBA changes them eventually. Furthermore, current advisory tools attempt to find near optimal partitioning strategies, which is very expensive. This is especially problematic if the database system has to handle bursts and peaks. For instance consider (i) a ticket system selling 10 million Rolling Stones tickets within three days; (ii) an online store such as Amazon selling much higher volumes before christmas; or (iii) an OLAP system having to cope with new query patterns. In these types of applications it is not acceptable for users to wait for the DBA and the advisory tool to reconfigure the system. If the system stalls due to a peak workload, the application provider may loose a lot of money.

1.2 Research Goals and Challenges

Our **goal** is to research a database store that decides on the suitable partitioning strategy *automatically*, i.e. without any human intervention. As the search space of possible partitions is huge [29], it is clear from the beginning that an *optimal* automatic partitioning is not always feasible. However, we believe that an automatic (or: *online*) partitioning will in most cases be much better than the one suggested by even a skilled DBA — similarly the physical query execution plans being in most cases better than hand-crafted plans. The risk of not reaching optimality is similar to the risk of adaptive indexing [14] and cracking [19]. However, the possible gains of such an approach may be similarly tremendous.

This leads to interesting **research challenges**:

(1.) There exist a plethora of state-of-the-art offline algorithms, e.g. [23, 24, 2], for suggesting suitable vertical and horizontal partitions. However, given the

huge search space, it turns out that their runtime complexity is unacceptably high to be applicable in an online setting where the available time to decide on a new partitioning is rather limited. Therefore we must develop new algorithms.
(2.) We need algorithms that decide on data partitioning automatically and while the database is running, i.e. take decisions to repartition the data.
(3.) Any automatic repartitioning must not block or stall the database and/or access to entire tables, a problem more likely in archival disk-based databases.

1.3 Contributions

In this paper, we present AutoStore, a fully automatic database store, to solve these challenges. To the best of our knowledge, this work is the first to solve the database partitioning problem with a fully automatic *online approach*. The contributions of this paper are:

(1.) We express partitioning as general One-Dimensional Partitioning Problem (1DPP), with Vertical (VPP) and the Horizontal Partitioning Problem (HPP) as subproblems of it. Both subproblems may be solved by solving 1DPP (Section 2).
(2.) We present AutoStore, an online self-tuned database store that is a step towards implementing the OctopusDB vision [13, 21]. The core components of AutoStore are: dynamic workload monitor, partitioning unit clusterer, partitioning analyzer, and partitioning optimizer (Section 3).
(3.) We present an online database partitioning algorithm O^2P (One-dimensional Online Partitioning) to solve 1DPP (Section 4).
(4.) We show an extensive evaluation of our algorithm over TPC-H and SSB benchmarks. We present experimental results from mixed OLTP/OLAP workloads over a main-memory and a BerkeleyDB implementation (Section 5).

2 Vertical and Horizontal Partitioning

2.1 Preliminaries

Typically, users partition databases horizontally based on data value ranges, hashes, or lists. This is because data values are comparable across a column. However, this is not true for data values across a row. Therefore, sophisticated partitioning methods have been developed for VPP. In this section we will revisit the basics of VPP. This also serves as the ground work for our 1DPP.
Naïve Approach. Database researchers pointed out the heuristic [16] and NP-hard [29] nature of partitioning problem pretty early. The number of ways to partition vertically, for x attributes, is given by bell number $B(x)$. The naïve approach to find the optimal solution is to enumerate all bell numbers. The complexity of the naïve approach is $O(x^x)$, making it infeasible for large databases.
Affinity based Approach. The naïve approach considers all possible partitions, even the ones having attributes which are *never* accessed together. To address this, *attribute affinity* was introduced as a measure of pairwise attribute

similarity [18, 23, 10, 24, 11]. The core idea of affinity based partitioning is to compute affinities between every pair of attributes and then to cluster them such that high affinity pairs are as close in neighborhood as possible. To compute affinity between different attributes, we need to know their access patterns. A *usage function* $U(q, a)$ denotes whether or not query q references attribute a. $U(q, a) = 1$ if q references a and 0 otherwise. For example, in TPC-H Lineitem table, $U(Q_1, \text{PartKey}) = 1$ as Query 1 references attribute PartKey. The usage function may also be extended to incorporate query weights, reflecting the importance levels or relative frequencies of queries. To measure the affinity between two attributes a_i and a_j, the *affinity function* $A(a_i, a_j)$ simply counts their co-occurrences in the query workload, i.e. $A(a_i, a_j) = \sum_q U(q, a_i) \cdot U(q, a_j)$. For instance, in Lineitem table, $A(\text{PartKey}, \text{SuppKey}) = 5$, as attributes PartKey and SuppKey co-occur in five queries. The affinity function produces a 2D affinity matrix between every pair of attributes. The goal now is to cluster the matrix such that the cells having similar affinity values are placed close together in the matrix. Every order of rows and columns in the matrix gives a new *ordering of attributes* ($\preceq$). For example, consider the following affinity matrices for PartKey, SuppKey, and Quantity attributes in TPC-H Lineitem table. The left matrix represents an attribute ordering PartKey $\preceq$ SuppKey $\preceq$ Quantity, whereas the right matrix represents ordering PartKey$\preceq$Quantity$\preceq$SuppKey.

	PartKey	SuppKey	Quantity
PartKey	8	5	6
SuppKey	5	8	4
Quantity	6	4	9

	PartKey	Quantity	SuppKey
PartKey	8	6	5
Quantity	6	9	4
SuppKey	5	4	8

Given attribute ordering $\preceq$, an *affinity measure* $M(\preceq)$ measures the quality of the affinity clustering as $M(\preceq) = \sum_{i=1}^{x} \sum_{j=1}^{x} A(a_i, a_j)[A(a_i, a_{j-1}) + A(a_i, a_{j+1})]$. It holds that $A(a_0, a_j) = A(a_i, a_0) = A(a_{x+1}, a_j) = A(a_i, a_{x+1}) = 0$. For the left matrix above $M(\preceq) = 404$ and for the right matrix $M(\preceq) = 440$. Indeed, the right matrix has better clustering since affinity between attributes PartKey and Quantity (=6) is more than that between PartKey and SuppKey (=5). Thus, the objective of affinity matrix clustering problem now is to maximize the affinity measure. One (greedy) approach is to place the attributes one-by-one such that the *contribution to the affinity measure* at each step is maximized [23]. The contribution to the affinity measure of a new attribute a_k when placed between two already placed attributes a_i and a_j is: $Cont(a_i, a_j, a_k) = 2 \cdot \sum_{z=1}^{n} [A(a_z, a_i) \cdot A(a_z, a_k) + A(a_z, a_k) \cdot A(a_z, a_j) - A(a_z, a_i) \cdot A(a_z, a_j)]$. In this clustering approach, we first place two random attributes; then, in the neighborhood, we place the attribute which maximizes the contribution to the affinity measure. We repeat this process until all attributes are placed.

2.2 Problem Statement

In this section we express HPP and VPP as a general 1DPP. The first step to do so is to identify the smallest indivisible units of storage.

Definition 1. *A **partitioning unit set** $P_u = \{u_1, u_2, .., u_n\}$ is the set of n smallest pieces of data.*

Definition 2. *A **partitioning unit ordering** $\preceq$ defines an order on the partitioning units in P_u.*

Partitioning units could be attributes along the vertical axis or tuples along the horizontal axis. However, partitioning at the tuple level may not make sense due to large number of partitioning units and hence high complexity. Therefore, we usually consider sets of tuples, based on some key, as partitioning units (horizontal partitioning). Similarly, we could also consider groups of columns as partitioning units (vertical partitioning). Below, we introduce some new concepts needed for our one-dimensional partitioning problem statement. First, we express partitioning as a logical partitioning, to be able to use it in an algorithm.

