A Comparison of Knives for Bread Slicing

Alekh Jindal Endre Palatinus Vladimir Pavlov Jens Dittrich

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

ABSTRACT

Vertical partitioning is a crucial step in physical database design in row-oriented databases. A number of vertical partitioning algorithms have been proposed over the last three decades for a variety of niche scenarios. In principle, the underlying problem remains the same: decompose a table into one or more vertical partitions. However, it is not clear how good different vertical partitioning algorithms are in comparison to each other. In fact, it is not even clear how to experimentally compare different vertical partitioning algorithms. In this paper, we present an exhaustive experimental study of several vertical partitioning algorithms. We categorize vertical partitioning algorithms along three dimensions. We survey six vertical partitioning algorithms and discuss their pros and cons. We identify the major differences in the use-case settings for different algorithms and describe how to make an apples-to-apples comparison of different vertical partitioning algorithms under the same setting. We propose four metrics to compare vertical partitioning algorithms. We show experimental results from the TPC-H and SSB benchmark and present four key lessons learned: (1) we can do four orders of magnitude less computation and still find the optimal layouts, (2) the benefits of vertical partitioning depend strongly on the database buffer size, (3) HillClimb is the best vertical partitioning algorithm, and (4) vertical partitioning for TPC-H-like benchmarks can improve over column layout by only up to 5%.

1. INTRODUCTION

1.1 Background

Vertical partitioning is a physical design technique to partition a given logical relation into a set of physical tables. This is a common design step with analytical workloads in traditional as well as in modern data management systems such as HBase [9], Vertica [20], Hadoop++ [12], and HYRISE [6]. The basic purpose is to improve the I/O performance of disk-based systems. For instance, consider the TPC-H `PartSupp` table and the following query workload:

```
Q1: SELECT PartKey,SuppKey,AvailQty,SupplyCost
    FROM PartSupp;
Q2: SELECT AvailQty,SupplyCost,Comment
    FROM PartSupp;
```


Proceedings of the VLDB Endowment, Vol. 6, No. 6

For such a workload, we could choose to partition `PartSupp` into three vertical partitions: $\texttt{P}_1(\texttt{PartKey}, \texttt{SuppKey})$, $\texttt{P}_2(\texttt{AvailQty}, \texttt{SupplyCost})$, and $\texttt{P}_3(\texttt{Comment})$. Now Q_1 accesses partitions $\texttt{P}_1$ and $\texttt{P}_2$, while Q_2 accesses partitions $\texttt{P}_2$ and $\texttt{P}_3$. Thus, both Q_1 and Q_2 read only the required attributes and this improves the I/O performance of these queries. Vertical partitioning not only improves the query I/O performance, but also strongly affects other physical design decisions such as compression and indexing, as well as query processing techniques such as parallel and distributed query processing. Thus, it is no surprise that vertical partitioning has been researched extensively in the past with researchers proposing a plethora of approaches [10, 7, 15, 4, 16, 5, 3, 8, 17, 1, 6, 12, 11]. As special cases, two extremes of vertical partitioning are traditionally more popular, namely: (i) full vertical partitioning (i.e. column layouts) and (ii) no vertical partitioning (i.e. row layouts).

1.2 Effects of Vertical Partitioning

Let us now understand the effects of vertical partitioning on database design decisions in more detail. The major trade-off in vertical partitioning is the row size of partitions: large row sized (wide) partitions resemble row layout, while smaller row sized (narrow) partitions are more similar to column layout. Below, we briefly discuss the major pros and cons of vertical partitioning by contrasting the wide and narrow vertical partitions. Note that in this paper we are considering the majorly used row-oriented database systems and how to boost their performance using vertical partitioning. Of course, the other alternative could be using a different system, e.g. column store, in order to boost performance. However, replacing the existing, typically row-oriented, database system is not possible in many situations due to legacy reasons.

Bandwidth. The width of vertical partitions has a considerable effect on I/O bandwidth, and hence on query performance. Wide vertical partitions force the queries referencing fewer attributes to additionally read the accompanying attributes in the partition. For example, for queries Q_1 and Q_2 above, if we split `PartSupp` into the following two vertical partitions: $\texttt{P}_4(\texttt{PartKey}, \texttt{SuppKey}, \texttt{AvailQty}, \texttt{SupplyCost})$ and $\texttt{P}_5(\texttt{Comment})$ then query Q_2 is forced to read attributes `PartKey` and `Suppkey` in addition to `AvailQty`, `SupplyCost`, and `Comment`. These additional reads affect the I/O bandwidth of Q_2. In the extreme case, if all attributes are put together into a single vertical partition (which yields a row layout), then all except the referenced attributes are read unnecessarily.

Robustness. On the other hand, wide vertical partitions produce predictable query run times, because the majority of queries would have to touch the same number of partitions. For example, for queries Q_1 and Q_2 above, if we keep all attributes of table

PartSupp in a single vertical partition (i.e. row layout) then both queries Q_1 as well as Q_2 have the same I/O performance, since they both access all five attributes. Scan-only systems such as [18] are examples of such robust query processing systems.

Joins. Narrow vertical partitions penalize queries referencing lots of attributes. This is because the queries need to touch multiple vertical partitions. For example, for the workload in Section 1.1, if we split table PartSupp into three vertical partitions: P_1(PartKey, SuppKey), P_2(AvailQty, SupplyCost), and P_3(Comment) then query Q_2 must touch partitions P_2 and P_3. With this, the database engine needs to reconstruct the tuples from the referenced vertical partitions using tuple reconstruction joins. Since each vertical partition is stored as a separate physical table, these tuple reconstruction joins could be pretty expensive: they can negatively affect the query plans and incur CPU overheads.

Random I/O. Tuple reconstruction joins in narrow vertical partitions incur very high random I/O costs. This is because all referenced vertical partitions must be read at the same time for tuple reconstruction. For this to happen, the database buffer must be split into *sub-buffers* for each referenced vertical partition. As a consequence, now we have random I/Os each time any of the sub-buffers needs to be filled. For instance, Q_1 has twice the number of random I/Os for partitions P_1(PartKey, SuppKey) and P_2(AvailQty, SupplyCost) than for partition P_4(PartKey, SuppKey, AvailQty, SupplyCost).

1.3 Choosing a Vertical Partitioning Algorithm

Vertical partitioning strongly affects the query performance in many ways, as discussed above. A number of vertical partitioning algorithms have been proposed by several researchers over time [10, 7, 15, 4, 16, 5, 3, 8, 17, 1, 6, 12, 11]. As a result, users now have the problem of choosing a vertical partitioning algorithm. In contrast to physical design tools, which choose a layout given a vertical partitioning algorithm, the problem here is to choose the vertical partitioning algorithm in the first place. Essentially, the questions that we are looking at are:

- Which are the major algorithms proposed? What is the difference between those algorithms?
- For which settings were different algorithms proposed? What are their pros and cons?
- What are the primary differences between different vertical partitioning settings? Can we abstract the settings from the algorithms?
- How do we compare different algorithms in a common setting? What would be the right measures for comparison?
- How do the different algorithms compare against each other? When to use which algorithm?

Thus, there is an absence of a systematic and comparative study of vertical partitioning algorithms. This paper fills this gap.

1.4 Contributions

In this paper, we present an exhaustive experimental study on vertical partitioning algorithms. Our main contributions are as follows:

(1.) Given the large number of vertical partitioning algorithms proposed in the literature,we first understand the fundamental differences between them. To do so, we first classify them along three dimensions, namely: (i) search strategy, (ii) starting point, and (iii) candidate pruning (Section 2).

(2.) From the above categories, we survey six representative vertical partitioning algorithms, namely: (i) AutoPart [17], (ii) HillClimb [8], (iii) HYRISE [6], (iv) Navathe's algorithm [15], (v) O_2P [11], and (vi) Trojan layouts [12]. We present a brief summary and the context of each of the algorithms (Section 3).

(3.) We describe how the different vertical partitioning algorithms can be applied in the same setting. Even though each algorithm was proposed for a different setting, we can still unite them under a common umbrella (Section 4).

(4.) We present a systematic way of comparing different vertical partitioning algorithms. For this purpose, we introduce four metrics, namely: (i) *how fast* in terms of computation times, (ii) *how good* in terms of workload runtimes, (iii) *how fragile* in terms of predictable runtimes, and (iv) *where does it make sense* to use vertical partitioning (Section 5).

(5.) We show detailed experimental results from six vertical partitioning algorithms over the TPC-H benchmark and with row and column layouts as baselines. We discuss each of the four metrics for the six vertical partitioning algorithms (Section 6).

