H_2O: A Hands-free Adaptive Store

Ioannis Alagiannis[⋆] Stratos Idreos[‡] Anastasia Ailamaki[⋆]

[⋆]Ecole Polytechnique Fédérale de Lausanne
{ioannis.alagiannis, anastasia.ailamaki}@epfl.ch

[‡]Harvard University
stratos@seas.harvard.edu

ABSTRACT

Modern state-of-the-art database systems are designed around a single data storage layout. This is a fixed decision that drives the whole architectural design of a database system, i.e., row-stores, column-stores. However, none of those choices is a universally good solution; different workloads require different storage layouts and data access methods in order to achieve good performance.

In this paper, we present the H_2O system which introduces two novel concepts. First, it is flexible to support multiple storage layouts and data access patterns in a single engine. Second, and most importantly, it decides on-the-fly, i.e., during query processing, which design is best for classes of queries and the respective data parts. At any given point in time, parts of the data might be materialized in various patterns purely depending on the query workload; as the workload changes and with every single query, the storage and access patterns continuously adapt. In this way, H_2O makes no a priori and fixed decisions on how data should be stored, allowing each single query to enjoy a storage and access pattern which is tailored to its specific properties.

We present a detailed analysis of H_2O using both synthetic benchmarks and realistic scientific workloads. We demonstrate that while existing systems cannot achieve maximum performance across all workloads, H_2O can always match the best case performance without requiring any tuning or workload knowledge.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design - Access methods; H.2.4 [**Database Management**]: Systems - Query Processing

General Terms

Algorithms, Design, Performance

Keywords

Adaptive storage; adaptive hybrids; dynamic operators

1. INTRODUCTION

Big Data. Nowadays, modern business and scientific applications accumulate data at an increasingly rapid pace. This data explosion gives birth to new usage scenarios and data analysis opportunities but it also significantly stresses the capabilities of current data management engines. More complex scenarios lead to the need for more complex queries which in turn makes it increasingly more difficult to tune and set-up database systems for modern applications or to maintain systems at a well-tuned state as an application evolves.

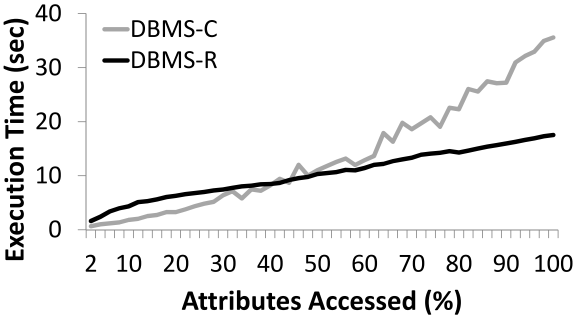


Figure 1: Inability of state-of-the-art database systems to maintain optimal behavior across different workload patterns.

The Fixed Storage Layout Problem. The way data is stored defines how data should be accessed for a given query pattern and thus it defines the maximum performance we may get from a database system. Modern state-of-the-art database systems are designed around a single data storage layout. This is a fixed decision that drives the whole design of the architecture of a database system. For example, traditional row-store systems store data one row at a time [20] while modern column-store systems store data one column at a time [1]. However, none of those choices is a universally good solution; different workloads require different storage layouts and data access methods in order to achieve good performance. Database systems vendors provide different storage engines under the same software suite to efficiently support workloads with different characteristics. For example, MySQL supports multiple storage engines (e.g., MyISAM, InnoDB); however, communication between the different data formats on the storage layer is not possible. More importantly, each storage engine requires a special execution engine, i.e., an engine that knows how to best access the data stored on each particular format.

Example. Figure 1 illustrates an example of how even a well-tuned high-performance DBMS cannot efficiently cope with various workloads. In this example, we test 2 state-of-the-art commercial systems, a row-store DBMS (DBMS-R) and a column-store DBMS (DBMS-C). We report the time needed to run a single analytical select-project-aggregate query in a modern machine. Figure 1 shows that none of those 2 state-of-the-art systems is a universally good solution; for different classes of queries (in this case depending on the number of attributes accessed), a different system

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610502.

is more appropriate and by a big margin (we discuss the example of Figure 1 and its exact set-up in more detail later on).

The root cause for the observed behavior is the fixed *data layout* and the fixed *execution strategies* used internally by each DBMS. These are closely interconnected and define the properties of the query engine and, as a result, the final performance. Intuitively, column stores perform better when columns are processed independently, while row-stores are more suited for queries touching many attributes. In both systems, the data layout is a static input parameter leading to compromised designs. Thus, it restricts these systems from adapting when the workload changes. Contrary to common belief that column-stores always outperform row-stores for analytical queries, we observe that row-stores can show superior performance in a class of workloads which becomes increasingly important. Such workloads appear both in business (e.g., network performance and management applications) and scientific domains (e.g., neuro-science, chemical and biological applications) and the common characteristic is that queries access an increased number of attributes from wide tables. For example, neuro-imaging datasets used to study the structure of human brain consist of more than 7000 attributes. In this direction, commercial vendors are continuously increasing the support for wide tables e.g., SQL Server today allows a total of 30K columns per table while the maximum number of columns per SELECT statement is now at 4096, aiming at serving the requirements of new research fields and applications.

Ad-hoc Workloads. If one knows the workload a priori for a given application, then a specialized hybrid system may be used which may be perfectly tuned for the given workload [16, 29]. However, if the workload changes a new design is needed to achieve good performance. As more and more applications, businesses and scientific domains become data-centric, more systems are confronted with ad-hoc and exploratory workloads where a single design choice cannot cover optimally the whole workload or may even become a bottleneck. As a result modern businesses often need to employ several different systems in order to accommodate workloads with different properties [52].

H_2O: An Adaptive Hybrid System. An ideal system should be able to combine the benefits of all possible storage layouts and execution strategies. If the workload changes, then the storage layout must also change in real time since optimal performance requires workload-specific storage layouts and execution strategies.

In this paper, we present the H_2O system that does not make any fixed decisions regarding storage layouts and execution strategies. Instead, H_2O continuously adapts based on the workload. Every single query is a trigger to decide (or to rethink) how the respective data should be stored and how it should be accessed. New layouts are created or old layouts are refined on-the-fly as we process incoming queries. At any given point in time, there may be several different storage formats (e.g., rows, columns, groups of attributes) co-existing and several execution strategies used. In addition, the same piece of data may be stored in more than one formats if different parts of the query workload need to access it in different ways. The result is a query execution engine DBMS that combines Hybrid storage layouts, Hybrid query plans and dynamic Operators (H_2O).

Contributions. Our contributions are as follows:

- We show that fixed data layout approaches can be sub-optimal for the challenges of dynamic workloads.
- We show that adaptive data layouts along with hybrid query execution strategies can provide an always tuned system even when the workload changes.
- We discuss in detail lightweight techniques for making on-the-fly decisions regarding a good storage layout based on query patterns, for refining the actual data layouts and for compiling on-the-fly the necessary operators to provide good access patterns and plans.
- We show that for dynamic workloads H_2O can outperform solutions based on static data layouts.

2. BACKGROUND AND MOTIVATION

In this section, we provide the necessary background regarding the basics of column store and row store layouts and query plans. Then we motivate the need for a system that always adapts its storage and execution strategies. We show that different storage layouts require completely different execution strategies and lead to drastically different behavior.

2.1 Storage Layout and Query Execution

Row-stores. Traditional DBMS (e.g., Oracle, DB2, SQL Server) are mainly designed for OLTP-style applications. They follow the N-ary storage model (NSM) in which data is organized as tuples (rows) and is stored sequentially in slotted pages. The row-store data layout is optimized for write-intensive workloads and thus inserting new or updating old records is an efficient action. On the other hand, it may impose an unnecessary overhead both in terms of disk and memory bandwidth if only a small subset of the total attributes of a table is needed for a specific query. Regarding query processing, most NSM systems implement the volcano-style processing model in which data is processed one tuple (or block) at a time. The tuple at a time model comes with nearly negligible materialization overhead in memory; however, it leads to increased instruction misses and a high function call overhead [7, 40].