Definition 3. *A **split vector** S is a row vector of (n-1) split lines in ordering $\preceq$, where a split line s_j is defined between partitioning units u_j and u_{j+1} as follows:*

$$s_j = \begin{cases} 1 & \text{if there is split between } u_j \text{ and } u_{j+1} \\ 0 & \text{for no split} \end{cases}.$$

A split vector S captures the logical partitioning over a given dataset. For instance, a split vector S_1=[0,0,0,1,0,1,1] corresponds to a partitioning of $u_1, u_2, u_3, u_4|u_5, u_6|u_7|u_8$. However, in order to estimate costs using a cost-based query optimizer, a split vector still needs to be translated in terms of partitioning units:

Definition 4. *A **partition** $p_{m,r}(S,\preceq)$ is a maximal chunk of adjacent partitioning units from u_m to u_r, such that split lines s_m to s_{r-1} are all 0.*

Definition 5. *A **partitioning scheme** $P(S,\preceq)$ over relation R is a set of disjoint and complete partitions, i.e.*

$$\bigcup_x p_{m_x,r_x}(S,\preceq) = R,$$
$$p_{m_x,r_x}(S,\preceq) \cap p_{m_y,r_y}(S,\preceq) = \phi, \forall x, y \text{ such that } x \neq y.$$

Partitioning scheme expresses the actual arrangement of partitioning units, given a split vector. For instance, for split vector S_1, partition $p_{1,4}(S_1,\preceq)$ is $\{u_1, u_2, u_3, u_4\}$ and partitioning scheme $P(S_1,\preceq)$=$\{p_{1,4}(S_1,\preceq)$,$p_{5,6}(S_1,\preceq)$, $p_7(S_1,\preceq)$, $p_8(S_1,\preceq)\}$. Finally, in order to evaluate partitioning schemes in an online setting, we need to model the online query workload.

Definition 6. *An **Online Workload** W_{t_k} is a stream of queries $\{q_0, ..., q_{t_{k-1}}, q_{t_k}\}$ seen till time t_k, where $t_k > t_{k-1} > ... > 0$.*

Further, let $C_{\text{est.}}(W_{t_k}, P(S,\preceq))$ denote the execution cost of workload W_{t_k} as estimated by a cost-based optimizer. Now, we express our one-dimensional partitioning problem as follows.

One-dimensional Online Partitioning Problem. *Given an online workload W_{t_k} and partitioning unit ordering $\preceq$, find the split vector S' that minimizes the estimated workload execution cost, i.e.*

$$S' = \underset{S}{\operatorname{argmin}}\, C_{\text{est.}}\Big(W_{t_k}, P(S,\preceq)\Big). \tag{1}$$

The complexity of the above problem depends on the number of partitioning schemes P, which in turn depends on the range of values of split vector S. Note that the one-dimensional partitioning problem has the following sub-problems: (1) *Vertical Partitioning*, if the partitioning unit set P_u is a set of attributes, and (2) *Horizontal Partitioning*, if the partitioning unit set is a set of horizontal ranges.

In the next section we describe our AutoStore system and discuss how it solves the one-dimensional partitioning problem in an online setting.

3 AutoStore

In this section we present AutoStore, an automatically and online partitioned database store. The workflow in AutoStore is as follows: (1) dynamically monitor the query workload and dynamically cluster the partitioning units; (2) analyze the affinity matrix at regular intervals and decide whether or not to create the partitions; and (3) keep data access unaffected while monitoring workload and analyzing partitions. The salient features of AutoStore are as follows:

(1.) *Physical Data Independence.* Instead of exposing physical data partitioning to the user, AutoStore hides these details. This avoids mixing the logical and physical schema. Thus, AutoStore offers better physical data independence, which is not the case in traditional databases.
(2.) *DBA-oblivious Tuning.* The DBA is not involved in triggering the right data partitioning. AutoStore self-tunes its data.
(3.) *Workload Adaptability.* AutoStore monitors the *changes* in query workload and automatically adapts data partitioning to it.
(4.) *Generalized Partitioning.* AutoStore treats partitioning as a 1DPP. Subproblems VPP and HPP are handled equivalently by rotating the table through ninety degrees, i.e. changing the partitioning units from attributes to ranges.
(5.) *Cost, Benefit Optimization.* To decide whether or not to actually partition data, AutoStore considers both the costs as well as the benefits of partitioning.
(6.) *Uninterrupted Query Processing.* AutoStore makes use of our online algorithm O^2P which amortizes the computationally expensive partitioning analysis over several queries. The query processing remains uninterrupted.

Below we discuss four crucial components — workload monitor, partitioning unit clusterer, partitioning analyzer, and partitioning optimizer — which make online self-tuning possible in AutoStore.
Workload Monitor. Online partitioning faces the challenge of creating good partitions for future queries based on the seen ones. One might consider partitioning after every incoming query. However, not only could this be expensive (due to data shuffling), the next incoming query could be entirely different from the previous one. Hence, we maintain a *query window* to capture the workload pattern and have greater confidence over partitioning decisions. Additionally, we slide the query window once it grows to a maximum size, to capture the latest workload trends, . We denote a sliding query workload having N queries as

$W_{t_k}^N \subseteq W_{t_k}$. After every `CheckpointSize` number of new queries, AUTOSTORE triggers partitioning analysis, i.e., it takes a snapshot of the current query window and the partitioning unit ordering and determines the new partitioning.

Partitioning Unit Clusterer. The partitioning unit clusterer is responsible for re-clustering the affinity matrix after each incoming query. The affinity matrix clustering algorithm (in Section 2.1) has the following issues: (i) it recomputes all affinity values, and (ii) it reclusters the partitioning units from scratch. We need to adapt it for online partitioning in AUTOSTORE. The core idea is to compute all affinities once and then for each incoming query update only the affinities between referenced partitioning units. Note that the change in each of these affinity values will be 1, due to co-occurrence in the incoming query. For example, consider TPC-H Lineitem table having the affinity matrix shown at left below. Now, for an incoming query accessing PartKey and SuppKey, only the affinities between them are updated (gray cells in the right affinity matrix above). Likewise, we need to re-cluster only the referenced partitioning units. To do this, we keep the first referenced partitioning unit at its original position, and for the i^{th} referenced unit we consider the left and right positions of the $(i-1)$ referenced units already placed. We calculate the net contribution of i^{th} referenced unit to the global affinity measure as: ($Cont$ at the new position) – ($Cont$ at the current position). We choose the position that offers maximum net contribution to the global affinity measure and repeat the process for all referenced partitioning units. To illustrate, in the right affinity matrix above, we first place PartKey and then consider placing SuppKey to the left (net contribution=48) and right (net contribution=0) of Partkey. Thus, we will place SuppKey to the left of PartKey.

	PartKey	Quantity	SuppKey
PartKey	8	6	5
Quantity	6	9	4
SuppKey	5	4	8

	PartKey	Quantity	SuppKey
PartKey	9	6	6
Quantity	6	9	4
SuppKey	6	4	9

Partitioning Analyzer. The partitioning analyzer of AUTOSTORE analyzes partitioning every time `CheckpointSize` number of queries are added by the workload monitor. The job of the partitioning analyzer is to take a snapshot of the query window as input, enumerate and analyze the partitioning candidates, and emit the best partitioning as output. In the brute force enumeration approach, we consider all possible values (0 or 1), for each split line in the split vector S of Equation 1. We then pick the split vector which produces the lowest estimated workload execution cost $C_{\text{est.}}(W_{t_k}^N, P(S,\preceq))$. Each split vector gives rise to a different candidate partitioning scheme. The size of the set of candidate partitioning schemes is 2^{n-1}. In Section 4 we show how the O^2P algorithm significantly improves partitioning analysis in an online setting.

Partitioning Optimizer. Given the partitioning scheme P' produced by the partitioning analyzer, the partitioning optimizer decides whether or not to transform the current partitioning scheme P to P'. The partitioning optimizer considers the expected costs of transforming the partitioning scheme as well as the expected benefits from it. We discuss these considerations below.

Cost Model. We use a cost model for full table and index scan operations over one-dimensional partitioned tables. To calculate the partitioning costs, we first

find the partitions in P which are no longer present in P': $P_{\text{diff}} = P \setminus P'$. Now, to transform from P to P' we simply have to read each of the partitions in P_{diff} and store it back in the required partitions in P'. For instance, the transformation cost for vertical partitioning can be estimated as twice the scan costs of partitions in P. From such a transformation cost model, the worst case transformation cost is equal to twice the full table scan, whereas the best case transformation cost is twice the scan cost of the smallest partitioning unit.

Benefit Model. Same as we compute the cost of partitioning, we also need the benefit of partitioning in order to make a decision. We model partitioning benefit as the difference in the cost of executing the query window on the current and the new partitioning, i.e. $B_{\text{transform}} = C_{\text{est.}}(W^N_{t_k}, P(S,\preceq)) - C_{\text{est.}}(W^N_{t_k}, P'(S,\preceq))$.