(6.) Finally, we discuss the 4 key lessons learned (Section 7).

2. CLASSIFICATION OF VERTICAL PARTITIONING ALGORITHMS

There are several vertical partitioning algorithms proposed in the literature. Instead of simply listing them, it would be more interesting to see the major differences between the core ideas of those algorithms. To do this, we categorize the vertical partitioning algorithms along three dimensions based on the way they attack the vertical partitioning problem. Table 1 shows the classification of the evaluated vertical partitioning algorithms. We describe each of these dimensions and categories below.

2.1 Search Strategy

First of all, we differentiate different vertical partitioning algorithms based on their search strategy in the solution space.

Brute Force. Algorithms in this category follow the naive approach of enumerating all possible vertical partitionings and picking the one giving the best estimated query performance. In this way, a brute force algorithm computes the best possible vertical partitioning over a given query workload and cost model. Unfortunately, the number of possible vertical partitionings grow dramatically with the number of attributes. For instance, for the 16 attributes of the TPC-H Lineitem table, the number of possible vertical partitionings is 10.5 million. Therefore, brute force is not a practical approach for large number of attributes.

Top-down. Algorithms in this category start from the set containing all attributes and try to break it into smaller and smaller subsets. The idea is to assume *no-vertical-partitioning*, i.e. row layout, as the ground truth and to improve upon it as much as possible. The improvement is usually measured in terms of the expected cost of a query workload (using a cost model). Early vertical partitioning algorithms [15, 16] were based on this approach. As the starting point the attributes are arranged in some *order*, e.g. an ordered sequence in [15] or a connected graph in [16]. Typically, there is a preparatory step which determines this order, e.g. attribute affinity matrix clustering in [15]. Thereafter, the attribute set is recursively (and greedily) divided into smaller subsets till no improvement in the expected workload costs is seen. Every split step preserves the initial ordering of the attributes. Inspired from those early works, a recent algorithm does online vertical partitioning using the top-down approach [11]. The vertical partitioning algorithms in top-down category converge faster for highly regular attribute access

Dimension	Category	AutoPart [17]	HillClimb [8]	HYRISE [6]	Navathe [15]	O_2P [11]	Trojan [12]	Brute Force
Search Strategy	Brute Force							■
	Top-down				■	■		
	Bottom-up	■	■	■			■	
Starting Point	Whole workload				■	■		■
	Attribute subset	■	■	■				
	Query subset						■	
Candidate Pruning	No pruning	■	■	■	■	■		■
	Threshold-based						■	

Table 1: Classification of the evaluated vertical partitioning algorithms.

patterns, i.e. lots of queries accessing almost the same attributes. This is because only few splitting steps are required. On the other hand, top-down algorithms consider vertical partitions incrementally. This means that for any vertical partition to appear in the final solution, its supersets must appear in all previous iterations. This might not happen in many situations.

Bottom-up. In contrast to top-down, the bottom-up approach starts with minimally small vertical partitions. All algorithms in this category define the latter property of a partition differently. The underlying assumption is that it does not make sense to sub-divide these initial vertical partitions into smaller vertical partitions. The idea then is to recursively merge the vertical partitions into bigger partitions as long as there is an improvement in expected query costs. Three main algorithms [3, 8, 17] fall into this category. As the preliminary step, the algorithms produce the set of minimally small vertical partitions. These can be partitions containing only a single attribute (column layout), as in [8], or the set of *primary partitions*, which are partitions containing attributes that are always accessed together in all queries, as in [3, 17]. Thereafter, the algorithms recursively consider merging two or more partitions. Additionally, [17] also creates overlapping partitions, i.e. partitions having one or more attributes in common, thereby allowing for partial replication of attributes. The bottom-up algorithms converge faster for highly fragmented attribute access patterns, i.e. queries accessing little or no attributes in common. This is because after a few merge steps the costs will not improve any more. Similar to the top-down class, the bottom-up algorithms consider vertical partitions incrementally, i.e. greedily. For bottom-up algorithms this means that for any vertical partition to appear in the final solution, its subsets must appear in all previous iterations.

2.2 Starting Point

Apart from the search strategy, different vertical partitioning algorithms may have different starting points. For example, an algorithm may start with only a subset of the attributes or with only a subset of the workload queries. This is an important consideration because it helps to first sub-divide the vertical partitioning problem into smaller problems and find the solution to each of them.

Whole workload. Algorithms in this category do neither divide the queries nor the attributes at the start.

Attribute subset. Algorithms in this category compute vertical partitioning for a subset of the attributes. For example, [6] first sub-divides attribute sets into groups using a k-way partitioner and then computes the vertical partitioning for each group using a top-down algorithm. Finally, to produce the final solution, [6] combines the solutions from different sub-problems. Computing vertical partitioning for attribute subsets reduces the complexity of the algorithm dramatically. However, such algorithms find the solution for each subset locally and have to later merge them.

Query subset. Algorithms in this category compute vertical partitioning for only a subset of the queries in the workload. For example, [12] first sub-divides the workload into query groups depending on the similarity between queries and finds the layout for each query group using a bottom-up algorithm. It is easier to find vertical partitioning for query subsets, since they are likely to have more similar access patterns, and hence the algorithm converges quickly. [12] does not combine the solutions from different query subsets, as it creates multiple vertical partitionings, one for each dataset replica. Starting from query subsets is a very practical approach because typical workloads contain several classes of queries, each having very similar access patterns.

2.3 Candidate Pruning

Finally, vertical partitioning algorithms may also prune the vertical partitioning candidates in order to reduce the search space.

No pruning. Most algorithms considered in this paper do not apply pruning to the search space, but generate possible solutions in each iteration excluding locally sub-optimal ones.

Threshold-based. Algorithms with threshold-based pruning prune the input set based on some heuristics. For example, algorithms [1] and [12] prune the set of column groups based on their interestingness, which denotes how well a given column group speeds up the queries. The complexity of these algorithms therefore depends on the effectiveness of their pruning threshold. Threshold-based pruning algorithms face one basic problem: the algorithm needs to generate all candidates before actually pruning them. This could be pretty expensive and hence slow. On the flip side, however, threshold-pruning approach sees the global picture (not local or incremental) and hence is expected to produce better results.

3. EVALUATED ALGORITHMS

In this paper, we cover a wide range of representative vertical partitioning algorithms from the early state-of-the-art to the most recent ones. We choose these algorithms to cover all categories and include the earliest vertical partitioning algorithm as well as five other recent vertical partitioning algorithms published in the last decade. Below we describe each of these algorithms.

Brute Force. The total number of vertical partitioning combinations, using brute force, are given by *Bell numbers*. The n^{th} Bell number B_{n+1} is given as: $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$. For example, for the TPC-H customer table, having eight attributes, the number of possible vertical partitionings is given by $B_8 = 4140$. Bell numbers can be represented as a sum of *Sterling numbers*[1]: $B_{n+1} = \sum_{k=0}^{n} \left\{ {n \atop k} \right\}$ where

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \cdot \left\{ {n-1 \atop k} \right\}, \text{and} \left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1.$$

The complexity of the above brute force vertical partitioning algorithm is $O(n^n)$ (for n attributes).

Navathe. One of the earliest approximation-based approaches to vertical partitioning was proposed by Navathe et al [15]. This is a top-down algorithm and focuses primarily on disk-based systems.

[1]The Sterling number $\left\{ {n \atop k} \right\}$ gives the number of ways to partition n attributes into k partitions.

The core idea of this algorithm is as follows. Given a set of attributes and a set of queries referencing those attributes, the algorithm constructs an *attribute affinity matrix*. Cell (i, j) of the attribute affinity matrix denotes the number of times attribute i co-occurs with attribute j (also called their *affinity*). Thereafter, the algorithm clusters the cells of the matrix such that attributes with higher affinity are close together. The authors propose to use the bond energy algorithm [14] for matrix clustering. After that, the algorithm splits the clustered set of attributes into vertical partitions recursively.

HillClimb. The HillClimb algorithm is a bottom-up algorithm proposed in the early 2000s [8]. This algorithm focuses on data layouts within a data page. It proceeds as follows. It starts with column layout, i.e. each attribute resides in a different vertical partition. Thereafter, in each iteration, the algorithm finds and merges two partitions which, when merged, provide the best improvement in expected query costs. This means that in each iteration the number of vertical partitions is reduced by one. The algorithm stops iterating when there is no improvement in expected query costs. To facilitate computing the expected query costs, the algorithm precomputes and maintains a dictionary of the costs of all possible vertical partitions (or column groups). However, the size of such a dictionary grows quickly to several gigabytes for a large number of attributes. As a result, we have found that the runtime of the algorithm can be dramatically improved without maintaining such a dictionary. Thus, we used this improved version of HillClimb.