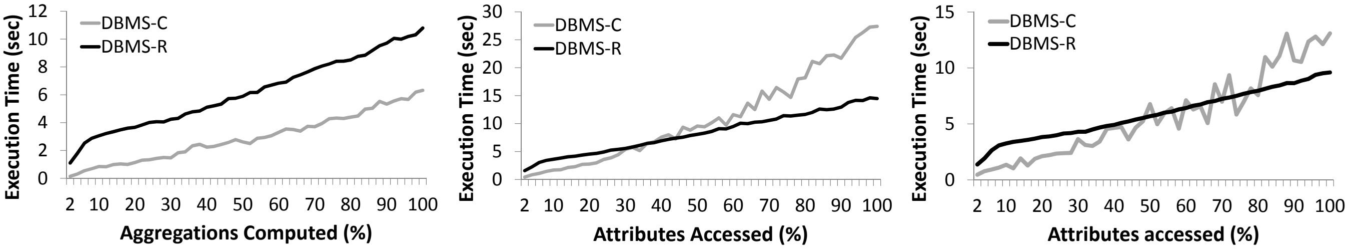
Column-stores. In contrast, modern column-store DBMS (e.g., SybaseIQ [35], Vertica [32], Vectorwise [54], MonetDB [7]) have been proven the proper match for analytical queries (OLAP applications) since they can efficiently execute queries with specific characteristics such as low projectivity and aggregates. Column-stores are inspired by the decomposition storage model (DSM) in which data is organized as columns and is processed one column at a time. The column-store data layout allows for loading in main memory only the relevant attributes for a query and thus significantly reducing I/O cost. Additionally, it can be efficiently combined with low-level architecture-conscious optimizations and late materialization techniques to further improve performance. On the other hand, reconstructing tuples from multiple columns and updates might become quite expensive.

Query Processing. Lets assume the following query.

$$Q1\text{: } select\ a+b+c\ from\ R\ where\ d{<}v_1\ and\ e{>}v_2$$

In a typical row-store query execution, the system reads the data pages of relation R and processes single tuples one-by-one according to the operators in the query plan. For $Q1$, firstly, the query engine performs predicate evaluation for the two conditional statements. Then, if both predicates qualify, it computes the expression in the select clause. The aforementioned steps will be repeated until all the tuples of the table have been processed.

For the same query, a column-store follows a different evaluation procedure. The attributes processed in the query are accessed independently. Initially, the system reads column d (assuming d is the highly selective one) and evaluates the predicate $d < X$ for all the values of column d. The output of this step is a list of tuple IDs of the qualifying tuples which is used to fetch all the qualifying tuples of e and materialize them in a new intermediate column. Then, the intermediate column is accessed and the predicate $e > Y$ is evaluated. Finally, a new intermediate list of IDs is created for the qualifying tuples considering both predicates in the where clause. The



(a) Selectivity 100% (no *where* clause) (b) Selectivity 40% (c) Selectivity 1%

Figure 2: DBMS-C vs. DBMS-R: the "optimal" DBMS changes with the workload.

latter list of tuple IDs is used to filter columns *a*, *b* and *c* processed in the select clause before applying the sum operator to compute the final aggregation result. The above query processing algorithm is based on a late tuple re-construction policy. There are more possible implementations and optimizations for the same query plan (e.g., using early materialization, bit-vectors instead of list of IDs, considering the vectorized execution paradigm). Nevertheless, the common characteristic is the materialization overhead of intermediate results which becomes significant when many attributes are accessed in the same query.

Overall, a column-store DBMS exploits different execution strategies than a row-store DBMS to fully benefit from the column-oriented data layout [2]. To identify the optimal way for executing a query not only the storage layout but the execution model should be considered. Each choice of data layout and execution strategy comes with pros and cons, and the right combination depends on the target application and workload.

2.2 One Size Does Not Fit All

We now revisit our motivating experiment from Section 1 to discuss in more detail the fact that even well-tuned systems cannot provide optimal performance when the workload changes.

Software and Methodology. We use two state-of-the-art disk-based commercial DBMS, a row-store and a columns-store. To preserve anonymity we refer to the column-store DBMS as "DBMS-C" and to the row-store DBMS as "DBMS-R". The data fits in main memory and we report execution time from hot runs to focus on the in-memory processing part of the query engine and avoid any interference with disk I/O operations and especially compression that can hide storage layout specific characteristics. Additionally, indexes are not used. Both systems compute query results over uncompressed data in memory and are tuned to use all the available CPUs on our server. Comparing full systems is not trivial as these systems are very complex and full of rich features that may affect performance. To the best of our knowledge the above comparison isolates as best as possible the performance relevant to the storage layout and execution patterns in these commercial systems.

Database Workload. The input relation consists of 50 million tuples and each tuple contains 250 attributes with integers randomly distributed in the range $[-10^9, 10^9]$. We examine two different types of queries: a) project and b) select-project. In both cases the queries compute aggregations on a set of attributes and the projectivity progressively increases from 2% to 100%. We use aggregations to minimize the number of tuples returned from the DBMS and thus we avoid any overhead that might affect the execution times. The second set of queries has an extra *where* clause consisting of multiple filter conditions. The attributes accessed in the *where* clause and in the *select* clause are the same. We generate the filter conditions so as the selectivity remains the same for all queries. The purpose of these sets of queries is to study the behavior of the two different DBMS when gradually the number of attributes involved in the query increases. For this input relation, we report a 13% larger memory footprint for DBMS-R. This is due to the overhead that comes with traditional organization of attributes into tuples and pages. Accessing more data is translated into an additional performance penalty for the above read-only workloads.

Results. Figure 2 complements the graph in Figure 1. Figure 2(a) illustrates the difference in terms of performance between DBMS-C and DBMS-R when the queries compute only aggregations. DBMS-C is always faster from 6*x* when only 5 attributes are accessed up to 65% when all attributes are accessed. In Figures 2(b) and 2(c), we observe the same behavior as in Figure 1 even though the selectivity is lower, 40% and 1% respectively. When few attributes are accessed DBMS-C is faster; however, as the number of attributes accessed both in the select and the where clause increases, we find that there is a crossover point where query processing with DBMS-C is no longer the optimal for the given queries.

Discussion. We observe that none of the two systems attains optimal performance for the whole experiment. On the contrary, which is the "best" DBMS changes as we modify the query characteristics. Row-stores expected to perform poorly when analytical queries are executed on wide tables without index support. However, we show that even with such a setup row-stores can actually be faster for certain queries demonstrating the need to have the option to move from one layout to another. Overall, selecting the underlying data layout (row-store or column-store) is a critical first tuning decision which is hard to change if the workload evolves. In this work we focus on full-table scans and we do not investigate index-accesses. Deciding which index to build, especially if there is no a priori workload knowledge is a problem orthogonal to the techniques we present.

3. THE H_2O SYSTEM

Column-stores and row-stores are extremes of the design space. If we knew the workload exactly, we could prepare the perfect hybrid design, i.e., store the frequently accessed columns together and we could also create execution strategies that perfectly exploit these layouts. However, workload knowledge is not always available while preparing all possible layouts and execution strategies up front is not possible due the vast number of choices. There is not enough space to store these alternatives and there is not enough time to prepare them. Furthermore, a system would need an equal number of specialized operators/code to properly access these layouts in order to extract all possible benefits.

In this section, we discuss the design of H_2O an adaptive hybrid query execution engine which identifies workload changes and evolves both the data organization and the execution strategy according to the workload needs. Additionally, we show how different storage data layouts can coexist in the same query engine and be combined with different execution strategies, how H_2O creates access operators on-the-fly and finally, we discuss the adaptation mechanism used by H_2O to change the data layouts.

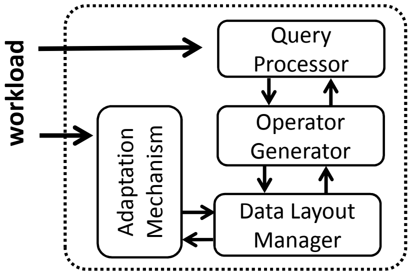


Figure 3: H_2O architecture.

Architecture. Figure 3 shows the architecture of H_2O . H_2O supports several data layouts and the Data Layout Manager is responsible for creating and maintaining the different data layouts. When a new query arrives, the Query Processor examines the query and decides how the data will be accessed. It evaluates the alternative access plans considering the available data layouts and when the data layout and the execution strategy have been chosen the Operator Generator creates on-the-fly the proper code for the access operators. The adaptation mechanism of H_2O is periodically activated to evaluate the current data layouts and propose alternative layouts to the Layout Manager.

3.1 Data Storage Layouts

H_2O supports three types of data layouts:

Row-major. The row-major layout in H_2O follows the typical way of organizing attributes into tuples and storing tuples sequentially into pages (Figure 4b). Attributes are densely-packed and no additional space is left for updates.

Column-major. In the column-store layout data is organized into individual columns (Figure 4a). Each column maintains only the attribute values and we do not store any tuple IDs.

Groups of Columns. The column-major and row-major layouts are the two extremes of the physical data layout design space but not the only options. Groups of columns are hybrid layouts with characteristics derived from those extremes. The hybrid layouts are integral part and the driving force of the adaptive design we have adopted in H_2O. A group of columns is a vertical partition containing a subset of the attributes of the original relation (Figure 4c).