Partitioning Decision. For each transformation made, we would have recurring benefits over all similar workloads. Hence, we need to factor in the expected frequency of the query window. This could be either explicitly provided by the user or modeled by the system. For instance, an exponential decaying model with shape parameter y: Workload Frequency$(f) = \frac{1}{1-y^{-\texttt{MaxWindowSize}}}$ gives higher frequency to smaller query windows. AutoStore creates the new partitioning only if the total recurring partitioning benefit ($\texttt{pBenefit} = f \cdot \ B_{\text{transform}}$) is expected to be higher than the partitioning cost ($\texttt{pCost} = C_{\text{transform}}$).

Partitioning Transformation/Repartitioning. Repartitioning data from P to P' poses interesting algorithmic research challenges. As stated before the overall goal should be to minimize transformation costs. In addition, the database or even single tables must not be stalled, i.e. by halting incoming queries. Fortunately, these problems may be solved. In a *read-only* system any table or horizontal partition may be transformed in the background, i.e. we transform P to P' and route all incoming queries to P. Only if the transformation is finished, we atomically switch to P'. For *updates*, this process can be enriched by keeping a differential file or log L of the updates that are arriving while the tranformation is running. Any incoming query may then be computed by considering P and L. If the tranformation is finished, we may eventuall decide to merge P' with P. The right strategy for doing this is not necessarily to merge immediately. A similar discussion as for LSM trees [25] and exponential files [20] applies. For repartitioning vertical layouts there are other interesting challenges. None of them would require us to stall incoming queries or halt the database. We are planning to evaluate these algorithms in a separate study.

4 O^{2}P Algorithm

The partitioning analyzer in Section 3 described the brute force approach of enumerating all possible values of split vector S in the one-dimensional partitioning problem. This approach has exponential complexity and hence is not desirable in an online setting. In this section, we present an online algorithm O^{2}P (One-dimensional Online Partitioning) which solves 1DPP in an online setting. O^{2}P does not produce the optimal partitioning solution. Instead, it uses a number of techniques to come up with greedy solution. The greedy solution is

not only dynamically adapted to workload changes, it also does not loose much on partitioning quality as well. Below we highlight the major features in O^2P:

(1.) Partitioning Unit Pruning. Several partitioning units are never referenced by any of the queries in the query window. For instance, `RetailPrice` is not referenced in TPC-H `Part` table. Due to one-dimensional clustering, such partitioning units are expected to be in the beginning or the end of the partitioning unit ordering. Therefore, O^2P prunes them into a separate partition right away. As an example, consider 3 leading and 2 trailing partitioning units, out of total 10 partitioning units, to be not referenced. Then, the following split lines are determined: $s_1 = s_2 = 0$, $s_3 = 1$, $s_9 = 0$, $s_8 = 1$.
(2.) Greedy Split Lines. Instead of enumerating over all possible split line combinations — as in the brute force — O^2P greedily sets the best possible split line, one at a time. O^2P starts with a split vector having all split lines as 0 and at each iteration it sets (to 1) only one split line. To determine which split line to set, O^2P considers all split lines unset so far, and picks the one giving the lowest workload execution cost, i.e. the $(i+1)^{th}$ split line to be set is given by: $s_{i+1} = \text{argmin}_{s \in \text{unset}(S_i)}\ C_{\text{est.}}(W^N_{t_k}, P(S_i + U(s), \preceq))$, where $U(s)$ is a unit vector having only split line s as set; corresponding split vector is: $S_{i+1} = S_i + U(s)$.
(3.) Dynamic Programming. Observe that the partitions not affected in the previous iteration of greedy splitting will have the same best split line in the current iteration. For example, consider an ordering of partitioning units with binary partitioning: $u_1, u_2, u_3, u_4 | u_5, u_6, u_7, u_8$. The corresponding split vector is: $[0, 0, 0, 1, 0, 0, 0]$ with only split line s_4 set to 1 and all other split lines set to 0. Now, we consider all unset split lines for partitioning. Suppose s_2 and s_6 are the best split lines in the left and right partitions respectively and amongst them s_2 is the better split line. In next iteration, we already know that s_6 is the best split line in the right partition and only need to evaluate s_1 and s_3 in the left partition again. To exploit this O^2P maintains the best split line in each partition and reevaluates split lines only in partitions which are further split. Since it performs only one split at a time (greedy), it only needs to reevaluate the split lines in the most recently split partition. Algorithm 1 shows the dynamic programming based enumeration in O^2P. First, O^2P finds the best split line and its corresponding cost in: the left and right parts of the last partition, and all previous partitions (Lines 1–6). If no valid split line is found then O^2P returns (Lines 7–9). Otherwise, it compares these three split lines (Line 10), chooses the one having lowest costs (Lines 11–35), and repeats the process (Line 36).

Theorem 1. O^2P *produces the correct greedy result.*

Theorem 2. *If consecutive splits reduce the partitioning units by z elements, then the number of iterations in* O^2P *is* $\left[\frac{(n-3)(n-z-1)}{2z} + 2n - 3\right]$.

Lemma 1. *Worst case complexity of* O^2P *is* $O(n^2)$.

Lemma 2. *Best case complexity of* O^2P *is* $O(n)$. *(All proofs in Appendix D.)*

Algorithm 1: `dynamicEnumerate`

```
   Input  : S, left, right, PrevPartitions
   Output: Enumerate over possible split vectors
 1 SplitLine sLeft = BestSplitLine(S,left);
 2 Cost minCostLeft = BestSplitLineCost(S,left);
 3 SplitLine sRight = BestSplitLine(S,right);
 4 Cost minCostRight = BestSplitLineCost(S,right);
 5 SplitLine sPrev = BestSplitLine(S,PrevPartitions);
 6 Cost minCostPrev = BestSplitLineCost(S,PrevPartitions);
 7 if invalid(sLeft) and invalid(sRight) and invalid(sPrev) then
 8 |  return;
 9 end
10 Cost minCost = min(minCostLeft, minCostRight, minCostPrev);
11 if minCost == minCostLeft then
12 |  SetSplitLine(S, sLeft);
13 |  if sRight > 0 then
14 |  |  AddPartition(right, sRight, minCostRight);
15 |  end
16 |  right = sLeft+1;
17 else if minCost == minCostRight then
18 |  SetSplitLine(S, sRight);
19 |  if sLeft > 0 then
20 |  |  AddPartition(left, sLeft, minCostLeft);
21 |  end
22 |  left = right;
23 |  right = sRight+1;
24 else
25 |  SetSplitLine(S, sPrev);
26 |  if sRight > 0 then
27 |  |  AddPartition(right, sRight, minCostRight);
28 |  end
29 |  if sLeft > 0 then
30 |  |  AddPartition(left, sLeft, minCostLeft);
31 |  end
32 |  RemovePartition(sPrev);
33 |  left = pPrev.start();
34 |  right = sPrev+1;
35 end
36 dynamicEnumerate(S, left, right, PrevPartitions);
```

(4.) Amortized Partitioning Analysis. O^2P computes the partitioning lazily over the course of several queries, i.e. it performs a subset of iterations each time AutoStore triggers the partitioning analyzer. Thus, O^2P amortizes the cost of computing the partitioning scheme over several queries. This makes sense, because otherwise we may end up spending a large number of CPU cycles, and blocking query execution, even though the partitioning may not be actually done (due to cost-benefit considerations). The partitioning analyzer returns the best split vector only when all iterations in the current analysis are done.

(5.) Multi-threaded Analysis. Since our partitioning analysis works on a window snapshot of the workload, O^2P can also delegate it to a separate secondary thread while the normal query processing continues in the primary thread. This approach completely separates query processing from partitioning analysis. However, the entire database will need to be locked by the primary thread once the partitioning optimizer decides to partition the data.

5 Experiments

The goal of our experiments are four-fold: (1) to evaluate the performance of O^{2}P, (2) to evaluate the partitioning analysis in AutoStore on realistic TPC-H and SSB workloads, (3) to compare the query performance of a main-memory based implementation of AutoStore with No and Full Vertical Partitioning, and (4) to evaluate the performance of AutoStore on a real system: BerkeleyDB. We present each of these in the following. All experiments were executed on a large computing node having Intel Xeon 2.4GHz CPU with 64GB of main memory, and running on Ubuntu 10.10 operating system.