AutoPart. The AutoPart is a bottom-up algorithm introduced in 2004 to compute vertical partitionings over large scientific datasets [17]. First, AutoPart *categorically* partitions the table horizontally (based on selection predicates), such that each horizontal partition is accessed by a different subset of queries. Thereafter, AutoPart finds vertical partitioning for each horizontal partition. As a starting point, AutoPart generates the set of primary partitions (called *atomic fragments*). A vertical partition is atomic if all queries accessing it, reference all attributes in the partition. In other words, there are no queries which access a subset of an atomic fragment. Thereafter, in each iteration, the fragments are extended by either combining them with atomic fragments or with fragments from the previous iteration. The process is repeated till there is no improvement in estimated costs of the query workload. Note that an attribute may occur in multiple fragments (i.e. replicated) when combined. Thus, it might be possible that multiple partition combinations are now suitable to answer a given query. In such a case, we need to select the partitions to read. It turns out that partition selection is as difficult a problem as vertical partitioning itself.

HYRISE. The HYRISE is a multi-level algorithm proposed in 2010 to compute vertical partitionings for main-memory resident data processing systems [6]. In contrast to disk-based systems, the goal here is to minimize the number of cache misses. In the first step, the algorithm generates the set of *primary partitions*, which are the same as the atomic fragments in AutoPart, i.e. sets of attributes that are always accessed together. Then, the algorithm builds an affinity graph for the primary partitions, where primary partitions are represented as nodes and the co-accessed frequency of two primary partitions as edge weights. HYRISE then partitions this graphs such that each sub-graph contains at most K primary partitions. This is done using a K-way graph partitioner. Thereafter, HYRISE finds the layout for each sub-graph separately. In each iteration, the primary partitions (belonging to the same sub-graph) which give the maximum cost improvement are merged. The merged partition replaces the primary partitions and the process is repeated until there is no more improvement in cost. As the final step, HYRISE tries to combine the vertical partitions obtained from different sub-graphs.

O_2P. One-dimensional online partitioning (O_2P) is a top-down algorithm proposed in 2011 with the focus on real time partitioning [11]. The goal is to determine a vertical partitioning in an online setting, i.e. *while* the query workload is being executed. It starts from Navathe's algorithm and transforms it into an online vertical partitioning algorithm. To do so, it dynamically updates as well as clusters the affinity matrix for each incoming query. This is done by adapting the bond energy algorithm [14] , used in Navathe, to an online setting. To compute the vertical partitioning, O_2P employs a greedy approach to create one (the best) new vertical partition in each step. It also uses dynamic programming to remember the costs of non-best vertical partitions from the previous step. These two techniques make the partitioning analysis in O_2P extremely fast and hence suited for an online setting.

Trojan. The Trojan layouts algorithm was proposed in 2011 to create vertical partitioning for big data [12]. It is a threshold-pruning based algorithm. Unlike previous algorithms, it considers large data block sizes and existing data block replication, both being a reality for big data. As the first step it enumerates all possible column groups and keeps only the ones that are *interesting*. It introduces a novel interestingness measure for column groups, based on the mutual information between the attributes of a column group. The algorithm prunes all column groups whose interestingness fall below a certain threshold. The interesting column groups are then merged into a *complete* (i.e. containing all attributes) and *disjoint* (i.e. not containing any attribute twice) set of vertical partitions. This is done by mapping vertical partitioning to a 0-1 knapsack problem. The Trojan algorithm works especially well with data replication, such as found in HDFS. To take into account the default data replication in HDFS, it first groups queries and maps each query group to a different data replica. It uses the same column grouping algorithm for query grouping as well. Then, for each query group, it computes the column groups independently.

4. METHODOLOGY

The vertical partitioning algorithms described above have all been proposed for different scenarios and under different settings. Below, let us try to understand the major differences between them.

(1.) *Granularity.* Different algorithms are targeted for different data granularity, such as data page, e.g. HillClimb, database block, e.g. Trojan, and file, e.g. AutoPart.
(2.) *Hardware.* The algorithms can optimize for different hardware, such as hard disk, e.g. Navathe, and main-memory, e.g. HYRISE.
(3.) *Workload.* The algorithms may work with different assumptions for the query workload. We can consider a fixed set of queries (offline workload), e.g. AutoPart, or a dynamically growing set of queries (online workload), e.g. O_2P.
(4.) *Replication.* An algorithm may or may not consider data replication. Even if the algorithm considers replication, it may either consider replicating all attributes (full replication), e.g. Trojan, or only a subset of the attributes (partial replication), e.g. AutoPart.
(5.) *System.* Different algorithms are proposed in different implementations of data managing systems, e.g. Hadoop (Trojan), BerkeleyDB (O_2P), main-memory implementation (HYRISE), etc.

Table 2 classifies the six algorithms (from Section 3) analyzed in this paper according to their granularity, hardware-, workload-, and replication characteristics. We can see that no two algorithms have

Parameters	Values	AutoPart [17]	HillClimb [8]	HYRISE [6]	Navathe [15]	O_2P [11]	Trojan [12]	**Our Unified Setting**
Granularity	DATA PAGE		■					
	DATABASE BLOCK						■	
	FILE	■		■	■	■		■
Hardware	HARD DISK	■	■		■	■	■	
	MAIN MEMORY			■				
Workload	OFFLINE	■	■	■	■		■	■
	ONLINE					■		
Replication	PARTIAL	■						
	FULL						■	
	NONE		■	■	■	■		■
System	OPEN SOURCE					■	■	
	COST MODEL	■			■			■
	CUSTOM		■	■				

Table 2: Settings for different vertical partitioning algorithms.

the same combination of these parameters. It seems that the different vertical partitioning algorithms use quite different configurations even though they have exactly the same underlying functionality: decompose a table into vertical partitions. In order to have an apples-to-apples comparison, we use the same configuration for all vertical partitioning algorithms. To the best of our knowledge, this is the first paper to survey vertical partitioning algorithms under a common configuration. Essentially this means that we strip the granularity, hardware-, workload-, and replication characteristics from the different vertical partitioning algorithms, leaving just the core vertical partitioning functionality. Below we describe the common configuration used in our experiments.

Common Granularity. For all algorithms, we consider the storage layout of the vertically partitioned table to be as follows: the table is split into one or more vertical partitions (column groups), which are stored in separate files. Thus, each data page contains data from only a single vertical partition. At query time, we assume that the database system does the following: read all vertical partition's files which contain any of the attributes referenced by the incoming query. This means that even if a query references only some of the attributes in a vertical partition, we still need to read all attributes in the vertical partition's file.

Common Hardware. We use the following common testbed for all algorithms: a single node machine with a quad-core Intel Xeon 5150 processor running at 2.66 GHz with 4 MB L2 cache, having 16 GB RAM and 1.5 TB HDD, running OpenSuse 12.1 64 bit. We consider the commonly used disk-based storage when evaluating vertical partitioning algorithms. We measured the disk characteristics of our testbed using Bonnie++ [2]. We obtained the following results: a disk read bandwidth of 90.07 MB/s, disk write bandwidth of 64.37 MB/s and average disk seek time of 4.84 ms.

Common Workload. We consider read-only analytical applications for comparing different vertical partitioning algorithms. To do so, we take the query workload from the widely used TPC-H benchmark, and assume a scale factor of 10. We partition each table in TPC-H separately, as done by other researchers [3]. We take all 22 queries from the TPC-H benchmark. However, we consider only scan and projection query operators. This is because in our cost model, we model only the I/O costs for accessing the data, while excluding the query execution costs. We do this for two reasons. First, almost all vertical partitioning algorithms consider only scan and projection operators. Since we are doing a comparative study of different algorithms, we consider the same set of operators for all algorithms. Extending these algorithms to consider other operators, such as selection, will be an interesting future work. Second, the overall query execution costs make sense only when all physical design decisions, including indexes and materialised views, are considered. Instead, in this paper, we are focussing on vertical partitioning and hence we want to isolate the impact of vertical partitioning created by different algorithms. Furthermore, overall query execution costs depend heavily on the query optimizer and executor of the database system and hence it is not possible to model them in a general setting.

Common Replication. AutoPart and Trojan make use of partial and full data replication respectively. However, in order to make a fair comparison, we would need to tweak other algorithms to allow for data replication as well. Moreover, data replication adds several new dimensions for consideration. These include storage space constraints, read versus update performances, and most importantly picking the right replica at query time. Hence, we believe that vertical partitioning with data replication requires an independent exhaustive study, which is beyond the scope of this paper. Instead, in this paper, we limit to no data replication.