In H_2O groups of columns are workload-aware vertical partitions used to store together attributes that are frequently accessed together. Attributes d and e in Figure 4c can be such an example. The width of a group of columns depends on the workload characteristics and can significantly affect the behavior of the system. Wide groups of columns in which only few attributes are accessed decrease memory bandwidth utilization, similarly with a row-major layout while a narrow group of columns might come with increased space requirements due to padding. For all the above data layouts, we consider fixed length attributes.

3.2 Continuous Layout Adaptation

H_2O targets dynamic workloads in which data access patterns change and so it needs to continuously adapt. One extreme approach is to adapt for every query. In this context, every single query can be a potential trigger to change how the respective data is stored and how it should be accessed. However, this is feasible in practice only if the cost of generating a new data layout can be amortized over a number of future queries. Covering more than one query with a new data layout can help to amortize the cost faster. H_2O gathers statistics regarding the incoming queries. The recent query history is used as a trigger to react in changes of the workload. H_2O decides a candidate layout pool by estimating the expected benefit and selecting the most fitting solution.

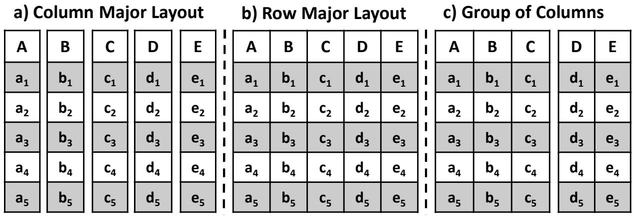


Figure 4: Data Layouts.

Monitoring. H_2O uses a dynamic window of N queries to monitor the access patterns of the incoming queries. The window size defines how aggressive or conservative H_2O is and the number of queries from the query history that H_2O considers when evaluating the current schema. For a given set of input queries H_2O focuses on statistics about attribute usage and frequency of attributes accessed together. The monitoring window is not static but it adapts when significant changes in the statistics happen. H_2O uses the statistics as an indication of the expected queries and to prune the search space of candidate data layouts. The access patterns are stored in the form of two affinity attribute matrices [38] (one for the *where* and one for the *select* clause). Affinity among attributes expresses the extent to which they are accessed together during processing. The basic premise is that attributes accessed together and have similar frequencies should be grouped together. Differentiating between attributes in the *select* and the *where* clause allows H_2O to consider appropriate data layouts according to the query access patterns. For example, H_2O can create a data layout for predicates that are often evaluated together.

Alternative Data Layouts. Determining the optimal data layout for a given workload is equivalent to the well-known problem of vertical partitioning which is NP-hard [48]. Enumerating through all the possible data layouts is infeasible in practice especially for tables with many attributes (e.g., a table with 10 attributes can be vertically partitioned into 115975 different partitions). Thus, proper heuristic techniques should be applied to prune the immense search space without putting at risk the quality of the solution.

H_2O starts with the attributes accessed by the queries to generate potential data layouts. The initial configuration contains the narrowest possible groups of columns. When a narrow group of columns is accessed by a query, all the attributes in the group are referenced. Then, the algorithm progressively improves the proposed solution by considering new groups of columns. The new groups are generated by merging narrow groups with groups generated in previous iterations. The generation and selection phases are repeated multiple times until no further improvement is possible for the input workload.

For a given workload $W = \{q_1, q_2, ..., q_n\}$ and a configuration C, H_2O evaluates the workload and transformation cost T using the following formula.

$$cost(W, C_i) = \sum_{j=1}^{n} q_j(C_i) + T(C_{i-1}, C_i) \qquad (1)$$

Intuitively, the initial solution consists of attributes accessed together within a query and by merging them together H_2O reduces the joining overhead of groups. The size of the initial solution is in the worst case quadratic to the number of narrow partitions and allows to effectively prune the search space without putting at risk the quality of the proposed solution. H_2O considers attributes accessed together in the *select* and the *where* clause as different potential groups which allows H_2O to examine more executions strategies (e.g., to exploit a group of columns in the *where* clause to generate a vector of tuple IDs for the qualifying tuples). H_2O also considers the transformation cost from one data layout to another in the evaluation method. This is critical, since the benefit of a new data layout depends on the cost of generating it and on how many times H_2O is going to use it in order to amortize the creation cost.

Data Reorganization. H_2O combines data reorganization with query processing in order to reduce the time a query has to wait for a new data layout to be available. Assuming Q1 from Section 2 and two data layouts $R1(a,b,c)$ and $R2(d,e)$. The selected data layout from the adaptation mechanism requires to merge those two data layouts into $R(a,b,c,d,e)$. In this case, blocks from $R1$ and $R2$ are read and stitched together into blocks with tuples (a,b,c,d,e). Then, for each new tuple, the predicates in the *where* clause are evaluated and if the tuple qualifies the arithmetic expression in the *select* is computed. The early materialization strategy allows H_2O to generate the data layout and compute the query result without scanning the relation twice. The same strategy is also applied when the new data layout is a subset of a group of columns.

H_2O follows a lazy approach to generate new data layouts. It does not apply the selected data layout immediately but it waits until the first query requests a new layout. Then, H_2O creates the new data layout as part of the query execution. The source code for the physical operator that generates the new data layout while computing the result of the input query is created by applying code generation techniques described in Section 3.4.

Oscillating Workloads. An adaptation algorithm should be able to detect changes to the workload and act quickly while avoiding overreacting for temporary changes. Actually, such a trade-off is part of any adaptation algorithm and it is not specific to H_2O. In the case of H_2O, adapting too fast might create additional overhead during query processing while slow adaptation might lead to suboptimal performance. H_2O minimizes the effect of false-positives due to oscillating workloads by applying the lazy data layouts generation approach described in this subsection. To completely eliminate the effect of oscillating workloads requires predicting future queries with high probability; however this is not trivial. H_2O detects workload shifts by comparing new queries with queries observed in the previous query window. It examines whether the input query access pattern is new or if it has been observed with low frequency. New access patterns are an indication that there might be a shift in the workload. In this case, the adaptation window decreases to progressively orchestrate a new adaptation phase while when the workload is stable, H_2O increases the adaptation window.

3.3 Execution Strategies

Traditional query processing architectures assume not only a fixed data layout but predetermined query execution strategies as well. For example, in a column-store query plan a predicate in the *where* clause is evaluated using vectors of tuple IDs to extract the qualifying tuples while a query plan for a row-store examines which tuples qualify one-by-one and then forwards them in the next query operator. In this paper, we show that a data layout should be combined with the proper execution strategy in a query plan. To maximize the potential of the selected query plan, tailored code should be created for the query operators in the plan (e.g., in filters it enhances predicate evaluation). Having multiple data layouts in H_2O also requires to support the proper execution strategies. Having different execution strategies means providing different implementations integrated in the H_2O query engine.

H_2O provides multiple execution strategies and adaptively selects the best combination of data layout and execution strategy according to the requirements of the input query. Execution strategies in H_2O are designed according to the vectorized query processing model [7] in which data is represented as small arrays (vectors). Vectors fit in the L1 cache for better cache locality.

Row-major. The execution strategies for a row-major layout follow the volcano execution model. Additionally, predicate evaluation is pushed-down to the scan operator for early tuple filtering.

Column-major. For a column-major layout the execution strategy of H_2O materializes intermediate results from filter evaluations into vectors of matching positions. Similarly, it handles the output of complex arithmetic expressions. For example, in Q1 of Section 2 computing the expression $a+b+c$ results into the materialization of two intermediate columns, one for $a+b$ and one for the result of the addition of the previous intermediate result with c.

Groups of columns. Regarding group of columns, there is no unique execution strategy. Data layouts can apply any of the strategies used for columns-major or row-major layouts. For example, predicate evaluation can be pushed-down or it can be computed using vectors of matching positions as in the case of a column-major layout. During query processing, H_2O evaluates the alternative execution strategies and selects the most appropriate one.

All executions strategies materialize the output results in memory using contiguous memory blocks in a row-major layout.

3.4 Creating Operators On-the-fly

The data independence abstraction in databases provides significant flexibility by hiding many low-level details (e.g., how the data is stored). However, it comes with a hit in performance due to the considerable interpretation overhead [46]. For example, computing the qualifying tuples for Q1 from Section 2 using volcano-style processing requires evaluating a conjunctive boolean expression ($d<v_1$ and $e>v_2$) for each one of the tuples. In the generic case, the engine needs to be able to compute predicates for all the supported SQL data types (e.g., integer, double) and additionally complicated sub-expressions of arbitrary operators. Thus, a generic operator interface leads to spending more time executing function calls and interpreting code of complex expressions than computing the query result [7]. Column-stores suffer less from the interpretation overhead but this comes with the cost of expensive intermediate results (e.g., need to materialize lists of IDs) and larger source code base (e.g., maintain a different operator implementation per data type).