The algorithms of Navathe et. al. [23] and Hankins et. al. [17] have similar complexity. Therefore, we label them as NV/HC. To compare and obtain a cost analysis of different components in O^{2}P, we switch them on incrementally. Thus, we have five different variants of O^{2}P: (i) only partitioning unit pruning (O^{2}Pp), (ii) pruning+greedy (O^{2}Ppg), (iii) pruning+greedy+dynamic (O^{2}Ppgd), (iv) pruning+greedy+dynamic+amortized (O^{2}Ppgda), and (v) pruning +greedy+dynamic+multi-threaded (O^{2}Ppgdm).

5.1 Comparing Online Algorithms

First, we evaluate the complexities of the different variants of O^{2}P. We vary the number of partitioning units for each of the variants and record the number of iterations taken by them. Figure 1 shows the performance of the different variants of O^{2}P while varying the number of partitioning units. In the NV/HC approaches the number of iterations grow exponentially with the number of partitioning units. For O^{2}Pp, the number of iterations depend on the number of non-referenced partitioning units. O^{2}Ppg and O^{2}Ppgd, however, outperform NV/ HC by several orders of magnitude and tend to be linear in the number of partitioning units. Note that O^{2}Ppgda and O^{2}Ppgdm will have the same number of iterations as in O^{2}Ppgd, and hence we do not show them in the figure.

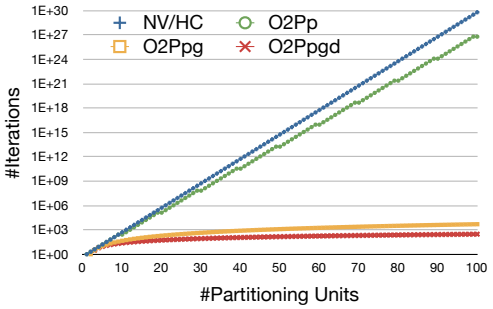
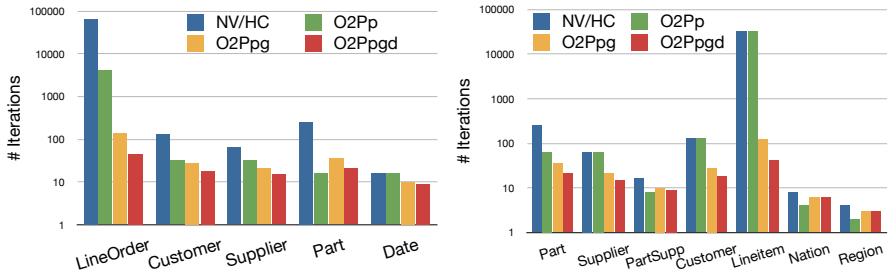


Fig. 1. Number of iterations in different algorithms [assuming 10% dead attributes]

5.2 Evaluating Partitioning Analyzer

We now evaluate O^{2}P on multiple benchmark datasets and workloads. Figure 2(a) shows the number of iterations in different variants of O^{2}P for different tables in Star Schema Benchmark (SSB). We can see that O^{2}Pp indeed improves



(a) SSB Benchmark (b) TPC-H Benchmark

Fig. 2. Number of iterations in different algorithms over SSB and TPC-H benchmarks on different tables.

over NV/HC on this realistic workload. O^{2}Ppg and O^{2}Ppgd are even better. Figure 2(b) shows the iterations in different variants of O^{2}P for TPC-H dataset. For Lineitem table, O^{2}Ppgd has just 42 iterations compared to 32,768 iterations in NV/HC. O^{2}Ppgda and O^{2}Ppgdm have the same number of iterations as O^{2}Ppgd, hence we do not show them in the figure.

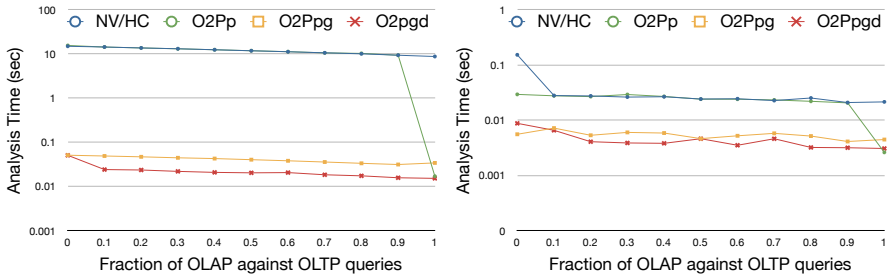
Next, we evaluate the actual running time of different O^{2}P variants while varying the read-only workload. We vary the 100-query workload from OLTP style (1% tuple selectivity, 75-100% attribute selectivity) to OLAP style (10% tuple selectivity, 1-25% attribute selectivity) access patterns. We run this experiment over Lineitem (Figure 3(a)) and Customer tables (Figure 3(b)). We observe that on Lineitem O^{2}Ppgd outperforms NV/HC by up to two orders of maginitude.

Now let us analyze the *quality* of partitioning produced by O^{2}P. We define the quality of partitioning produced by an algorithm as the ratio of the expected query costs of optimal partitioning and the expected query costs of partitioning produced by the algorithm. The table below shows the quality and the number of iterations for optimal, NV, and O^{2}P partitioning over mixed OLTP-OLAP workload

	Customer			Lineitem		
	Optimal	**Navathe**	**O2P**	**Optimal**	**Navathe**	**O2P**
Quality	100%	99.29%	92.76%	100%	97.45%	95.80%
Iterations	100%	14.60%	2.28%	100%	2.42%	0.14%

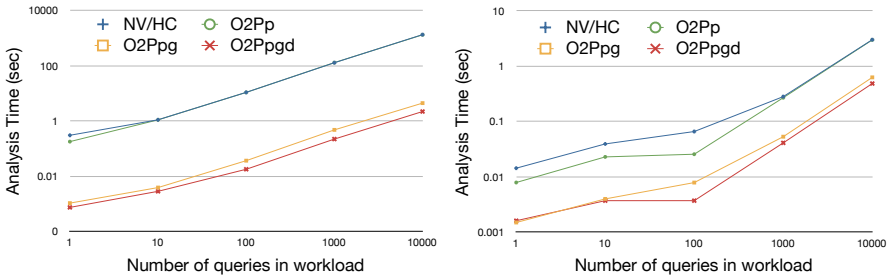
We can see that O^{2}P significantly reduces the number of iterations, without loosing much on partitioning quality.

Finally, we evaluate the scalability of O^{2}P when increasing workload size. We vary the workload size from 1 to 10,000 queries consisting of equal number of OLTP and OLAP-style queries. Figures 4(a) and 4(b) show the scalability of O^{2}P over TPC-H Lineitem and Customer tables respectively. We can see that all variants of O^{2}P algorithm scale linearly with the workload size. Hence, from now on we will only consider O^{2}Ppgd algorithm.



(a) TPC-H Lineitem (b) TPC-H Customer

Fig. 3. Running times of different algorithms over changing workload type [100 queries each].



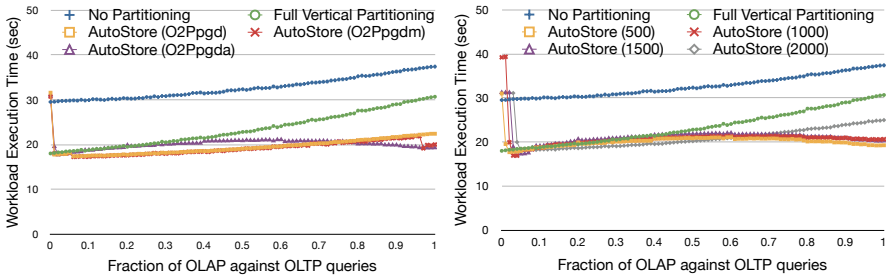
(a) TPC-H Lineitem (b) TPC-H Customer

Fig. 4. Running time of different algorithms over varying workload size [with 50% OLAP, 50%OLTP queries].

5.3 Evaluating Query Performance

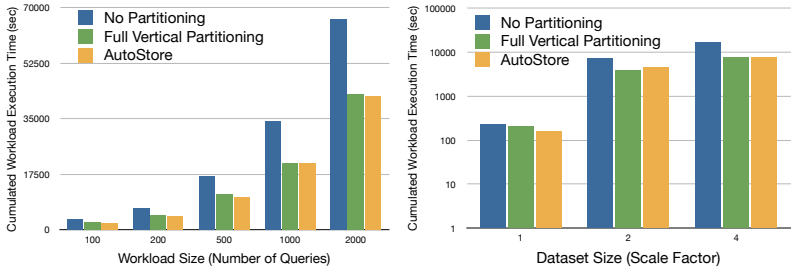
Now we evaluate the query execution performance of AutoStore in comparison with No and Full Vertical Partitioning. In this evaluation we use a main-memory implementation of AutoStore in Java. In order to show how AutoStore adapts vertical partitioning to the query workload, we use a universal relation de-normalized from a variant of the TPC-H schema [28]. Similar as in [28], we choose a vertical partition with part key, revenue, order quantity, lineitem price, week of year, month, supplier nation, category, brand, year, and day of week for our experiments. Further, since we consider equal size attributes only, we map all attributes to integer values, while preserving the same domain cardinality. We use a scale factor (SF) of 1.