Common System. We evaluate all algorithms using the estimated costs from our I/O cost model. We do this for two reasons. First, as discussed before, we focus on the I/O costs of queries in order to understand the effects of vertical partitioning in row-oriented database systems. Second, database systems typically create a different table for each vertical partition and later use joins for tuple reconstruction. This makes running just the leaf plans (in order to see the I/O costs) very expensive because no operators can be pushed down and we end up with high join cardinalities. As a result, the I/O costs are overshadowed by the join processing costs. In our recent work [13], we exploited UDFs to store and access data in column layouts without performing a join, i.e. we simply *merge* the columns. However, this works only for highly selective queries[2]. To the best of our knowledge, there is no freely available database system which queries vertically partitioned data without performing table joins.

Our system assumes buffered read- and write mechanisms for transferring data between disk and memory. This means that, at query time, we read all vertical partitions which contain any of the attributes, referenced by the incoming query, into an I/O buffer (say of size *Buff*). In our experiments we assume per-tuple query processing, i.e. the database system passes data tuple-by-tuple to the query executor. For this, the database system needs to reconstruct the tuples *while* reading the vertical partitions. To do so, we will require to buffer-read the vertical partitions at the same time. This means that we have to share the I/O buffer among the multiple vertical partitions being read. In our cost model, we share the I/O buffer in proportion to the tuple size of the vertical partitions being read. If S is the total row size of all referenced partitions and s_i is the row size of vertical partition i, then the I/O buffer allocated to partition i is given as:

$$\text{buff}_i = \left\lfloor \text{Buff} \cdot \frac{\text{s}_i}{\text{S}} \right\rfloor .$$

Given block size b, the number of blocks that can be read at a time into the buffer for partition i are:

$$\text{blocks}_i^{\text{buff}} = \left\lfloor \frac{\text{buff}_i}{\text{b}} \right\rfloor .$$

[2]For low selectivities the UDF call overhead shadows the performance gain due to a different layout.

If the table has N rows, the total number of blocks on disk for partition i are:

$$\text{blocks}_i = \left\lceil \frac{\text{N}}{\left\lfloor \frac{\text{b}}{\text{s}_i} \right\rfloor} \right\rceil .$$

Assume that we have to perform a seek every time the I/O buffer for partition i needs to be filled. Then the number of times the I/O buffer gets full determines the seek cost of reading partition i. Given an average seek time t_s of the disk, the seek cost of reading partition i is given as:

$$\text{cost}_i^{\text{seek}} = t_s \cdot \left\lceil \frac{\text{blocks}_i}{\text{blocks}_i^{\text{buff}}} \right\rceil .$$

On the other hand, the scan cost of partition i is determined by the total number of blocks of partition i to be read. Given disk bandwidth *BW*, the scan cost of partition i is given as:

$$\text{cost}_i^{\text{scan}} = \frac{\text{blocks}_i \cdot \text{b}}{BW}.$$

Finally, for a query Q referencing a P_Q set of vertical partitions, the total I/O cost is the sum of the seek- and scan costs of all referenced partitions:

$$\text{cost}_Q = \sum_{i \in P_Q} \left(\text{cost}_i^{\text{seek}} + \text{cost}_i^{\text{scan}} \right).$$

The total I/O costs of the entire workload will be the sum of the I/O costs of each query in the workload.

5. COMPARISON METRICS

As discussed in the previous section, we apply the same setting to all vertical partitioning algorithms. However, since there is no prior work comparing different vertical partitioning algorithms, it is not clear how to compare them, i.e. the comparison metrics are not defined. The authors of HYRISE compared their algorithm against HillClimb in terms of query costs. However, we believe that other measures such as time taken to compute the layouts are equally important. Thus, in this section, we systematically introduce four comparison metrics for vertical partitioning algorithms and describe them below.

How fast? Vertical partitioning being an NP-hard problem, the first thing that comes to mind is *how fast* is a given algorithm, i.e. how long does it take to come up with a solution. Additionally, the optimization time should be seen in comparison to the table size (or indirectly the layout creation time). For example, if it takes fifteen minutes to create the layouts (i.e., a large table) then it might be acceptable to spend an hour to find the layouts.

How good? Since the goal of a vertical partitioning algorithm is to improve the workload runtime, it is important to know the expected workload runtime. Additionally, it is important to know how much does vertical partitioning improve the workload runtime over row and column layouts. Note that this improvement comes at a price: we need to invest in the optimization and the creation time. Thus, we need to see the time invested (optimization + creation) compared to the expected workload execution cost benefits.

In fact, the ratio of these two quantities gives the fraction (or the multiple) of query workload that we need to execute before the time invested pays off over the workload runtime improvements.

How fragile? Heterogenous hardwares/software settings are common in data centers these days. However, vertical partitioning algorithms can be computationally expensive, therefore it is not possible to recompute them for each and every hardware/software setting. Thus, we need to know how fragile the different vertical partitioning algorithms are over different parameters in the cost model (which models the hardware/software settings). We measure algorithm fragility as the change in workload runtime when there is a change in a cost model parameter. Fragility, thus defined, gives hints on whether or not we should re-run the vertical partitioning algorithm if the hardware/software settings change.

Where does it make sense? The fragility metric above measures how far off is the workload performance, if we optimize vertical partitioning for one cost model and use it over another. However, at the same time, it is also important to know how does the workload performance change if we re-optimize vertical partitioning over different cost models. Thus, we optimize for each new cost model parameter and show the workload performance. This helps us to find the sweet spots for vertical partitioning, i.e. the cost model parameters for which vertical partitioning makes the most sense.

6. SIMULATIONS AND EXPERIMENTS

We now present the results from the six vertical partitioning algorithms considered in Section 3. We implemented all algorithms in Java 6 and tried to keep the implementations as close to the original descriptions as possible. However, we did adapt the algorithms to the unified settings shown in Table 2. For example, Trojan was adapted to work without considering data replication. We ran all experiments on the common hardware described in Section 4. We organize the results along the four comparison metrics introduced in Section 5. We repeated each measurement five times and report the average. We discarded the results of the first five runs to allow for just-in-time compilation in the JVM to complete and use the results of the second five runs. We used cold caches, both for the operating system as well as the hard disk, for all runs.

6.1 Comparing Optimization Time

In this section we address the following questions:

How do the algorithms compare in terms of optimization time?

Figure 1 shows the optimization times for different vertical partitioning algorithms. We can see that the fastest algorithm (O_2P) is

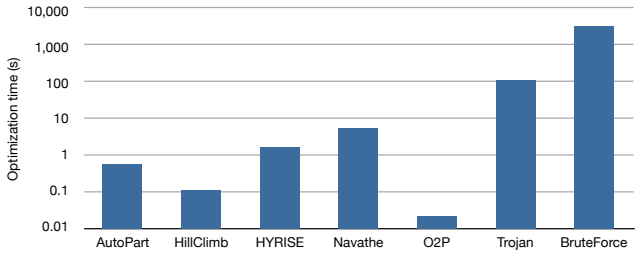


Figure 1: Optimization time for different algorithms

5 orders of magnitude faster than BruteForce. Even the slowest algorithm (Trojan) is 3 orders of magnitude faster than BruteForce. Thus, all algorithms find a vertical partitioning solution much faster than BruteForce. The optimization times of AutoPart, HillClimb, HYRISE, Navathe, and O_2P are quite acceptable (at most 5 seconds), however, Trojan and BruteForce have very high optimization times (1.5 minutes and 1 hour, respectively). The time to transform from row layout to vertically partitioned layout for scale factor 10 is around 420 seconds for all algorithms. This means that it takes much longer to transform the layout than it takes to compute the layout.

How do the optimization times change with the workload size?

Let us now see how the optimization times change with the workload size. Recall that, for every vertical partitioning candidate, an algorithm computes the expected cost of each query in the query

workload. Thus, we expected higher optimization time for larger query workloads. Figure 2 shows the optimization times of the different algorithms over varying workload size. We vary the TPC-H workload size by taking the first k queries, k varying from 1 to 22. We can see that Navathe and AutoPart have a much steeper in-

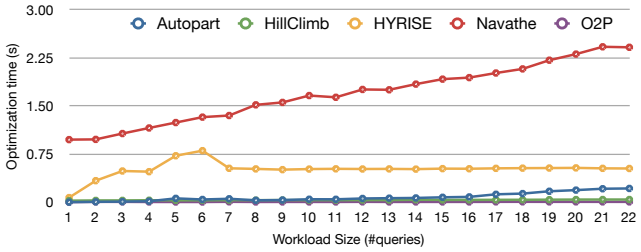


Figure 2: Optimization time over varying workload size

crease in optimization time in comparison to HYRISE, HillClimb, and O_2P. In general, these algorithms scale well with the workload size. We have excluded Trojan and BruteForce in the figure because of their extremely high optimization time (at least 2 orders of magnitude higher than the others), which distorts the graph.