H_2O maintains data into various data layouts and thus to obtain the best performance needs to include the implementation of numerous execution strategies.

For example, having a system that has all the possible code upfront is not possible especially when workload-aware layouts should be combined in the same query plans. The potential combinations of data layouts and access methods are numerous. To compute $a > X$ and $(a+b) > X$ requires different physical operators. A generic operator can cover both cases. However, the performance will not be optimal due to the interpretation overhead; the overhead of dynamically interpreting complex logic (e.g., expressions of predicates) to low level code. Thus, H_2O creates dynamic operators for accessing on-the-fly the data referenced from the query regardless of the way it is internally stored in the system (pure columnar or group of columns format). H_2O generates dynamic operators not only to reduce the interpretation overhead but also in order to combine workload-specific data layouts and execution strategies in the same access operator.

To generate layout-aware access operators, H_2O uses source code templates. Each template provides a high-level structure for different query plans (e.g., filter tuples with or without list of tuple IDs). Internally, a template invokes functions that generate the specialized code for specific basic operations; for example, accessing specific attributes in a tuple, evaluate boolean expressions, express

```
1   // Compiled equivalent of vectorized primitive
2   // Input: Column group R(a,b,c,d,e)
3   // For each tuple evaluate both predicates in one step
4   // Compute arithmetic expression for qualifying tuples
5   long q1_single_column_group(const int n,
6                   const T∗ res, T∗ R, T∗ val1, T∗ val2) {
7       int i, j = 0;
8       const T ∗ptr = R;
9       for ( i = 0 ; i < n; i++) {
10          if ( ptr[3] < ∗val1 && ptr[4] > ∗val2)
11              res[j++] = ptr[0] + ptr[1] + ptr[2];
12          ptr = getNextTuple(i);
13      }
14      return j;
15  }
```

Figure 5: Generated code for Q1 when all the data is stored in a single column group.

```
1   // Compiled equivalent of vectorized primitives
2   // Input: Column groups R1(a,b,c) and R2(d,e)
3   // For each batch of tuples call
4   nsel = q1_sel_vector(n, sel, R2, val1, val2);
5   q1_compute_expression(nsel, res, R1, sel);
6
7   // Compute arithmetic expression using the positions from sel
8   void q1_compute_expression(const int n,
9               const T∗ res, T∗ R1, int∗ sel) {
10      int i = 0;
11      const T ∗ptr = R1;
12      if (sel == NULL) {
13          for ( i = 0 ; i < n; i++) {
14              res[i] = ptr[0] + ptr[1] + ptr[2];
15              ptr = getNextTuple(i); }
16      } else {
17          for ( i = 0 ; i < n; i++) {
18              ptr = getNextTuple(sel, i);
19              res[sel[i]] = ptr[0] + ptr[1] + ptr[2]; }
20          }
21  }
22
23  // Compute selection vector sel for both predicates in R2(d,e)
24  int q1_sel_vector(const int n,
25                  const T∗ sel, T∗ R2, T∗ val1, T∗ val2) {
26      int i, j = 0;
27      const T ∗ptr = R2;
28      for ( i = 0 ; i < n; i++) {
29          if ( ptr[0] < ∗val1 && ptr[1] > ∗val2)
30              sel[j++] = i;
31          ptr = getNextTuple(i);
32      }
33      return j;
34  }
```

Figure 6: Generated code for Q1 when the needed attributes are stored into two different column groups.

complex arithmetic expressions, perform type casting, etc. The code generation procedure takes as input the needed data layouts from the data layout manager and the set of attributes required by the query, selects the proper template and generates as an output the source code of the access operator. The source code is compiled using an external compiler into a library and then, the new library is dynamically linked and injected in the query execution plan. To minimize the overhead of code generation, H_2O stores newly generated operators into a cache. If the same operator is requested by a future query, H_2O accesses it directly from the cache. The available query templates in H_2O support select-project-join queries and can be extended by writing new query operators.

Example. Figures 5 and 6 show two dynamically compiled equivalents of vectorized primitives for access operators for two different data layouts.

Figure 5 shows the generated code when all the accessed attributes for Q1 (a,b,c,d,e) are stored in the same column group. The on-the-fly code takes as input the group of columns, the constant values $val1$ and $val2$ used for the predicate evaluation and an output buffer for storing the result of the expression $a+b+c$ for the qualifying tuples. For each tuple H_2O evaluates in one step the two predicates for the conditional statement (Line 9) pushing down the selection to the scan operator. If both predicates are true then the arithmetic expression in the *select* clause is computed. The code is tailored for the characteristics of the available data layout and query. It fully utilizes the attributes in the data layout and thus avoids unnecessary memory accesses. Additionally, it is CPU efficient (the filter and the arithmetic expression are computed without any overhead) while it does not require any intermediate results. This code can be part of a more complex query plan.

The second on-the-fly code in Figure 6 is generated assuming two available groups of columns $R1(a,b,c)$ and $R2(d,e)$ storing the attributes in the *select* and *where* clause respectively. Since there are two groups of columns we can adopt a different execution strategy to optimize performance. In this case, the generated code exploits the two column groups by initiating a column-store like execution strategy. The query is computed using two functions; one for tuple selection and one for computing the expression. Initially, a selection vector containing the IDs of the qualifying tuples is computed. The selection vector is again computed in one step by evaluating the predicates together. The code that computes the arithmetic expression takes as parameter the selection vector, additionally to the group of columns and the values $val1$ and $val2$. Then, it evaluates the expression only for the tuple with these IDs and thus, avoiding unnecessary computation. On the other hand, the materialization of the selection vector is required.

3.5 Query Cost Model

To select the optimal combination of data layout and execution strategy, H_2O evaluates different access methods for the available data layouts and estimates the expected execution cost. The query cost estimation is computed using the following formula:

$$q(L) = \sum_{i=1}^{|L|} max(cost_i^{IO}, cost_i^{CPU}) \quad (2)$$

For a given query q and a set of data layouts L, H_2O considers the I/O and CPU cost for accessing the layouts during query processing. The cost model assumes that disk I/O and CPU operations overlap. In practice, when data is read from disk, disk accesses dominate the overall query cost since disk access latency is orders of magnitude higher than main-memory latency.

H_2O distinguishes between row-major and column-major layouts. Groups of columns are modeled similarly to the row-major layouts. The cost of sequential I/O is calculated as the amount of data accessed (e.g., number of tuples multiplied by the average tuple width for a row) divided by the bandwidth of the hard disk while the cost of random I/O additionally considers block accesses and read buffers (e..g, through a buffer pool).

H_2O estimates the CPU cost based on the number of cache misses incurred when a data layout is processed. Data cache misses have significant impact (due to cache misses cause CPU-stalls) on query processing [5] and thus, they can provide a good indication regarding the expected execution cost of query plans. A data cache miss occurs when a cache line has to be fetched from a higher level in the memory hierarchy, stalling the current instruction until needed data is available. For a given query Q and a given data layout L the cost model computes the number of data cache misses based on the data layout width, the number of tuples and the number of data words accessed for an access pattern following an approach similar to [16]. The cost of accessing intermediate results is also considered. This is important since not all execution strategies in H_2O generate intermediate results.

4. EXPERIMENTAL ANALYSIS

In this section, we present a detailed experimental analysis of H_2O. We show that H_2O can gracefully adapt to changing workloads by automatically adjusting the physical data layout and automatically producing the appropriate query processing strategies and access code. In addition, we present a sensitivity analysis on the basic parameters that affect the behavior of $\mathrm{H_2O}$ such as which physical layout is best for different types of queries. We examine how H_2O performs in comparison with approaches that use a static data layout advisor. We use both fine tuned micro-benchmarks and the real-life workload SDSS from the SkyServer project.[1]

The main benefit of hybrid data layouts comes during scan operations which are responsible for touching the majority of the data. In our analysis, we focus on scan based queries and we do not consider joins. In an efficient modern design a join implemented as in memory cache conscious join, e.g., radix join [36], would typically compute the join using only the join keys with positions (row ids) being the payload. Thus, the actual data layout of the data will have little effect during the join. On the other hand, post join projection will be affected positively (compared to using full rows) as we can fetch the payload columns from hybrid layouts.

System Implementation. We have designed and implemented a H_2O prototype from scratch using C++ with all the functionality of adaptively generating data layouts and the supporting code. Our code-generation techniques use a layer of C++ macros to generate tailored code. The compilation overhead in our experiments varies from 10 to 150 ms and depends on the query complexity. Orthogonally to this work, the compilation overhead can be further reduced by using the LLVM framework [34]. In all experiments, the compilation overhead is included in the query execution time.