Figure 5(a) shows the performance of No Partitioning, Full Vertical Partitioning, AutoStore with O^2Ppgd, AutoStore with O^2P pgdm and AutoStore with O^2Ppgda. We vary the fraction of data accessed, i.e. both the attribute and tuple selectivity along the x-axis. We vary the OLTP/OLAP read access patterns as in Section 5.2, with a step size of 0.01%. From the figure we can see that AutoStore automatically adapts to the changing workload, i.e. even though it starts with no-partitioning configuration, AutoStore matches or im-



(a) Comparing different methods (b) Varying query window size

Fig. 5. Comparison of No Partitioning, Full Vertical Partitioning, and AutoStore in main-memory implementation.



(a) Varying workload size (b) Varying dataset size

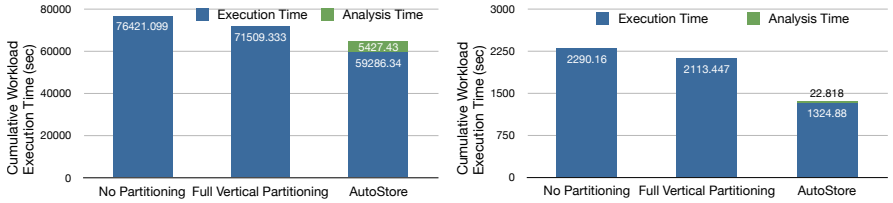
Fig. 6. Scalability performance of AutoStore.

proves full vertical partitioning performance. Therefore, from now on we consider only O^{2}Ppgda. Figure 5(b) shows the performance of AutoStore when varying query window size. From the figure we observe that larger query windows, e.g. query window of 2000 after 70% OLAP, become slower. This is because the partitioning analyzer has to now estimate the costs of more number of queries while analyzing partitioning schemes.

Scalability Experiments. Figure 6(a) shows AutoStore performance when the varying workload size from 100 to 2,000 queries. We observe that AutoStore scales well with workload size in comparison to No and Full Vertical Partitioning. Figure 6(b) shows the scalability of AutoStore when increasing the dataset size. We can see that AutoStore scales gracefully with data size.

5.4 Evaluation over Real System

Modern database systems, e.g. PostgreSQL, have a very strong coupling between their query processors and data stores. This makes it almost impossible to replace the underlying data store without touching the entire software stack on top. This limitation led us to consider BerkeleyDB (Java Edition), which is quite flexible in terms of physical data organization, for prototyping.



(a) TPC-H Lineitem (b) TPC-H Customer

Fig. 7. Total running time of varying OLTP-OLAP workload on different tables in BerkeleyDB.

In BerkeleyDB we store a key-value pair for each tuple. The key is composed of the partition ID and the tuple ID, and the value contains all attribute values in that partition. Since BerkeleyDB sorts the data on keys by default, we simple need to change the partition ID to change the partitioning scheme. Again, we vary the OLTP/OLAP read access patterns as in Section 5.2, with a step size of 0.01% over TPC-H dataset (total size 1GB). For different layouts, Figure 7(a) shows the total query execution times over TPC-H Lineitem table and Figure 7(b) shows the total query execution times over TPC-H Customer table. In general, AutoStore outperforms the best layout in each of the tables. For instance, even though AutoStore starts from a no-partitioning configuration, it improves over Full Vertical Partitioning by 36% in Customer table.

6 Related Work

Offline Horizontal and Vertical Partitioning. Horizontal partitioning is typically done based on values (range, hash, or list). A recent work proposed workload-based horizontal partitioning [12]. However, it is still offline. Vertical partitioning started with early approaches of heuristic based partitioning [16] of data files. The state-of-the-art work in vertical partitioning [23] develops the notion of attributes affinity, quantifying attribute co-occurrence in a given set of transactions. This work creates a clustered attribute affinity matrix in the first step and applies binary partitioning repetitively in the second step. A follow-up work [24] presents graphical algorithms to improve the complexity of their approach. Other works took the type of scan into account to analyze the disk accesses [10] and formulated an integer linear programming problem to arrive at the optimal partitioning for relational databases [10]. Next, researchers proposed transaction based vertical partitioning [9], arguing that since transactions have more semantic meaning than attributes, it makes more sense to partition attributes according to a set of transactions. However, all of these works considered data partitioning as a one-time *offline* process, in contrast to the online approach to data partitioning in AutoStore. Recent works integrate partitioning into physical database design tuning problem [3] with the objective of finding the configuration producing the minimum workload cost within the stor-

age bound. This work first produces *interesting column groups* by applying a heuristic-based pruning of the set of all column groups. The column groups are then merged before all possible partitioning schemes are enumerated. These steps are not feasible in an online setting. HYRISE [15] analyzes the access patterns to partitions data. However, this approach is (1) still offline, i.e. it is not able to adapt to changing workloads, (2) restricted to main memory, whereas AutoStore works for both disk and main memory DBMSs, and (3) is limited to vertical partitioning, whereas AutoStore solves VPP and HPP equivalently.
Online Physical Tuning. Dynamic materialized views [31] materialize the frequently accessed rows dynamically. [6] proposes online physical tuning for indexes, without looking at the partitioning problem. Database Cracking [19] dynamically sorts the data in column stores based on incoming queries. However, these works still do not address the online partitioning problem.

7 Conclusion

In this paper, we revisited database partitioning with the objective of automatically fitting data to queries to an online query workload. We presented AutoStore, an online self-tuning database store. AutoStore monitors the workload and takes partitioning decisions automatically. We generalized VPP and HPP to the 1DPP. We presented the O^2P algorithm to effectively solve 1DPP. We performed an extensive evaluation of our algorithms over TPC-H data. We showed experimental results from a main-memory and a BerkeleyDB implementations of AutoStore over mixed workloads. Our results show that O^2P is faster than earlier approaches by more than two orders of magnitude, and still produces good quality partitioning results. Additionally, our results show that over changing workloads AutoStore outperforms existing stores.
Acknowledgements. Work partially supported by DFG, M2CI.

References

1. S. Agarwal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*, 2004.
2. S. Agrawal, E. Chu, and V. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *SIGMOD*, 2006.
3. S. Agrawal et al. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *SIGMOD*, 2004.
4. I. Alagiannis et al. An Automated, Yet Interactive and Portable DB Designer. In *SIGMOD*, 2010.
5. N. Bruno and S. Chaudhuri. Physical Design Refinement: The "Merge-Reduce" Approach. In *EDBT*, pages 386–404, 2006.
6. N. Bruno et al. An Online Approach to Physical Design Tuning. In *ICDE*, 2007.
7. N. Bruno and S. Chaudhuri. Constrained Physical Design Tuning. *PVLDB*, 2008.
8. S. Chaudhuri and V. Narasayya. An Efficient Cost-driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, 1997.