The most important lesson learned in this section is that the optimization time of a vertical partitioning algorithm can be several orders of magnitude less than BruteForce. Still, as we will see in the next section, some algorithms can find the same (optimal) solution as the BruteForce.

6.2 Comparing Algorithm Quality

We investigate a series of five questions in this section. Let's start with the following one:

How do algorithms compare in terms of query performance?

Figure 3 shows the *estimated workload costs* for all queries of the TPC-H Benchmark – when using the partitionings produced by the different vertical partitioning algorithms. By estimated workload cost we mean the total I/O cost of the entire workload as described in Section 4. We can see that except for Navathe and O_2P, all al-

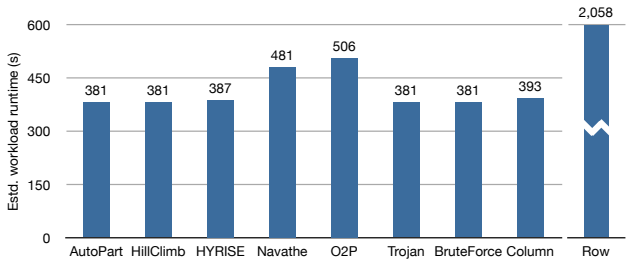


Figure 3: Estimated workload runtime for different algorithms

gorithms, including BruteForce, have very similar estimated workload costs. In fact, the layouts produced by AutoPart, HillCimb and Trojan have exactly the same workload cost as that by BruteForce. This is despite HillClimb requiring 5 orders of magnitude less optimization time than BruteForce. As a result, vertical partitioning with HillClimb can payoff for as little as 25% of the TPC-H workload (See Appendix A.1 for details).

Now let us analyze the improvement of vertical partitioning over Row or Column. We can see that the improvement over Row is as high as 80.11%. However, over Column the maximum improvement is only 4.75%. Column even outperforms the vertically partitioned layouts of Navathe and O_2P by 21% and 28%, respectively. This is a surprising result because we expected vertical partitioning to be very effective for analytical workloads. Let us now dig deeper to understand the high improvements over Row and low improvements over Column, by asking the following questions.

What fraction of the data read is unnecessary?

Note that a suitable vertical partitioning improves over Row because it reads less unnecessary data. Figure 4 shows the percentage of data read which is unnecessary, i.e. not needed by the queries.

$$\text{Unnecessary data read} = \frac{\text{Data read} - \text{Data needed}}{\text{Data read}} * 100\%$$

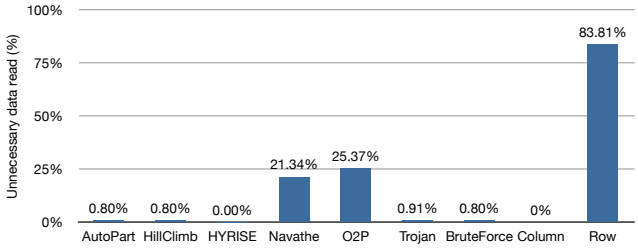


Figure 4: Fraction of unnecessary data read

We can see that Row reads 84% unnecessary data and all vertically partitioned layouts have a significant improvement over that. The layouts produced by AutoPart, HillClimb, and BruteForce read only 0.8% unnecessary data, while the layouts from HYRISE do not read *any* unnecessary data. This explains the dramatic improvements over row.

How many tuple reconstruction joins are performed?

Next, let us understand the low improvements of vertical partitioning over Column. Note that a suitable vertical partitioning improves over Column since it performs less tuple reconstruction joins. For each query, the number of tuple reconstruction joins per tuple are given as:

$$\text{\#Tuple-reconstruction joins} = \text{\#Vertical partitions accessed} - 1$$

Figure 5 shows the tuple reconstruction joins averaged over all tuples and all queries, when using each of the layouts. Column has to

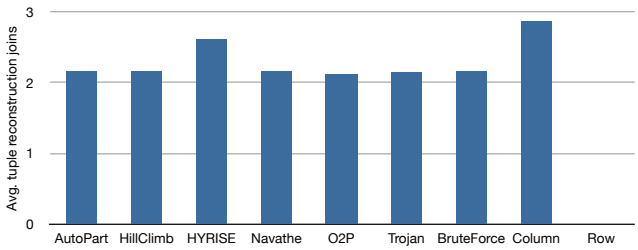


Figure 5: Average tuple reconstruction joins

join all attributes referenced by the query. However, vertically partitioned layouts also perform at least 72% of the joins performed by Column. Thus, none of the algorithms produce layouts which would dramatically reduce the tuple reconstruction joins, which increases the number of random I/Os in our cost model, hence the marginal improvement over Column. Note that the above estimated improvements are only in terms of I/O costs. In practice, tuple-reconstruction incurs additional CPU-costs as well.

How far is vertical partitioning from perfect materialized views?

We see above that the layouts produced by the vertical partitioning algorithms improve marginally over Column. This is in spite

of almost all algorithms having estimated costs very close to the BruteForce, which produces optimal layout (See Figure 3). Let us now see how far are the vertical partitioning layouts from perfect materialized views — a vertical partition, created for each query, containing exactly the attributes referenced by that query. Figure 6 shows the distance of each of the layout from the perfect materialized views (PMV).

$$\text{Distance from PMV} = \frac{\text{Est. costs of layout} - \text{Est. costs of PMV}}{\text{Est. costs of PMV}} * 100\%$$

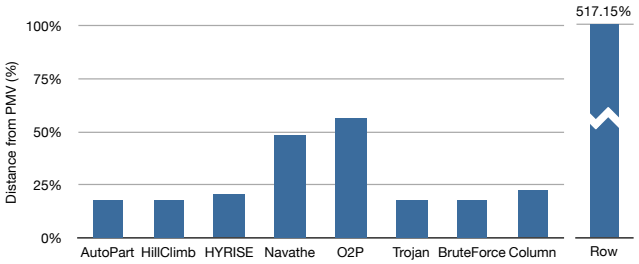


Figure 6: Distance from perfect materialized views

We can see that while Navathe and O_2P are 49% and 56% off from the perfect materialized views, respectively, HillClimb and AutoPart are as low as just 18% off from it. This is in spite of perfect materialized views needing much more storage space.

What is the effect of workload size on query performance?

We saw above that vertically partitioned layouts are up to 56% off from the perfect materialized views. Perfect materialized views and vertically partitioned layouts are two extremes: creating vertical partitions for *each* query versus creating vertical partitions for the *entire* query workload. Let us now see how the query performance changes in the middle.

In this experiment, we start from the perfect materialized views and gradually increase the workload size k (from 1 to 22). For each workload size, we compute the layouts and workload costs. Note that the partitionings produced by AutoPart, HillClimb, HYRISE, Trojan, and BruteForce have roughly the same estimated costs (See Figure 3), while the costs for Navathe and O_2P are always much higher, but quite close to each other. Thus, in the following we only consider HillClimb and Navathe. Figure 7 shows the estimated workload runtime improvements over Column for the layouts computed by HillClimb and Navathe, calculated in the following way:

$$\text{Improvement over Column} = \frac{\text{Est. costs of Column} - \text{Est. costs of layout}}{\text{Est. costs of Column}} * 100\%$$

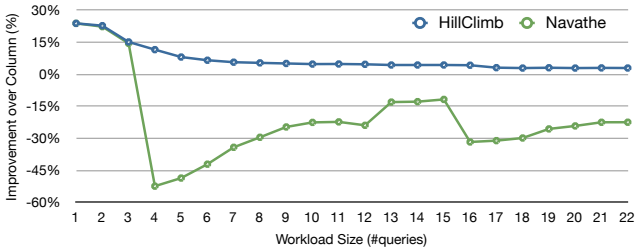


Figure 7: Estimated workload runtime improvements over Column when re-optimizing for the first k queries.

The improvement over Row remains roughly the same for both of them, so we have excluded it from this graph. However, the improvement over Column shows an interesting finding: for up to the first 3 queries, Navathe improves at least 15% over Column, but afterwards there is no improvement and it is always worse than Column. HillClimb on the other hand starts with an improvement of 24% over Column, which decreases to 6.5% for the first 6 queries, and remains roughly the same afterwards.