Experimental Setup. All experiments are conducted in a Sandy Bridge server with a dual socket Intel(R) Xeon(R) CPU E5-2660 (8 cores per socket @ 2.20 GHz), equipped with 64 KB L1 cache and 256 KB L2 cache per core, 20 MB L3 cache shared, and 128 GB RAMrunning Red Hat Enterprise Linux 6.3 (Santiago - 64bit) with kernel version 2.6.32. The server is equipped with a RAID-0 of 7 250 GB 7500 RPM SATA disks. The compiler used is icc 13.0.0.

4.1 Adapting to Workload Changes

In this experiment, we demonstrate how $\mathrm{H_2O}$ automatically adapts to changes in the workload and how it manages to always stay close to the optimal performance (as if we had perfectly tuned the system a priori assuming enough workload knowledge).

Micro-benchmark. Here we compare $\mathrm{H_2O}$ against a column-store implementation and a row-store implementation. In both cases, we use our own engines which share the same design principles and much of the code base with $\mathrm{H_2O}$; thus these comparisons purely reflect the differences in data layouts and access patterns.

For this experiment we use a relation R of 100 million tuples. Each tuple consists of 150 attributes with integer values randomly generated in $\left[-10^9, 10^9\right)$. We execute a sequence of 100 queries. The queries are select-project-aggregation queries and each query refers to z randomly selected attributes of R, where $z = [10, 30]$.

Figure 7 plots the response time for each query in the workload, i.e., we see the query processing performance as the workload evolves. In addition to $\mathrm{H_2O}$, column-store and row-store, we plot a fourth curve in Figure 7 which represents the optimal performance; that is, the performance we would get for each single query if we had a perfectly tailored data layout as well as the most appropriate code to access the data (without including the cost of creating the data layout). We did this manually assuming for the sake of comparison against the theoretical case of having perfect workload knowledge and ample time to prepare the layout for each query.

Row-store	Column-store	$\mathrm{H_2O}$
538.2 sec	283.7 sec	204.7 sec

Table 1: Cumulative Execution Time of the Queries in Figure 7.

For this experiment, relation R is initially stored in a column-major format. This is the more desirable starting point as it is easier to morph to other layouts. However, $\mathrm{H_2O}$ can adapt regardless of the initial data layout. The initial data layout affects the query performance of the first few queries only. We discuss such an experiment in detail later on.

Initially, $\mathrm{H_2O}$ executes queries using the available data layout. In this way, we see in Figure 7 that $\mathrm{H_2O}$ matches the behavior of the column-store system. Periodically, though, $\mathrm{H_2O}$ activates the adaptation mechanism and evaluates the current status of the system (set initially at a window size of 20 queries here but this window size also adaptively adjusts as the workload stabilizes or changes more rapidly). $\mathrm{H_2O}$ identifies groups of columns that are being accessed together in queries (e.g., 5 out of the 20 queries refer to attributes $a_1, a_5, a_8, a_9, a_{10}$). Then, $\mathrm{H_2O}$ evaluates alternative data layouts and execution strategies and creates a candidate list of 4 new groups of columns. $\mathrm{H_2O}$ does not create the new layout immediately. This happens only if a query refers to the attributes in this group of columns and can benefit from the creation of the new data layout. These estimations are performed using the cost model.

Following Query 20, and having the new candidate layouts as possible future layouts, in the next 10 queries, 2 out of the 4 candidate groups of columns are created. This happens for queries 23 and 29 and these queries pay the overhead of creating the new data layout. Query 23 pays a significant part of the creation overhead; however, 4 queries use the new data layouts and enjoy optimal performance. From query 29 up to query 68 no data reorganization is required. Another data reorganization takes place after 80 and a new data layout is added to $\mathrm{H_2O}$.

There are queries for which $\mathrm{H_2O}$ cannot match the optimal performance. For example a group of columns is better for queries 2 to 8; however, there is no query that triggers the adaptation mechanism. In the end of the first evaluation it recognizes the change in the workload and proposes the needed data layouts. For 80% of the queries $\mathrm{H_2O}$ executes queries using a column group data layout while for the rest of the queries it uses a column-major layout. For the given workload a row-major data layout is suboptimal.

Overall, we observe that $\mathrm{H_2O}$ outperforms both the static column-store and row-store alternative approaches by 38% and 1.6x respectively, as shown in Table 1. More importantly it can adapt to workload changes, meaning that it can be used in scenarios where otherwise more than one systems would be necessary. Query performance in $\mathrm{H_2O}$ gets closer to the optimal case without requiring a priori workload knowledge.

$\mathrm{H_2O}$ using Real Workload. In this experiment, we evaluate $\mathrm{H_2O}$ using the SkyServer workload. We test against a scenario where we use AutoPart [41], an offline physical, design tool for vertical partitioning, in order to get the best possible physical design recommendation. For the experiment, we use a subset of the "PhotoObjAll" table which is the most commonly used and 250 of the SkyServer queries. Figure 8 shows that $\mathrm{H_2O}$ manages to outperform the choices of the offline tool. By being able to adapt to individual queries as opposed to the whole workload we can optimize performance even more than an offline tool.

Dynamic window. In this experiment, we show how $\mathrm{H_2O}$ benefits from a dynamic adaptation window when the workload changes. We use as input the same relation R as in the previous experiment

[1] http://skyserver.sdss.org

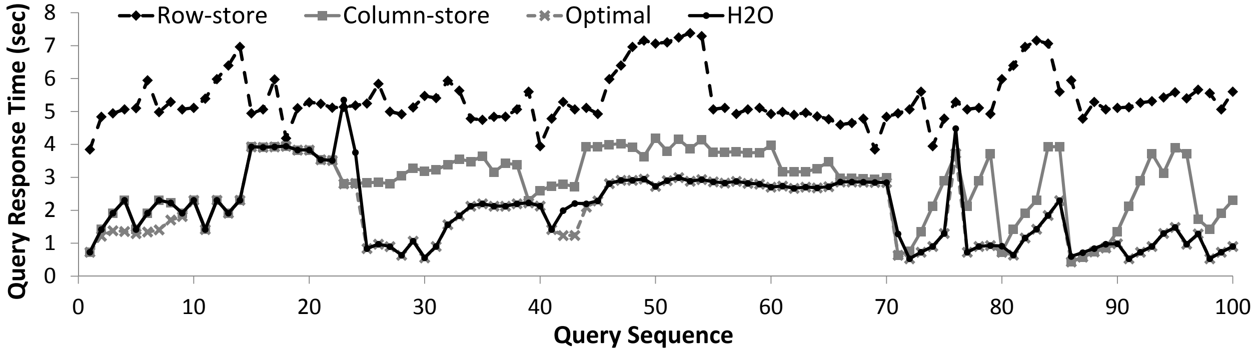


Figure 7: H_2O vs. Row-store vs. Column-store.

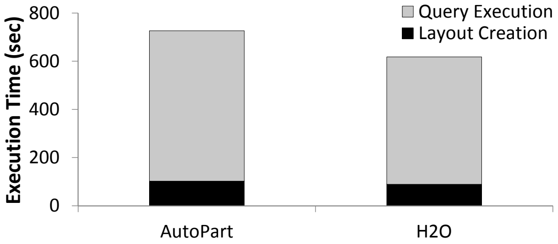


Figure 8: H_2O vs. AutoPart on the SkyServer workload.

in which data this time is organized in a row-major format and a query sequence of 60 queries. Each query refers 5 to 20 attributes. The queries compute arithmetic expressions. The first 15 queries focus on a set of 20 specific attributes while the other 45 queries to a different one. We compare two variations of H_2O, with static and with dynamic window. The size of the window is 30 queries.

Figure 9 depicts the benefit of the dynamic window. H_2O with dynamic window detects the shift in the workload after the 15th query and progressively decreases the window size to force a new adaptation phase. The adaptation algorithm is finally triggered in the 25th query generating new groups of columns layouts that can efficiently serve the rest of the workload. On the other hand, when using a static window we cannot adapt and we have to wait until the 30th query before generating new layouts and thus fails to adapt quickly to the new workload. Overall, H_2O with dynamic windows manages to adapt to changes in the workload following even if they do not happen in a periodic way.

4.2 H2O: Sensitivity Analysis

Next, we discuss a sensitivity analysis of various parameters that affect the design and behavior of H_2O . For the experiments in this section, we use a relation *R* containing 100 million tuples. Each tuple contains 150 attributes with integer values randomly generated in the range $\left[-10^9, 10^9\right)$.