9. W. W. Chu and I. T. Ieong. A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems. *IEEE TSE*, 19(8):804–812, 1993.
10. D. W. Cornell and P. S. Yu. A Vertical Partitioning Algorithm for Relational Databases. In *ICDE*, 1987.
11. D. W. Cornell and P. S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE TSE*, 16(2):248–258, 1990.
12. C. Curino et al. Schism: a Workload-Driven Approach to Database Replication and Partitioning. In *PVLDB*, 2010.
13. J. Dittrich and A. Jindal. Towards a One Size Fits All Database Architecture. In *CIDR*, 2011.
14. J.-P. Dittrich, P. M. Fischer, and D. Kossmann. AGILE: Adaptive Indexing for Context-aware Information Filters. In *SIGMOD*, 2005.
15. M. Grund et al. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 2010.
16. M. Hammer et al. A Heuristic Approach to Attribute Partitioning. *ACM TODS*, 1979.
17. R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, 2003.
18. J. A. Hoffer and D. G. Severance. The Use of Cluster Analysis in Physical Data Base Design. In *VLDB*, 1975.
19. S. Idreos et al. Database Cracking. In *CIDR*, 2007.
20. C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16:417–437, 10 2007.
21. A. Jindal. The Mimicking Octopus: Towards a one-size-fits-all Database Architecture. In *VLDB PhD Workshop*, 2010.
22. H. Kimura et al. CORADD: Correlation Aware Database Designer for Materialized Views and Indexes. In *VLDB*, 2010.
23. S. Navathe, et al.. Vertical Partitioning Algorithms for Database Design. *ACM TODS*, 1984.
24. S. Navathe and M. Ra. Vertical Partitioning for Database Design: A Graphical Algorithm. In *SIGMOD*, 1989.
25. P. E. O'Neil, et al. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 1996.
26. O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-Aware Storage Layout for Database Systems. In *SIGMOD*, 2010.
27. S. Papadomanolakis and A. Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM*, 2004.
28. V. Raman et al. Constant-Time Query Processing. In *ICDE*, 2008.
29. D. Sacca and G. Wiederhold. Database Partitioning in a Cluster of Processors. *ACM TODS*, 10(1):29–56, 1985.
30. K. Schnaitter et al. COLT: Continuous On-Line Database Tuning. In *SIGMOD*, 2006.
31. J. Zhou et al. Dynamic Materialized Views. In *ICDE*, 2007.
32. D. C. Zilio et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004.

A Physical Data Independence?

Theoretically, DBMSs claim to have a clear separation between the logical and the physical database design. However, this is not entirely true in practice. For instance, consider table partitioning in Oracle 11g [1]. Below is how a user specifies horizontal partitioning logically within the table schema:

```
create table BOOKSHELF_RANGE_PART
(Title  VARCHAR2 (100) primary key,
Publisher  VARCHAR2 (20),
CategoryName  VARCHAR2 (20),
Rating  VARCHAR2 (2),
constraint  CATFK2 foreign key (CategoryName)
references CATEGORY (CategoryName)
)
partition by range (CategoryName)
(partition PART1 values less than ('B') tablespace PART1_TS,
partition PART2 values less than (MAXVALUE)
        tablespace PART2_TS);
```

Here, the user specified two logical horizontal partitions (`PART1` and `PART2`) within the table schema. Thereafter the database system takes care of physically storing and accessing these partitions. However, such a logical definition is not possible for vertical partitioning. In Oracle 11g (and other databases), to partition a table vertically, database administrators need to create and load new tables – one for each partition – in the database store. To query the partitioned tables, application developers need to formulate join queries (or create views) over the different partitions. To change or modify the partitioning, the administrators again need to create and load new tables, and drop the old tables.

B System Details

Figure 8 shows the architecture of AutoStore.

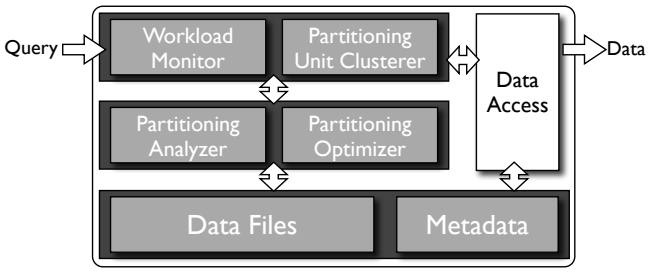


Fig. 8. Architecture of AutoStore

The workload monitors intercepts each incoming query and maintains a sliding query window. The partitioning unit clusterer reclusters the affinity matrix for each incoming as well as outgoing query in the sliding query window. AutoStore triggers the partitioning analyzer at regular intervals (`CheckpointSize`).

[1] K. Loney. Oracle Database 11g: The Complete Reference, McGraw-Hill, 2009

The partitioning analyzer comes up with the best partitioning scheme. The partitioning optimizer then evaluates the best partitioning scheme for its cost and benefit. The actual data is transformed only in case the partitioning optimizer decides to change the partitioning scheme to the new one. At all other times the data remains available for query processing.

C Algorithm Details

In this section, we provide the algorithm details for partitioning unit clustering (static and dynamic) and for the actual partitioning analysis (brute force, dead units pruning, greedy, amortized).

C.1 Partitioning Unit Clustering

Algorithm 2 shows a one-dimensional partitioning unit clustering algorithm to maximize the affinity measure. The algorithm places the first partitioning unit at the first place (Lines 1–5) and then for subsequent i^{th} unit it considers all possible i positions (Lines 6–16), and places the partitioning unit such that its contribution to global affinity measure is maximized (Line 17). This process continues till all partitioning units are placed (Lines 18–20). Note that this algorithm clusters a set of partitioning units statically.

Algorithm 2: `OneDCluster`

```
   Input  : ⪯', ⪯, i
   Output: Clustered one dimensional partitioning unit vector
 1 PartUnit newPartUnit = GetPartUnit(⪯, i);
 2 if i == 1 then
 3 |  PutPartUnit(⪯', newPartUnit, i);
 4 |  OneDCluster(⪯', ⪯, i+1);
 5 else
 6 |  contribution = 0;
 7 |  position = 0;
 8 |  for k=1 to i do
 9 |  |  PartUnit leftPartUnit = GetPartUnit(⪯', k-1);
10 |  |  PartUnit rightPartUnit = GetPartUnit(⪯', k+1);
11 |  |  contribution' = C(leftPartUnit, rightPartUnit, newPartUnit);
12 |  |  if contribution' > contribution then
13 |  |  |  contribution = contribution';
14 |  |  |  position = k;
15 |  |  end
16 |  end
17 |  PutPartUnit(⪯', newPartUnit, position);
18 |  if i < SizeOf(⪯) then
19 |  |  OneDCluster(⪯', ⪯, i+1);
20 |  end
21 end
```

Algorithm 3 shows the dynamic version of one dimensional partitioning unit clustering. Though it resembles the earlier Algorithm 2, there is one major difference: we only consider the partitioning units referenced by query q for shuffling,

since only they could have any change in their affinities. The first partitioning unit referenced in the query is kept at its original position (Lines 1-5) and for the i^{th} referenced unit we consider the left and right positions of the $(i-1)$ referenced units already placed (Lines 7–23). The position that offers maximum contribution to the global affinity measure is chosen (Line 26). The process continues till all referenced partitioned units are placed (Lines 26–28).

Algorithm 3: `DynamicOneDCluster`

```
   Input  : ⪯, ref(q), i
   Output: Dynamically clustered one dimensional partitioning unit vector
 1 PartUnit newPartUnit = GetPartUnit(ref(q), i);
 2 if i == 1 then
 3 |   l = GetPartUnitPos(⪯, newPart);
 4 |   PutPartUnit(⪯, newPartUnit, l);
 5 |   DynamicOneDCluster(⪯, ref(q), i+1);
 6 else
 7 |   contribution = 0;
 8 |   position = 0;
 9 |   for k=1 to i-1 do
10 |   |   PartUnit currPartUnit = GetPartUnit(ref(q), k);
11 |   |   l = GetPartUnitPos(⪯, currPartUnit);
12 |   |   PartUnit leftPartUnit = GetPartUnit(⪯, l-1);
13 |   |   PartUnit rightPartUnit = GetPartUnit(⪯, l+1);
14 |   |   contribution' = C(leftPartUnit, currPartUnit, newUnit);
15 |   |   position' = l;
16 |   |   if C(currPartUnit, rightPartUnit, newUnit) > contribution' then
17 |   |   |   contribution' = C(currPartUnit, rightPartUnit, newUnit);
18 |   |   |   position' = l+1;
19 |   |   end
20 |   |   if contribution' > contribution then
21 |   |   |   contribution = contribution';
22 |   |   |   position = position';
23 |   |   end
24 |   end
25 |   PutPartUnit(⪯, newPartUnit, position);
26 |   if i < SizeOf(ref(q)) then
27 |   |   DynamicOneDCluster(⪯, ref(q), i+1);
28 |   end
29 end
```

C.2 Partitioning Analysis

Brute Force. Algorithm 4 shows the brute force enumeration algorithm for partitioning analysis. We enumerate over each of the unseen positions in the split vector (Line 1) and set all combination of them (Lines 2-7). For each combination, we compare its cost with the minimum cost so far and set the best split vector accordingly (Lines 9-13). In Section 4 we show several improvements over the brute force partitioning analysis approach.