Let us now investigate the reason for this behavior, considering only the first 6 queries, i.e. k ranging from 1 to 6. The Table below shows the percentage of unnecessary reads for these workloads:

k	1	2	3	4	5	6
HillClimb	0%	0%	0%	0%	0%	0%
Navathe	0%	0%	0%	37%	32%	30%

Table 3: Unnecessary data reads over the Lineitem table for the first k queries.

From the table, we can see that in case of Navathe, starting from $k = 4$ the fraction of unnecessary data read has jumped from 0% to more than 30%. This explains why Navathe suddenly become worse than Column. On the other hand, the fraction of unnecessary data read for HillClimb and Column stays 0% for all these values of k. To understand the declining performance of HillClimb in Figure 7, let us take a look at tuple-reconstruction joins. Table 4 shows the average number of tuple-reconstruction joins over the Lineitem table for up to the first 6 queries. From the table,

k	1	2	3	4	5	6
HillClimb	0.00	0.00	1.00	1.00	1.75	2.00
Column	6.00	6.00	4.50	3.67	3.50	3.40

Table 4: Average number of tuple-reconstruction joins per row of the Lineitem table for the first k queries.

we see that more tuple-reconstruction joins were performed with larger workload size. This is because, with increasing workload size, the size of partitions decreases and thus the number of referenced partitions increases. Thus, with increasing values of k, the difference between the query performances of HillClimb and Column decreases. As a result, we can conclude that the random I/O accounts for most of the difference in estimated costs between HillClimb and Column.

In summary, the most important conclusion in this section is that while vertically partitioned layouts improve significantly over Row on the TPC-H benchmark, the improvement over Column is still less than 5%.

6.3 Comparing Algorithm Fragility

Below we understand the fragility of each of the algorithms with the following main questions.

What is the effect of disk characteristics on query performance?

We proceed this experiment as follows. First, we run the algorithms for the same disk characteristics: 8 KB block size, 8 MB buffer size, 90 MB/s disk read-bandwidth and 4.84 ms seek time. Then, we take the layouts obtained from these disk characteristics and see how query performance would be affected, if these disk characteristics would change at query time. The idea is to see how much does the query performance deviate from the original setting's performance, if the layouts computed under one setting were used in another setting — also defined as fragility in Section 5. Figure 8 shows the fragility of layouts produced by each of the algorithms, when changing the buffer size.

$$\text{Fragility} = \frac{\text{Est. costs with new settings} - \text{Est. costs with old settings}}{\text{Est. costs with old settings}}$$

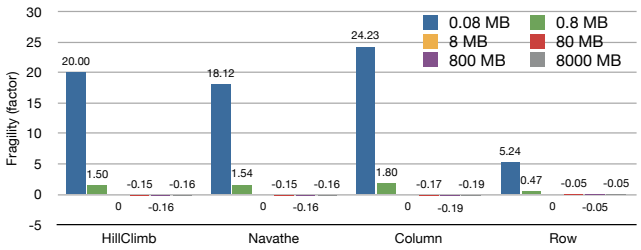


Figure 8: Algorithm fragility — estimated change in workload runtime due to changing the buffer size at query time.

From the figure, we can see that changing the buffer size can significantly affect the workload runtime, by up to 24 times. This is because buffer size strongly determines the number of random I/Os during query processing. Other disk parameters like block size, disk bandwidth, and disk seek time do not have such an impact on query performance. Interested readers can see Appendix A.2 for details. The take away message is that the performance of vertically partitioned layouts depends highly on the buffer size.

We also ran an experiment to see how the query workload costs change with changes in the query workload, i.e. to see how fragile the algorithms are to the workload changes. Our results show that query workload costs change by only 14% for up to 50% change in query workload.

6.4 Where does vertical partitioning make sense?

In this section we concern ourselves with the following issues:

What happens if we adapt to different disk characteristics?

In the previous section, we saw that the performance of vertically partitioned layouts depend strongly on the buffer size. So let us now see how much do the query times change, if the partitioning is adapted to the different buffer sizes. Figure 9 shows the estimated workload runtimes for two vertical partitioning algorithms (HillClimb and Navathe) normalized by the estimated workload runtime for Column, when the buffer size is changed.

$$\text{Normalized Estimated Costs} = \frac{\text{Estimated costs of the layout}}{\text{Estimated costs of Column}} * 100\%$$

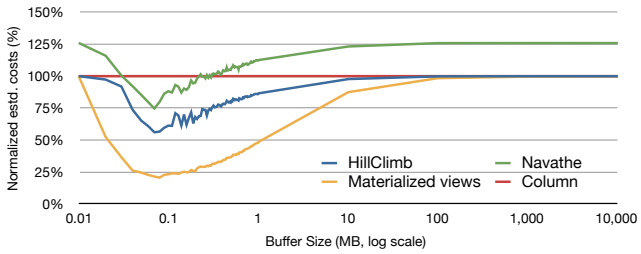


Figure 9: Estimated workload runtime compared to Column when re-optimizing for each buffer size.

Additionally, we also show the workload costs of the perfect materialized views as well as for Column. We do not show Row because it is out-performed by all other layouts for all buffer size values. In order to amplify the variation we compare the workload costs to Column for different buffer sizes. The first thing that we see is that in the best case, i.e. for the perfect materialized views, vertical partitioning pays off over Column only up to a buffer size of 100 MB. The layouts produced by HillClimb perform either better or the same as Column. HillClimb has the best improvement over Column for a buffer size of 100KB. The layouts produced by Navathe, on the other hand, perform better than Column only in a narrow range of approximately 30 KB to 300 KB. For all remaining buffer size values, Navathe performs worse than Column. For the sake of completeness, we also ran experiments to see the adaptivity of vertical partitioning algorithms over block size, disk bandwidth, and disk seek time. We have additionally examined the effects of scaling the dataset (see Appendix A.3 and A.4).

The key message from this experiment, and also from this paper, is that vertical partitioning makes sense only for small buffer sizes, e.g. less than 100 MB. This is indeed the case for many data management systems. For example, PostgreSQL has a default buffer size of 8 MB. In case we can afford to have big buffers (due to large main-memory or dedicated nodes) it is better to use column layout. We also repeated this experiment with different dataset sizes. Interested readers can see Figures 13(a) and 13(b).

7. LESSONS LEARNED

In this paper, we compared different vertical partitioning algorithms and studied ways to pick one vertical partitioning algorithm over another for row-oriented database systems. Traditionally, vertical partitioning and index selection have been treated as different problems[3] and hence we do not consider selection predicates and indexes in our study. However, we did consider putting the selection attributes in a different partition. But it turns out that this affects the data layouts only when the selectivity is higher than 10^{-4} for uniformly distributed datasets, such as TPC-H. Below we discuss the key lessons learned in this paper.

1. We don't really need brute force. The brute force algorithm spends an extremely long time to compute the layouts (more than an hour for TPC-H). On the other hand, the vertical partitioning algorithms evaluated in this paper terminate in at most a few minutes. In fact, AutoPart and HillClimb take less than 1 second to compute the layouts for all tables in the TPC-H benchmark. Still both AutoPart and HillClimb find *exactly* the same solution as the brute force algorithm. HYRISE takes slightly more than a second to compute the layouts but it is only 2.21% off from the brute force algorithm, in terms of query costs. Similarly Trojan takes a couple of minutes for optimization, however it is just 0.01% off from the brute force algorithm in terms of estimated runtime. This is an important result and shows that we do not really need the brute force algorithm. Several heuristics, as proposed in different algorithms, are good enough.

2. Watch out for the buffer size. The performance of vertically partitioned layouts depend heavily on the database buffer size. In fact, the buffer size can impact the query workload runtimes by as much as factor 20. Thus buffer size is a crucial consideration when computing vertical partitioning. Furthermore, our measurements reveal that vertical partitioning improves over column layout only for buffer sizes less than 100 MB. This means if we can have a system with buffered reads of more than 100 MB at a time, then we better use the column layout. Put another way: if we want to avoid vertical partitioning then we must increase the buffer size of our database system. This is one of the core results of this paper.

3. HillClimb is the best algorithm for disk-based systems. Amongst the six vertical partitioning algorithms compared in this paper, HillClimb turns out to be the best for the TPC-H queries. HillClimb offers the best trade-off between optimization time and workload runtime performance. It spends 4 orders of magnitude less time in optimization and still finds the same vertical partitioning as the brute force algorithm. As a result, the optimization time

[3]In fact, most of the vertical partitioning algorithms do not consider selectivities.

of HillClimb pays off the earliest (just after 25% of TPC-H workload) over row layout. Furthermore, from our experience HillClimb is also one of the easiest algorithms to understand and implement.