4.2.1 Effect of Data Layouts

In this experiment, we present the behavior of the different data layouts integrated in H_2O using queries with different characteristics. Queries are executed using column-major, row-major and group of columns layouts. Each group of columns contains only the attributes accessed by the query. All queries are executed using the H_2O custom operators. We use as input a wide table (100 million tuples, 150 attributes) to stress test H_2O and examine how some representative query types behave as we increased the number of attributes accessed. We consider simple select project queries and queries which compute aggregations and arithmetic expressions. Queries are based on variations of the following templates:

i. "*select* a, b, ..., *from* R *where* <predicates>" for projections

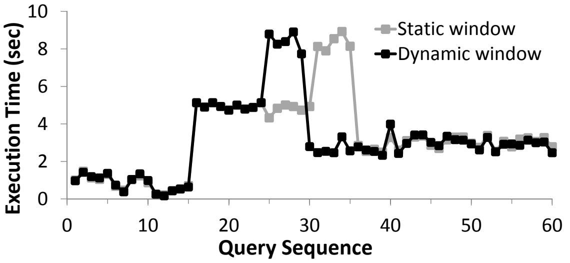


Figure 9: Static vs. dynamic adaptation window.

ii. "*select* max(a), max(b),..., *from* R *where* <predicates>" for aggregations

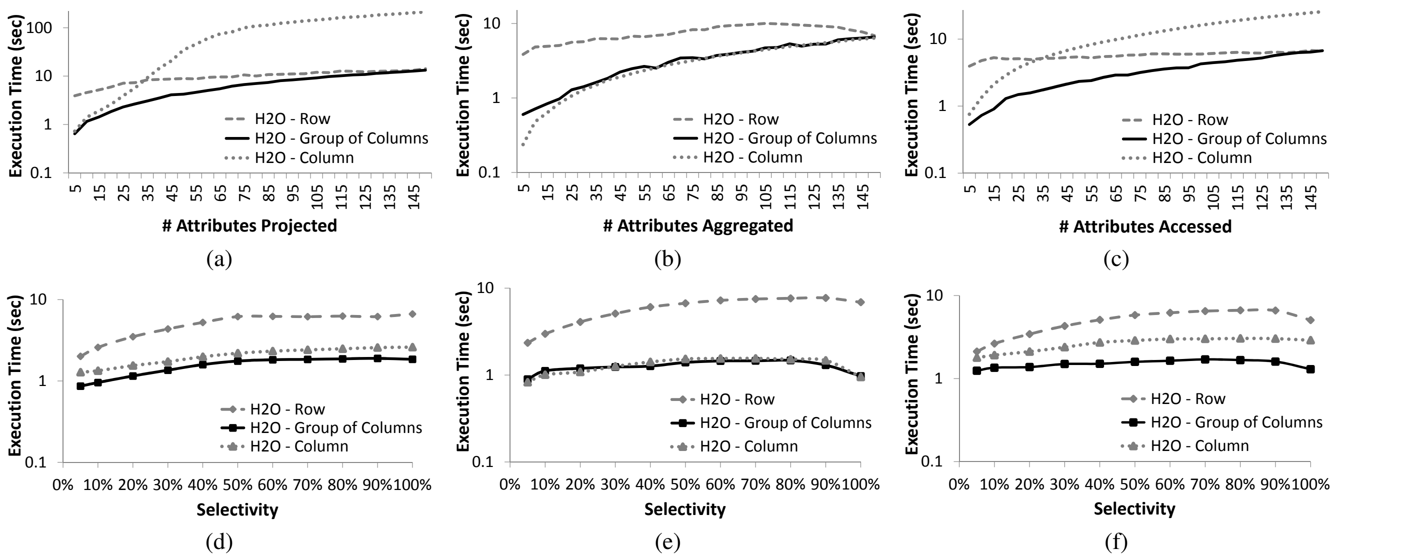
iii. "*select* a + b + ... *from* R *where* <predicates>" for arithmetic expressions

The accessed attributes are randomly generated. The cost of creating each group of columns layout is not considered in the measurements. We examine queries with and without *where* clause. In Figures 10(a-c) we vary the number of attributes accessed from 5 to 150 while there is no *where* clause. In Figures 10(d-f) each query accesses 20 attributes randomly selected from *R* while one of these attributes is the predicate in the *where* clause. We progressively vary selectivity from 0.1% to 100%. Figure 10 depicts the results (in all graphs the *y*-axis is in log scale). We report numbers from hot runs and each data point we report is the average of 5 executions. We discuss each case in detail below.

Projections. Figure 10(a) plots the query execution time for the three alternative data layouts as we increase the projected attributes while there is no *where* clause in the queries. The group of columns layout outperforms the row-major and column-major layouts regardless of the number of projected attributes. For projections of less than 30 attributes a column-major layout is faster than the row-major layout; however, when more than 20% of the attributes are referenced, performance falls up to 15X due to the high tuple reconstruction cost. As expected performance of the row-major layout and of groups of columns matches when all attributes are accessed.

For the same query template (i), in Figure 10(d) we additionally consider the effect of predicates in the *where* clause. We keep the number of projected attributes the same (20 attributes) while we vary selectivity from 0.1% to 100%. Regardless of the selectivity working with groups of columns is faster than using the other two data layouts.

Aggregations. Figure 10(b) shows the response time as we increase the number of aggregations in the queries. Using a column-major layout outperforms the other two data layouts. The biggest performance difference is when 5 aggregations are computed 1.5X and 15X from the group of columns layout and the row-major layout respectively. The gap between groups of columns and column-



(a) (b) (c)

(d) (e) (f)

Figure 10: Basic operators of H_2O.

major layout gets smaller as more aggregations are computed. The group of columns layout improves the performance in comparison to the row-major layout by reducing cache misses.

In Figure 10(e) we vary the selectivity of a query that computes 20 aggregations. The column-major layout and the group of columns layout clearly outperform the row-major layout. The column-major layout is slightly faster than the row-major layout for low selectivities but as selectivity increases the group of columns layout is marginally faster.

Arithmetic Expressions. Figure 10(c) plots the execution time for each one of the data layouts as we vary the number of attributes accessed to compute an arithmetic expression. Group of columns surpasses the column-major layout from 42% when few attributes are accessed up to 3X when more accesses required. The difference in performance is due to the cost of having to materialize intermediate results in case of the column-major layout. On the other hand, combining a group of columns with volcano-style executions allows for avoiding this overhead.

Figure 10(f) plots execution time for queries based on the third template with *where* clause. The results show that the group of columns layout has superior performance for the whole selectivity range since does not require the usage of intermediate results.

Discussion. All the above data layouts are part of H_2O. H_2O combines them with the proper execution strategies to always achieve the best performance. H_2O explores this design space to generate cache friendly operators without intermediate results if possible.

4.2.2 Effect of Groups of Columns

At any given point in time, H_2O maintains multiple layouts. Typically these are multiple column groups since plain column-major or row-major extreme cases. In this way, one reasonable question is how does performance vary depending on which column group we access. It might be that we do not always have available the optimal column group or we have to fetch the data needed for a single query from multiple column groups or we have to access a given column group containing more columns than the ones we need.

We first test the case where queries need to access only a subset of the attributes stored in a column group. Here, we use a group of 30 randomly selected attributes from *R*. The queries compute aggregations with filter following the template presented in the previous subsection, accessing 5, 10, 15, 20 and 25 randomly selected out of the 30 attributes of the column group. We examine the difference in performance for queries with 1%, 10%, 50% and 100% selectivity using the same attribute in the *where* clause. For each query, we compare the execution time with the optimal case in which a tailored data layout has been created containing only the needed attributes to answer this particular query.

Figure 11 depicts the results. It shows the performance penalty to access the whole column group as opposed to accessing the perfect column group as a ratio. The graph plots the results for each query grouped according to the selectivity. We observe that as less useful attributes are accessed the higher the performance penalty. This penalty varies with the number of useful attributes accessed. For example, when only 5 out of the 30 attributes of the group of columns are accessed, we observe the most significant drop in performance, up to 142%. This drop is due to the unnecessary memory accesses in comparison with the optimal group of columns. On the other hand, when a query accesses 25 out of the 30 attributes the overhead is almost negligible (3% in the worse case).