Dead Units Pruned. Algorithm 5 shows partitioning analysis with partitioning unit pruning (O^2Pp). We find the leading and the trailing partitioning units which are dead (not referenced by any query) and put them in separate partitions (Lines 3-27). For the remaining partitioning units we apply the same brute force approach of finding the best split lines (Lines 28-30).

Algorithm 4: `bfEnumerate`

```
Input  : S, x, y
Output: Enumerate over possible split vectors
1  for i=x to y do
2  |  for j=x to i-1 do
3  |  |  UnsetSplitLine(S, j);
4  |  end
5  |  if i < y then
6  |  |  SetSplitLine(S, i);
7  |  |  bfEnumerate(S, i+1, y);
8  |  else
9  |  |  Cost newCost = C_est.(W^N_{t_k}, P(S, ⪯));
10 |  |  if newCost < minCost then
11 |  |  |  bestSplitVector = S;
12 |  |  |  minCost = newCost;
13 |  |  end
14 |  end
15 end
```

Algorithm 5: `PrunedPartitioningUnitsAnalysis`

```
Input  : window, ⪯
Output: Find the best split vector
1  SplitVector bestSplitVector = UnsetAll(n-1);
2  q = 0;
3  for x = 1 to n-1 do
4  |  if GetPartUnit(⪯, x) ∉ ref(q) | ∀ q ∈ window then
5  |  |  UnsetSplitLine(bestSplitVector, x);
6  |  |  q = q+1;
7  |  else
8  |  |  break;
9  |  end
10 end
11 if q > 0 then
12 |  SetSplitLine(bestSplitVector, q);
13 end
14 r = 0;
15 for y = n to 1 do
16 |  if GetPartUnit(⪯, y) ∉ ref(q) | ∀ q ∈ window then
17 |  |  if y < n then
18 |  |  |  UnsetSplitLine(bestSplitVector, y);
19 |  |  end
20 |  |  r = r+1;
21 |  else
22 |  |  break;
23 |  end
24 end
25 if r > 0 then
26 |  SetSplitLine(bestSplitVector, n-r);
27 end
28 Cost minCost = MaxCost;
29 bfEnumerate(bestSplitVector, q+1, n-r);
30 return bestSplitVector;
```

Greedy. Algorithm 6 shows O^{2}Ppg algorithm. In each iteration, we find the best split line from among the unset lines (Lines 5-15). We continue this until the maximum possible partitioning is achieved (Lines 16-23). The best split vector is set to the one having lowest cost.

Amortized. Algorithm 7 shows partitioning analysis amortized over several queries (O^{2}Ppgda). AutoStore creates a snapshot of the query window when-

Algorithm 6: `greedyEnumerate`

```
   Input  : S, x, y
   Output: Enumerate over possible split vectors
 1 boolean maxPartitioned = true;
 2 Cost minLocalCost = MaxCost;
 3 SplitLine l = -1;
 4 for i=x to y do
 5 |   if not ContainsSplitLine(usedSplitLines, i) then
 6 |   |   SetSplitLine(S, i);
 7 |   |   Cost newLocalCost = C_est.(W^N_{t_k}, P(S, ⪯));
 8 |   |   if newLocalCost < minLocalCost then
 9 |   |   |   l = i;
10 |   |   |   minLocalCost = newLocalCost;
11 |   |   end
12 |   |   UnsetSplitLine(S, i);
13 |   |   maxPartitioned = false;
14 |   end
15 end
16 if not maxPartitioned then
17 |   SetSplitLine(S, l);
18 |   if minLocalCost < minCost then
19 |   |   bestSplitVector = S;
20 |   |   minCost = minLocalCost;
21 |   end
22 |   AddSplitLine(usedSplitLines, l);
23 |   greedyEnumerate(S, x, y);
24 end
```

Algorithm 7: `AmortizedPartitioningAnalysis`

```
   Input  : window, ⪯, iterations
   Output: Find the best split vector
 1 SplitVector bestSplitVector = UnsetAll(n-1);
 2 Cost minCost = MaxCost;
 3 for i = 1 to iterations do
 4 |   if moreItrs then
 5 |   |   moreItrs = amtzdGreedyEnumerate(bestSplitVector, 1, n);
 6 |   else
 7 |   |   return bestSplitVector;
 8 |   end
 9 end
10 return null;
```

ever it receives the trigger to compute partitioning scheme. The partitioning analyzer uses this query window snapshot to compute the partitioning scheme while normal query processing continues uninterrupted. Each time AutoStore triggers the partitioning analyzer, it performs a (partial) set of iterations (Line 3), if remaining in the current analysis (Lines 4-5). We return the best split vector if there are no more iterations remaining (Lines 6-9) and null otherwise (Line 10). Later, if we detect the partitioning to be viable, we can halt the query processing to make the actual layout transformation. The idea is to amortize the cost of computing the right partitioning scheme over several queries instead of blocking the query processing at regular intervals. Algorithm 8 shows amortized greedy enumeration which computes just one split line in each iteration (Lines 4-15). If there are more iterations remaining, we save the current split vector and return *true* indicating more iterations (Lines 16-20). Otherwise we return *false* (Lines 21-23).

Algorithm 8: `amortizedGreedyEnumerate`

```
Input  : S, x, y
Output: Enumerate over possible split vectors
boolean maxPartitioned = true;
Cost minLocalCost = MaxCost;
SplitLine l = -1; for i=x to y do
    if not ContainsSplitLine(usedSplitLines, i) then
        SetSplitLine(S, i);
        Cost newLocalCost = C_est.(window, P_t(S, ⪯));
        if newLocalCost < minLocalCost then
            l = i;
            minLocalCost = newLocalCost;
        end
        UnsetSplitLine(S, i);
        maxPartitioned = false;
    end
end
if not maxPartitioned then
    SetSplitLine(S, l);
    AddSplitLine(usedSplitLines, l);
    SaveState(S, x, y);
    return true;
else
    return false;
end
```

D Proofs

Proof of Theorem 1.

Proof. For each partition $p_{m,r}$, let $I_{m,r}$ be a (n-1) × (n-1) mask such that:

$$(I_{m,r})_{i,j} = \begin{cases} \delta_{i,j} & ; m \leq i \leq r, m \leq j \leq r \\ 0 & ; \text{otherwise} \end{cases}$$

Where, $\delta_{i,j}$ is the *kronecker delta*: $\delta_{i,j} = 1$, for $i = j$ and 0 otherwise. Note that the masks of all partitions in a partitioning scheme sum up to identity matrix, i.e. $\sum_{p_{m,r} \in P(S,\preceq)} I_{m,r} = I$. The best split line given in $(i+1)^{th}$ iteration of the greedy algorithm can be written as:

$$\begin{aligned}
s_{i+1}^{\min} &= \underset{s \in \text{unset}(S_i)}{\text{argmin}} \; C_{\text{exec}}(W, P((S_i + U(s)) \cdot I, \preceq)) \\
&= \underset{s \in \text{unset}(S_i)}{\text{argmin}} \; C_{\text{exec}}(W, P((S_i + U(s)) \cdot (\sum_{p_{m,r} \in P(S_i,\preceq)} I_{m,r}), \preceq)) \\
&= \underset{s \in [m,r], \forall p_{m,r} \in P(S_i,O)}{\text{argmin}} \; C_{\text{exec}}(W, P((S_i + U(s)) \cdot I_{m,r}, \preceq)) \\
&= \underset{\forall p_{m,r} \in P(S_i,\preceq)}{\text{argmin}} \left(\underset{s \in [m,r]}{\text{argmin}} \; C_{\text{exec}}(W, P((S_i + U(s)) \cdot I_{m,r}, \preceq)) \right) \\
&= \underset{\forall p_{m,r} \in P(S_i,\preceq)}{\text{argmin}} \; s_{i+1}^{\min, p_{m,r}}
\end{aligned}$$

In other words, to find the best split line we just need to compare the best split lines in each of the partitions so far. Hence, we can retain the best split lines from a partition for future comparison. However, if a partition is further split then we need to recompute best split lines in the two new parts created.

Proof of Theorem 2

Proof. Number of split lines considered for the first time are $(n-1)$ Number of split lines considered after the first split are $(n-2)$ Since each split line reduces the partitioning unit elements by z, total number of subsequent iterations are $(n-2-z)+(n-2-2z)+...+1$. Thus, the total number of iterations in O^{2}P are:

$$\begin{aligned}\#\text{Iterations} =& (n-2-z)+(n-2-2z)+...+1+(2n-3)\\ =& \frac{(n-3)\cdot(n-z-1)}{2z}+2n-3\end{aligned}$$

Proof of Lemma 1

Proof. In the worst case $z=1$, i.e. successive split lines reduce the partition units by 1. The number of iterations in this case is $\frac{n(n-1)}{2}$. Thus, complexity of O^{2}P in worst case is $O(n^2)$.