4. Column layouts are often good enough. On the TPC-H benchmark (i.e. all 22 queries) the vertical partitioning algorithms could improve over column layout by only up to 3.7%. This is because the attribute access patterns over all 22 queries are quite fragmented and it is hard to find column groups which satisfy most of the queries. Indeed, the improvements over column layout go up to 24% when using a small subset of the TPC-H workload (see Figure 7). But still the improvements over column layout are not dramatic. To investigate this further, we tried three changes in our experimental setup — using a different benchmark, using a different cost model, and using a commercial database system which supports column grouping.

(a) *Using a different benchmark.* We used the Star Schema Benchmark[19]. The Star Schema Benchmark has less fragmented access pattern and so we expect wider column groups. Table 5 compares the results on the TPC-H and the Star Schema Benchmark (SSB).

Layout	TPC-H	SSB
AutoPart	3.71%	5.29%
HillClimb	3.71%	5.29%
HYRISE	1.58%	5.27%
Navathe	-21.47%	1.64%
O_2P	-27.74%	1.64%
Trojan	3.71%	0.05%
BruteForce	3.71%	5.29%

Table 5: Estimated improvement over column layout with different benchmarks.

We can see that even though column grouping improves over column layout by up to 5.29% on the Star Schema Benchmark, still the improvement is not dramatic. Thus, using column layouts in the first place for TPC-H–like workloads is not a bad idea. This will avoid the complicated vertical partitioning machinery.

(b) *Using a different cost model.* We used the main-memory cost model from HYRISE [6]. It models the number of cache misses when accessing data from a column grouped layout. For TPC-H queries, we show the estimated workload runtime improvements over column layout. Table 6 compares the results when using disk (HDD) and main-memory (MM) cost models.

Layout	HDD Cost Model	MM Cost Model
AutoPart	3.71%	0.00%
HillClimb	3.71%	0.00%
HYRISE	1.58%	0.00%
Navathe	-21.47%	-15.07%
O_2P	-27.74%	-15.53%
Trojan	3.71%	0.00%
BruteForce	3.71%	0.00%

Table 6: Estimated improvement over column layout with different cost models.

From the table we see that except for Navathe and O_2P, all other algorithms have no improvement over column layout in main-memory. This is due to the fact that the seek-costs compared to the scan costs are way smaller in main-memory than for disk-based systems, which means that a column-group cannot significantly decrease the data access costs in main-memory. Instead, column groups can potentially increase the amount of data read and hence be even worse than column layout (see Navathe and O_2P for main-memory). On the other hand, reading data in column layout causes the least possible number of cache-misses, thus allows for the fastest data access. Therefore, in terms of data access costs, it is hard to beat column layout in a main memory-based system. Indeed, in the HYRISE-paper [6], the hybrid layouts improve over column layout by just 3.8% in the total workload cost. This is even when the workload chosen in HYRISE paper uses very wide tables with up to 150 attributes and several queries accessing a large fraction of those attributes.

(c) *Using a commercial database system.* Finally, we used a commercial disk-based column-oriented database system (referred to as DBMS-X in the following), which supports column grouping. The idea is to compare vertically partitioned layouts with column layouts on TPC-H benchmark. To do so, we created and loaded two TPC-H databases with scale factor 10, one with column layout and the other with a vertically partitioned layout calculated by HillClimb. Like any other column store, DBMS-X relies heavily on compression and it cannot be turned off. The default compression for string and floating point numbers is Lempel-Ziv-Oberhumer-based (LZO), while for integer and date types the compression scheme is delta encoding. We executed the unmodified queries of the TPC-H workload on these two databases. Table 7 shows the total workload runtime[4] for row, column, and the vertically partitioned layout produced by HillClimb.

Compression	Row	Column	HillClimb
Default (LZO or Delta)	1652 s	377 s	450 s
Dictionary	1265 s	511 s	532 s

Table 7: TPC-H workload runtimes with scale factor 10 in DBMS-X for different layouts and compression schemes

When using the default compression the difference between column layout and HillClimb is quite high. This is due to the varying length encoding, used in the vertically partitioned layout as well, which makes the tuple-reconstructions within a segment of a column-group costly. We ran another experiment in which we forced all layouts to use the dictionary compression, which is a fixed-size encoding. With dictionary compression, the gap between column and HillClimb layout reduces. Still, column layout outperforms HillClimb.

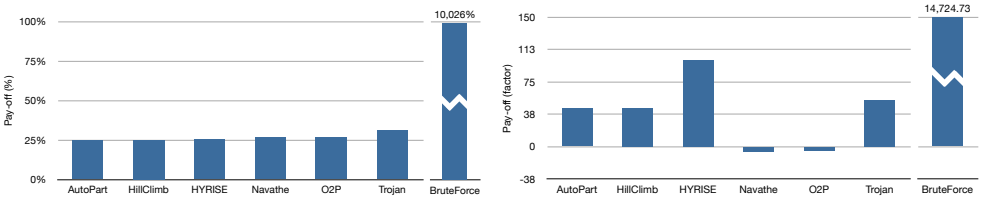
Having said the above, however, there are several practical limitations to using column layouts in legacy row stores. For instance, the standard practice to create a separate table for each vertical partition causes the column layouts to incur the maximum tuple header overheads. Thus, vertical partitioning is still a necessity for majority of row stores.

8. CONCLUSION

There are a number of vertical partitioning algorithms proposed in the literature. In this paper, we presented a systematic and comprehensive study of vertical partitioning algorithms. We categorized vertical partitioning algorithms along three dimensions and surveyed six different algorithms. We experimentally evaluated these six algorithms under a common configuration setting. We introduced four metrics to compare different vertical partitioning algorithms and showed results from the TPC-H benchmark. Our results identified the trade-offs between optimization time and workload runtime improvements, improvements over row and column layouts, and effects of database buffer size.

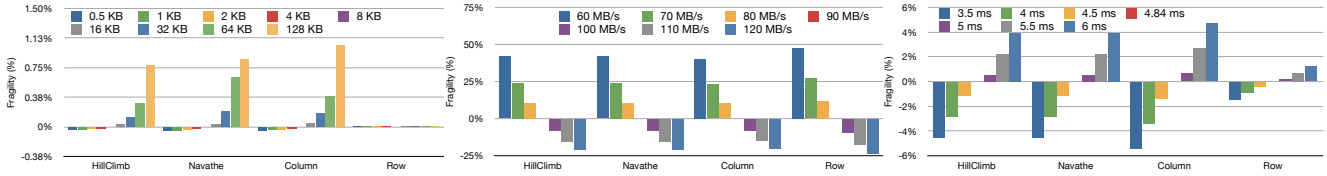
Acknowledgments. Research partially supported by BMBF.

[4]We excluded query 9 since DBMS-X has chosen a sub-optimal query plan for it, which caused an enormously high runtime.



(a) Pay-off over Row (b) Pay-off over Column

Figure 10: Pay-off in workload runtime improvements over optimization- and creation times.



(a) Changing the block size (b) Changing the disk's bandwidth (c) Changing the seek time

Figure 11: Algorithm fragility — estimated change in workload runtime due to changing a single parameter at query time.

9. REFERENCES

[1] S. Agrawal, V. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *ACM SIGMOD*, pages 359–370, 2004.

[2] Bonnie++, `coker.com.au/bonnie++`.

[3] W. W. Chu and I. T. Ieong. A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems. *IEEE Trans. Softw. Eng.*, 19(8):804–812, 1993.

[4] D. W. Cornell and P. S. Yu. A Vertical Partitioning Algorithm for Relational Databases. In *ICDE*, pages 30–35, 1987.

[5] D. W. Cornell and P. S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Trans. Softw. Eng.*, 16(2):248–258, 1990.

[6] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[7] M. Hammer and B. Niamir. A Heuristic Approach to Attribute Partitioning. In *ACM SIGMOD*, pages 93–101, 1979.

[8] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, pages 417–428, 2003.

[9] HBase, `hbase.apache.org`.

[10] J. A. Hoffer and D. G. Severance. The Use of Cluster Analysis in Physical Data Base Design. In *VLDB*, pages 69–86, 1975.

[11] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. In *BIRTE*, pages 65–80, 2011.

[12] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *ACM SOCC*, pages 21:1–21:14, 2011.

[13] A. Jindal, F. M. Schuhknecht, J. Dittrich, K. Khachatryan, and A. Bunte. How Achaeans Would Construct Columns in Troy. In *CIDR*, 2013.