Other than queries having to access only a subset of a column group, another important case is when a single query needs to access multiple column groups. The question for an adaptive system in this case is whether we should build a new column group to fit the query or fetch the data from existing column groups. To study this problem, we use an aggregation with filter query *Q* that refers 25 attributes from *R*. We vary the number of groups of columns the query has to access from 2 to 5. In each case, the union of groups of columns contains all the needed attributes. For example, when 2 column groups are accessed, the first column group contains 10 of the needed attributes and the second column group the remaining 15 attributes. We experiment with selectivity 1%, 10%, 50% and 100%. We compare the response time of *Q* using the optimal column group with the cases we have to increase the number of accessed groups of columns. Figure 12 plots the response of each query normalized by the response time of queries accessing all the attributes in a single group of columns. Accessing more than one group of columns in the same query does not necessarily impose an additional overhead. On the contrary, combining two groups of columns might even be beneficial for highly selective queries.

Discussion. Accessing only a subset of the attributes of a column group, accessing multiple column groups or accessing multiple groups each one containing a subset of the columns needed

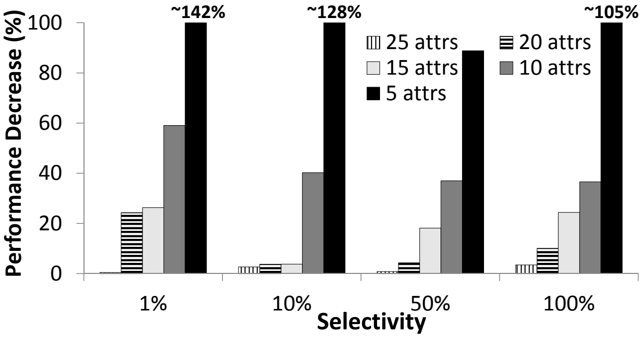


Figure 11: Accessing a subset of a column group.

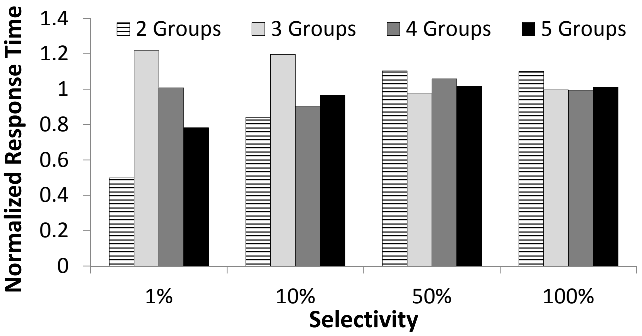


Figure 12: Accessing more than one group of columns.

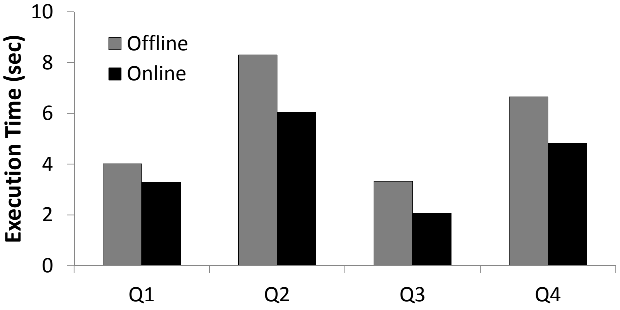


Figure 13: Online vs. Offline reorganization.

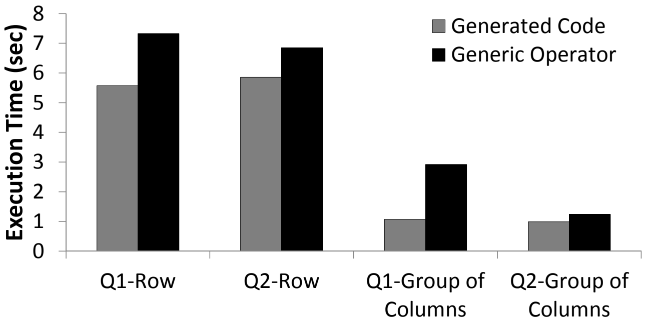


Figure 14: Generic Operator vs. Generated Code.

are few of the important scenarios when multiple column groups co-exist. The previous experiments show that groups of columns can be effective even if not the optimal group of columns is available and thus it is ok to extend monitoring periods and not to react instantly to changes in the workload in this case. Additionally, narrow groups of columns can be gracefully combined in the same query operator without imposing significant overhead.

4.2.3 Data Reorganization

H_2O adapts continuously to the workload. One approach would be to separate the preparation of data layouts from query processing. Instead H_2O answers an incoming query while adapting the physical layout for this queries. This is achieved by generating an operator that integrates the creation of a new data layout and the code needed to answer the query in one physical operator

In this experiment we show that online adaptation brings significant benefits. The set up is as follows. We assume that two new groups of columns are created from relation *R* (100 million tuples, 100 attributes with integer values). The first one contains 10 and the second 25 attributes. In the offline case, the column groups are created and then queries are executed while in the online case the column group creation and the query execution overlap. We test two scenarios. In the first scenario the initial layout is row-major (Q1 and Q2) while in the second scenario the initial layout is column-major (Q3 and Q4). In both cases, we test the cost to transform those initial layouts into the optimal set of column groups for a set of queries. We use two queries in each scenario. *Q*1 and *Q*3 trigger the generation of a new group of columns which contains 10 attributes, while *Q*2 and *Q*4 create a group of 20 columns. The queries compute 10 and 20 aggregations (without *where* clause) respectively on the attributes of the new layouts.

Figure 13 shows the results. The offline bars depict the cumulative response time for creating the column group and executing the query as two separate steps, i.e., first the data layout is created separately and only then we can process the queries. The specialized operator generated by H_2O performs the same tasks but in one step. The online case is faster regardless of the storage of the initial relation and the width of the new column group. The improvement varies from 22% to 37% when the initial relation is stored in a columnar layout and from 38% to 61% when the initial data is in a row-oriented layout. For all cases online reorganization is significantly faster than performing the same operation offline. By overlapping execution and data organization H_2O manages to improve significantly the overall execution time of the two tasks.

4.2.4 Importance of Dynamic Operators

In this experiment, we showcase the benefit of creating tailored code on-the-fly to match the available data layouts. The set-up is as follows. *Q*1 is an aggregation and *Q*2 is an arithmetic expression query following the templates presented in Section 4.2.1 and accessing 20 out of the 150 attributes of *R*. We test the execution time achieved by a generic database operator versus an operator that uses tailored code created on the fly to match the underlying layout. We examine the effect both for row-major layout and groups of columns. The code generation time is included in the overall query execution time of the dynamically generated operator and varies from 63 ms to 84 ms. Figure 14 shows the results. We observe from 16% up to 1.7x performance improvement by creating tailored code which is due to removing the interpretation overhead. This justifies the choices in H_2O in creating fully adaptive layouts and code on the fly.

5. RELATED WORK

A large body of recent work both in industry and academia propose different flavors of *hybrid* solutions to cope with the recent data deluge, complex workloads and hardware changes. The need for hybrid approaches is especially amplified when trying to provide a unified solution for workloads with different characteristics. In this section, we review this body of related work and we highlight how our work pushes the state-of-the-art even further.

A Case for Hybrid Systems. Recent research ideas have recognized the potential of exploring different physical data representations under a unified processing system [43, 11]. Fractured mirrors [43] exploit the benefits of both NSM and DSM layouts in the same system. Data is duplicated in both storage layouts and queries are executed using the appropriate mirror. This path has been recently adopted in industry, by the main-memory DBMS SAP HANA [12] which combines OLTP and OLAP workloads in the same system. Similarly, SQL Server which provides a row-store memory-optimized database engine for OLTP applications and an in-memory data warehouse using ColumnStore Index for OLAP applications [33]. The hybrid approach of mixing of columnar and row-oriented representations in the same tablespace provided by DB2 with BLU Acceleration [45] is the most related with H_2O. In addition, HyPer [29], a main-memory hybrid workload system, combines the benefits of OLTP and OLAP database by executing mixed workloads of analytical and transactional queries in parallel on the same database. Furthermore, recent vision works highlight the importance of hybrid designs. RodentStore [10] envisions a hybrid system which can be tuned through a declarative storage algebra interface while Idreos et al. [22] presents the idea of a fully adaptive system that directly reads raw files and stores data in a query driven way.

The above hybrid approaches expand the capabilities of conventional systems to support mixed workloads assuming, however, static (pre-configured) data layouts and thus cannot cope with dynamic workloads.

Our work here is focused on dynamic environments to enable quick reactions and exploration when the workload is not known up front and with limited time to invest in initialization costs. It focuses on hybrid data layouts for analytical workloads providing a fully adaptive approach both at the storage and the execution level to efficiently support dynamic workloads. In that respect, we also share common ground with other early works in exploration based techniques, e.g., [6, 10, 22, 26, 30, 37].