Proof of Lemma 2

Proof. Consider that each iteration in O^{2}P finds the best split line in the middle of the previous partition. i.e. $z=\frac{(n-2)}{2}$ for the first time after fixing the first two split lines, $z=\frac{(n-2)}{4}$ for the second time, and so on. Thus, the total number of iterations is $3n-6$. The complexity in the best case is $O(n)$.

E Cost Model

Tables 1 shows the cost models for attribute (vertical) partitioning units in a table with N rows. The costs depend on the partitions p_i in P containing at least one of the referenced attributes in A'. For each such partition p_i, we consider the costs for random and sequential I/O. Likewise, Figure 2 shows the cost model for range (horizontal) partitioning units in a table with M columns. The costs here now depend on the partitions p_i in P containing at least one of the referenced ranges in R'. We can see that the two cost models are very symmetric and could even be generalized into a single one.

Symbol	Meaning	Model
$C_{\text{scan}}(A',N,P)$	scan costs	$\sum_{p_i\in P_t,p_i\cap A'\neq\emptyset}\left\lceil\frac{N\cdot\sum_{p_i\in P,p_i\cap A'\neq\emptyset}\sum_{A_i\in p_i}colsize(A_i)}{m}\right\rceil\cdot C_{\text{random}}+$ $\sum_{p_i\in P,p_i\cap A'\neq\emptyset}\left\lceil\frac{N\cdot\sum_{A_i\in p_i}colsize(A_i)}{pageSize}\right\rceil/BW$
$C_{\text{index lookup}}(N)$	index lookup costs	$C_{\text{random}}\cdot\lceil log_{1+(pageSize-1)/2}(N\cdot(keySize+pointerSize)/pageSize)\rceil$
$C_{\text{cl. index scan}}(A',N,P)$	clustered index scan costs	$C_{\text{index lookup}}(N)+C_{\text{scan}}(A',\lceil sel\cdot N\rceil,P)$
$C_{\text{uncl. index scan}}(A',N,P)$	unclustered index scan costs	$C_{\text{index lookup}}+\lceil sel\cdot N\rceil\cdot\left(\sum_{p_i\in P,p_i\cap A'\neq\emptyset}1\right)\cdot(C_{\text{random}}+pageSize/BW)$

Table 1. Cost model for attribute partitioning units

Symbol	Meaning	Model
$C_{\text{scan}}(R', M, P)$	scan costs	$\sum_{p_i \in P, p_i \cap R' \neq \emptyset} \left\lceil \frac{M \cdot \sum_{p_i \in P, p_i \cap R' \neq \emptyset} \sum_{A_i \in p_i} numrows(R_i)}{m} \right\rceil \cdot C_{\text{random}} + \sum_{p_i \in P, p_i \cap R' \neq \emptyset} \left\lceil \frac{M \cdot \sum_{R_i \in p_i} numrows(R_i)}{pageSize} \right\rceil / BW$

Table 2. Cost model for range partitioning units

F Horizontal Partitioning Analysis

In this section we validate 1DPP approach in AutoStore, i.e. it treats vertical and horizontal partitioning equivalently. We just change the partitioning units from attributes to horizontal ranges. We vary the number of horizontal ranges (out of total 100 ranges) accessed on the x-axis, while the y-axis shows the total execution time of a workload of 100 queries. Figure 9 shows the comparison of AutoStore with No Partitioning and Full Horizontal (range) Partitioning. We can see that even though AutoStore starts with a No Partitioning configuration, it eventually matches the performance of full horizontal partitioning.

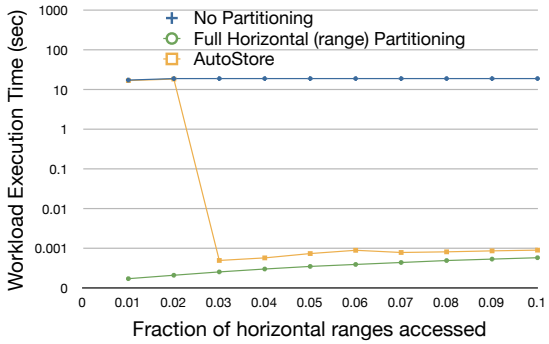


Fig. 9. Comparison of No Partitioning, Full Horizontal (range) Partitioning, and AutoStore in main-memory implementation.

G Extending Partitioning Analyzer to 2D

Now we shall show how our one dimensional online partitioning techniques can be extended to two dimensions i.e. both vertical and horizontal partitioning at the same time. We observe that partitioning analyzer is the only component in AutoStore which needs to be extended for 2D-partitioning. For that, first of all we will show some extension of our definitions. The partitioning units are now denoted as u_{ij} to indicate both horizontal and vertical dimension. The partitioning unit ordering $\preceq$ from Section 2.1, representing the relative positions of the partitioning units, is now extended to a two dimensional matrix:

$$\preceq = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ u_{21} & u_{22} & u_{33} & \dots & u_{2n} \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ u_{m1} & u_{m2} & u_{m3} & \dots & u_{mn} \end{bmatrix}$$

Here, we assume m partitions along the horizontal axis and n along the vertical axis. Similarly, the global affinity measure $M(\preceq)$ now takes into account neighboring partitioning units in two dimensions i.e. neighbors at left-right and at top-bottom. Hence, we adapt the measure from Section 2.1 to:

$$M(\preceq) = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} A(u_{ij},u_{kl})[A(u_{ij},u_{(k-1)l}) + A(u_{ij},u_{(k+1)l}) + A(u_{ij},u_{(k(l-1)}) + A(u_{ij},u_{k(l+1)})]$$

Consequently, the contribution to the affinity of placing partitioning unit u_{mn} between u_{ij} and u_{kl} in two dimensions is:

$$Cont(u_{ij},u_{kl},u_{mn}) = \sum_{x=1}^{m}\sum_{y=1}^{n}[A(u_{xy},u_{ij})\cdot A(u_{xy},u_{mn}) + A(u_{xy},u_{kl})\cdot A(u_{xy},u_{mn}) - A(u_{xy},u_{ij})\cdot A(u_{xy},u_{kl})]$$

The split lines from Section 2.2 are now four-valued as follows:

$$s_{jk} = \begin{cases} (1,1) & \text{if there are splits between } u_{jk}-u_{(j+1)k} \text{ and } u_{jk}-u_{j(k+1)} \\ (1,0) & \text{if there are splits between } u_{jk}-u_{(j+1)k} \\ (0,1) & \text{if there are splits between } u_{jk}-u_{j(k+1)} \\ (0,0) & \text{for no split} \end{cases}$$

Finally, the split vector from Section 2.2 is now is 2-d matrix:

$$S = \begin{bmatrix} s_{11} & s_{12} & s_{13} & \dots & s_{1(n-1)} \\ s_{21} & s_{22} & s_{33} & \dots & s_{2(n-1)} \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ s_{(m-1)1} & s_{(m-1)2} & s_{(m-1)3} & \dots & s_{(m-1)(n-1)} \end{bmatrix}$$

For such a split matrix, the partitioning analysis can try all possible values for all split lines in the brute force. This will have a complexity of $4^{(m-1)(n-1)}$. However, several of these partitioning schemes will not result in close bounded data partitions and hence would be invalid schemes. Several techniques from domain decomposition, grid-graph partitioning and tiling problems could be adapted to create valid partitioning schemes. However, we can also easily extend our greedy

and dynamic partitioning techniques from the single dimension to two dimensions. For greedy approach, we simply need to find the best (rectangle) cycle in the grid and successive iterations look for sub-cycles within existing ones. The worst case complexity would be of the order $\frac{m(m+1)\cdot n(n+1)}{4}$. For dynamic partition, we remember the best sub-cycles from previous iterations [2]. Detailed analysis of these techniques would be a part of future work.

[2] Alternatively, if we allow for non-rectangular partitions, instead of rectangular cycles we can also enumerate over all possible *polyomino* (i.e. any contiguous chunk of partitioning units) in each iteration. Several algorithms have been proposed to enumerate the polyominoes in a rectangle. Successive iterations look for sub-polyomino within existing ones. For dynamic partition, we remember the best sub-polyomino from previous iterations.