[14] W. T. M. Jr., P. J. Schweitzer, and T. W. White. Problem Decomposition and Data Reorganization by a Clustering Technique. *Operations Research*, 20(5):993–1009, 1972.

[15] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. *ACM TODS*, 9(4):680–710, 1984.

[16] S. B. Navathe and M. Ra. Vertical Partitioning for Database Design: A Graphical Algorithm. In *ACM SIGMOD*, pages 440–450, 1989.

[17] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM*, pages 383–392, 2004.

[18] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, pages 60–69, 2008.

[19] Star Schema Benchmark, `www.cs.umb.edu/~poneil/StarSchemaB.PDF`.

[20] Vertica, `vertica.com`.

APPENDIX

A. ADDITIONAL RESULTS

A.1 How soon does vertical partitioning pay-off?

Now let us see how soon do the efforts made in vertical partitioning pay off, i.e. the fraction (or factor) of workload for which the accumulated workload cost improvements exceed the optimization and layout creation costs. Thus, we define pay-off as follows:

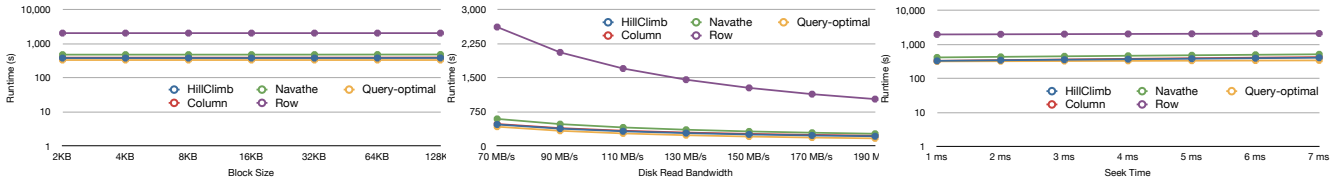
$$\text{Pay-off} = \frac{\text{Optimization time} + \text{Creation time}}{\text{Improvement in estimated workload costs}} * 100\%$$

Figure 10(a) shows when the algorithms pay off over Row. We can see that all algorithms pay off after approximately 25% of the workload has been executed. Due to the very high query costs for Row we do not see a variation of the pay-off for the different layouts. Pay-off after 25% of the workload means that just 25% of the TPC-H workload is enough motivation for computing the vertically partitioned layouts.

Figure 10(b) shows how soon vertical partitioning pays-off over Column . We can see that AutoPart pays off the earliest, after running the TPC-H workload 44.5 times. HYRISE is the last to pay off (after running the TPC-H workload 101 times). This long time to pay off over Column is due to the very small improvement (up to only 5%) in workload costs over Column. As a final remark, we see in Figure 10(b) that Navathe and O_2P have negative pay-off factors, This is because these two algorithms do not improve workload costs over Column.

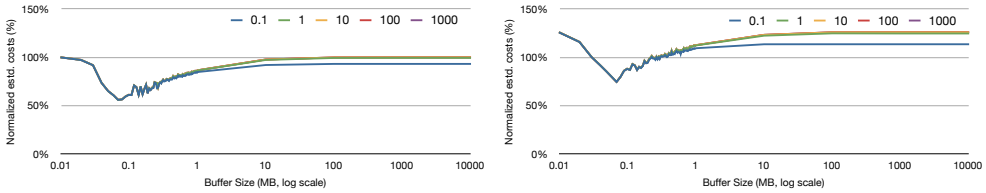
A.2 How fragile are algorithms to block size, disk bandwidth, and disk seek time?

Figures 11(a) to 11(c) show the fragility of vertical partitioning algorithms when changing block size, disk bandwidth, and disk seek time at query time, respectively. From Figure 11(a), we can see that changing disk block size has negligible impact – less than 1% – on query workload performance. This is because a database system needs to read integral number of blocks and changing the block size affects only the last block. Changing the disk bandwidth deviates the workload runtime by up to 42% (Figure 11(b)), while changing the seek time deviates the workload runtime by less than 5% (Figure 11(c)). Thus, we see that the performance of vertically partitioned layouts are stable over block size and disk seek time,



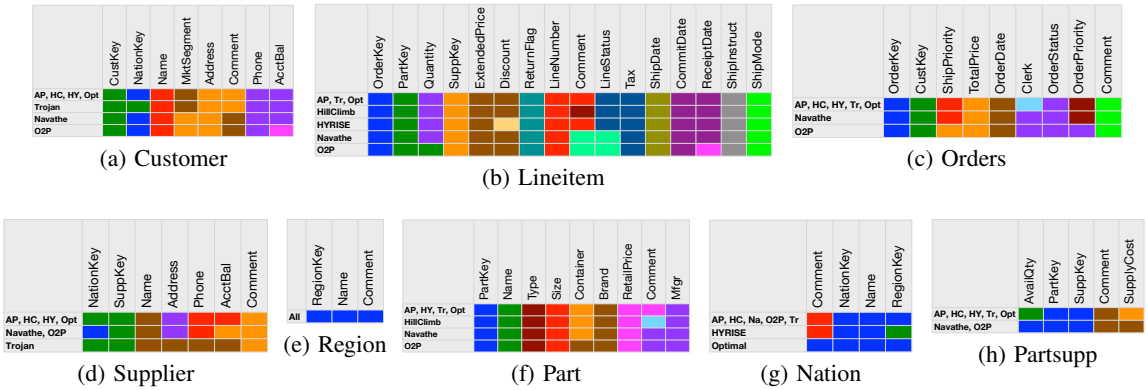
(a) Changing block size (b) Changing disk bandwidth (c) Changing disk seek time

Figure 12: Estimated workload runtime when re-optimizing for each block size, disk bandwidth and seek time



(a) Scaling dataset with HillClimb (b) Scaling dataset with Navathe

Figure 13: Sweet-spots for vertical partitioning — re-optimizing for each buffer size and each dataset size, and showing the estimated workload runtime compared to Column.



(a) Customer (b) Lineitem (c) Orders

(d) Supplier (e) Region (f) Part (g) Nation (h) Partsupp

Figure 14: The computed partitions for the TPC-H workload.

they are affected marginally by disk bandwidth, but they highly depend on buffer size, as seen in Section 6.3.

A.3 What are the sweet spots for block size, disk bandwidth, and disk seek time?

Figures 12(a) to 12(c) add to the findings of Subsection 6.4, and shows the estimated workload costs for the vertical partitioned layouts for different block sizes, disk bandwidths and disk seek times. We can see that the algorithms are almost unaffected by changes in block size (Figure 12(a)) and disk seek time (Figure 12(c)) – the standard deviations of the estimated costs compared to the averages are less than 0.5% and 9% respectively. To a certain degree, the algorithms are affected by changes in disk bandwidth (Figure 12(b)) – the afore mentioned metric is 30% in this case. But there are no interesting regions.

A.4 Do the sweet spots change with scaling dataset size?

Let us now examine the effects of changing the buffer size together with scaling the dataset (i.e. varying the scale factor of TPC-H). We recompute the layouts for every buffer-size and for every scale-factor, and compare the workload costs to Column. Figures 13(a) and 13(b) show the results for HillClimb and Navathe. We can see that there is a jump in improvements over Column from scale factor 0.1 to 1.0 and buffer size larger than 1 MB. This is because for scale factor 0.1 (i.e. 100 MB data size), each query reads the same amount of data as the buffer size. For all other regions in Figures 13(a) and 13(b), the impact of dataset size is negligible.

B. LAYOUTS

Figure 14 shows the vertical partitioned layouts for all tables in TPC-H workload. Two or more attributes having the same color in a given row means that they belong to the same vertical partition. For the Lineitem table (Figure 14(b)) AutoPart, Trojan, and Optimal produce the same results. HillClimb's results differ only in not grouping the two unreferenced attributes (LineNumber and Comment) together. The same occurred for the Part table (Figure 14(f)) where HillClimb left the two unreferenced attributes (RetailPrice and Comment) in separate partitions, contrary to AutoPart, HYRISE, Trojan and Optimal. The reason for Trojan producing a slightly different layout for the Customer and Supplier tables – compared to the other algorithms in the "HillClimb-class" – is that it uses an interestingness-measure as a heuristic making it sometimes chose sub-optimal column-groups as well. Navathe and O_2P form the second class of the vertical partitioning algorithms we have considered which is clearly visible on the partitioning results, since they always produce a partitioning which has significant differences from the results of the "HillClimb-class". For the Nation and Region tables (Figures 14(g) and 14(e)), the partitioning doesn't influence the I/O cost. This is because these two tables have only 25 and 5 rows, respectively, and hence they fit into one block.