Layout and Workload-aware Optimizations. When considering hybrid architectures, most existing approaches are centered around storage aspects, i.e., they optimize the way tuples are stored inside a database page and the proper way of exploiting them during query processing. Harizopoulos et al. [19] explore performance trade-offs between column- and row-oriented architectures in the context of read-optimized systems showing that the DSM layout performs better when only few attributes are accessed. Zukowski et al. [55] present a comprehensive analysis of the overheads of DSM and NSM models and show that combining the two layouts in the same plan can be highly beneficial for complex queries.

Organizing attributes into groups either inside the data blocks or the data pages extends the traditional space of NSM and DSM layouts with cache-friendly layouts allowing for workload specific optimizations. To improve cache locality of traditional NSM, Ailamaki et al. [4] introduce a page layout called PAX. In PAX, data is globally organized in NSM pages, while inside the page attributes are organized into vertical partitions optimizing for reducing the number of cache misses. A generalization of PAX layout is proposed in data morphing [18], where the tuples in a page can be stored in an even more flexible form combining vertical decomposition and arbitrary groups of attributes and increasing spatial locality. Multi-resolution Block Storage Model [53] stores attributes columnwise as PAX maintaining cache efficiency and groups disk blocks into "super-blocks" with tuples stored among the blocks of a super-block improving I/O performance of scan operations. In a similar fashion, Blink [44] vertically partitions columns in byte-aligned column groups called banks, allowing for efficient ALU operations. Blink assigns particular columns into banks trying to minimize padding overhead and wasted space.

All approaches above provide valuable design alternatives. However, one still needs to know the workload before deciding which system to use and needs multiple of those systems to accommodate varying workloads. H_2O pushes the state-of-the-art further by providing a design which continuously adapts to the workload.

Auto-tuning Techniques. In the past years, there has been considerable work on automated physical design tuning. These techniques facilitate the process of automatically selecting auxiliary data structures (e.g., indices, materialize views) for a given workload to improve performance. Offline approaches [3, 8, 41] assume a priori knowledge of the workload and cannot cope with dynamic scenarios while online approaches [49] try to overcome this limitation by monitoring and periodically tuning the system. Online partitioning [28] adapts the database partitions to fit the observed workload. However, the above approaches are designed assuming a static data layout while the execution strategies remain fixed.

Adapting Storage to Queries with Adaptive Indexing. Adaptive indexing [13, 14, 15, 17, 23, 24, 25, 27, 50] tackles the problem of evolving workloads in the context of column-stores by building and refining partial indexes during query processing. The moto of adaptive indexing is that the "queries define how the data should be stored". We share this motivation here as we also adapt the storage layout on-the-fly based on queries. However, adaptive indexing research has focused on refining the physical order of data within a single column at a time without considering co-locating values across columns. The work on partial sideways cracking considers multiple columns [25] but what it does is to physically reorganize more than one columns in the same way as opposed to enforcing co-location of values from multiple columns as we do here.

In addition, the interest in systems with hybrid storage layouts has given rise to layout-aware workload analysis tools. Data morphing [18] uses a cache miss cost model to select the proper attribute grouping within an individual page that maximizes performance for a given workload. The hybrid engine HYRISE [16] applies the same idea and presents an offline physical deign tool that uses a cache misses cost model to evaluate the expected performance of different data partitions and proposes the proper vertical partitioning for a given workload. A storage advisor for SAP HANA database is presented by Rösch et al. [47] considering both, queries and data characteristics, to propose horizontal and vertical partitioning schemas. H_2O extends AutoPart [41], an offline vertical partitioning algorithm to work for dynamic scenarios. The above approaches use a static storage layout that is determined when the relation is created, are optimized assuming a known workload and cannot adapt to dynamic changes in the workload. In this paper, we highlight that no static layout can be optimal for every query and we design a system that can autonomously refine its storage layouts and execution strategies as the workload evolves.

Just-In-Time Compilation. Compilation of SQL queries into native code goes back to System R [9]. Recently, there have been many efforts to take advantage of dynamic translation techniques, such as Just-In-Time (JiT) compilation in the context of DBMS to alleviate the interpretation overhead of generic expressions, improve data locality, generate architecture specific source code and thus, significantly enhance performance. JiT techniques have been applied to main-memory DBMS using the LLVM framework [39, 34], column-stores [51], group of column systems [42] and stream processing systems [21] to generate code for the whole query execution plan [31], to revise specific code segments or to generate building primitives for higher-order composite operators. H_2O applies similar techniques to generate layout-aware access operators.

6. CONCLUSIONS

Traditional systems use a static and fixed design regarding data layouts. However, as applications become more and more data-driven and with ad-hoc workloads it becomes increasingly hard for a single traditional system, i.e., with a fixed layout, to be able to efficiently cover a multitude of scenarios. In this way, today it is not uncommon for businesses to employ more than one systems.

In this paper, we showcase the problem that for analytical queries multiple layouts can be beneficial depending on the query workload. To get the optimal performance, we not only need the optimal layout but also query processing strategies and access operator code that are tailored for a given layout. All these together reduce cache misses, instruction misses and interpretation overhead during query execution. We propose H_2O, a system that adaptively and continuously adjusts all three of these elements. It generates on-the-fly the appropriate layouts, execution strategies and code to match the workload, as the workload evolves. Using both synthetic benchmarks and real life experiments, we demonstrate that H_2O gracefully adapts as workloads change and stays close to the optimal performance without requiring any workload knowledge.

Adaptive systems in which new data layouts are created or old layouts are refined on-the-fly as incoming queries are processed can create new exciting research paths. For example, one challenging area with potential high impact is to study (adaptive) indexing together with adaptive data layouts and execution strategies.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions on how to improve the paper. This work has been supported by the EU FP7, project No. 317858 “BigFoot - Big Data Analytics of Digital Footprints” and Swiss National Science Foundation, project No. CRSII2 136318/1, “Trustworthy Cloud Storage”.

7. REFERENCES

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] D. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, 2008.

[3] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, 2004.

[4] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[5] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, 1999.

[6] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: efficient query execution on raw data files. In *SIGMOD*, 2012.

[7] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[8] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, 2005.

[9] D. Chamberlin et al. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[10] P. Cudré-Mauroux, E. Wu, and S. Madden. The case for RodentStore: An adaptive, declarative storage system. In *CIDR*, 2009.

[11] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, 2011.

[12] F. Färber et al. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[13] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 5(7):656–667, 2012.

[14] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *VLDB J.*, 23(2):303–328, 2014.

[15] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[16] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. HYRISE - a main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

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

[18] R. Hankins and J. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*, 2003.

[19] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, 2006.

[20] J. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[21] M. Hirzel et al. IBM streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7, 2013.

[22] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. Here are my queries. Where are my results? In *CIDR*, 2011.

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

[24] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[25] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[26] S. Idreos and E. Liarou. dbTouch: Analytics at your fingertips. In *CIDR*, 2013.

[27] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *PVLDB*, 4(9), 2011.

[28] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. In *BIRTE*, 2011.

[29] A. Kemper and T. Neumann. Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[30] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB*, 4(12):1474–1477, 2011.

[31] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[32] A. Lamb et al. The Vertica analytic database: C-Store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[33] P.-Å. Larson et al. Enhancements to SQL Server column stores. In *SIGMOD*, 2013.

[34] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[35] R. MacNicol and B. French. Sybase IQ Multiplex - designed for analytics. In *VLDB*, 2004.

[36] S. Manegold, P. Boncz, and M. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3):231–246, 2000.

[37] A. Nandi and H. V.Jagadish. Guided interaction: Rethinking the query-result paradigm. In *VLDB*, 2011.

[38] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, 1984.

[39] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[40] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.

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

[42] H. Pirk et al. CPU and cache efficient management of memory-resident databases. In *ICDE*, 2013.

[43] R. Ramamurthy, D. DeWitt, and Q. Su. A case for fractured mirrors. *VLDB J.*, 12(2):89–101, 2003.

[44] V. Raman et al. Constant-time query processing. In *ICDE*, 2008.

[45] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[46] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled query execution engine using JVM. In *ICDE*, 2006.

[47] P. Rösch, L. Dannecker, G. Hackenbroich, and F. Faerber. A storage advisor for hybrid-store databases. *PVLDB*, 5(12):1748–1758, 2012.

[48] D. Saccà and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Trans. Database Syst.*, 10(1):29–56, 1985.

[49] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: continuous on-line tuning. In *SIGMOD*, 2006.

[50] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2), 2013.

[51] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *DaMoN*, 2011.

[52] M. Stonebraker and U. Çetintemel. “One size fits all”: An idea whose time has come and gone. In *ICDE*, 2005.

[53] J. Zhou and K. Ross. A multi-resolution block storage model for database design. In *IDEAS*, 2003.

[54] M. Zukowski and P. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.

[55] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN*, pages 47–54, 2008.