An Improved Dynamic Vertical Partitioning Technique for Semi-structured Data

Sahel Sharify∗, Alan Lu∗, Jin Chen∗, Arnamoy Bhattacharyya∗, Ali Hashemi∗, Nick Koudas∗, Cristiana Amza∗

∗University of Toronto
{sahel.sharifi, chenjinchenjin}@gmail.com, {luwen, arnamoyb, hashemi, amza}@ece.utoronto.ca, koudas@cs.toronto.edu

Abstract—Semi-structured data such as JSON has become the de facto standard for supporting data exchange on the Web. At the same time, relational support for JSON data poses new challenges due to the large number of attributes, sparse attributes and dynamic changes in both workload and data set, which are all typical in such data. In this paper, we address these challenges through a lightweight, in-memory relational database engine prototype and a flexible vertical partitioning algorithm that uses simple heuristics to adapt the data layout for the workload, on the fly. Our experimental evaluation using the Nobench dataset for JSON data, shows that we outperform Argo, a state-of-the-art data model that also maps the JSON data format into relational databases, by a factor of 3. We also outperform Hyrise, a state-of-the-art vertical partitioning algorithm designed for in-memory databases, by 24%. Furthermore, our algorithm is able to achieve around 40% better cache utilization and 35% better TLB utilization. Our experiments also show that our partitioning algorithm adapts to workload changes within a few seconds.

I. INTRODUCTION

The JSON semi-structured data format has become very popular for data exchange and processing on the Web and across cloud applications; JSON is now commonly used to support most social networking sites, such as, Facebook and Twitter. Moreover, recent efforts towards supporting relational capabilities for accessing JSON data formats have shown promising results [1]–[4].

For example, fast query processing times on the order of seconds are reported for certain types of continuous data analytics queries on JSON data e.g., on Facebook data [2].

JSON supports powerful, yet flexible data representation, including hierarchical, dynamic, self-describing data structures. This creates challenges for mapping JSON data to relational representations, including because of data sparseness, i.e., a large number of NULL values for some attributes in their relational representation.

In this paper, we design, implement and evaluate a novel architecture-aware technique for supporting fast data analytics on JSON data stored in a relational representation. Our technique named Dynamic Vertical Partitioning (DVP), adapts the relational data layout on the fly in order to improve locality of access and memory footprint, hence cache and TLB behavior of the queries. Specifically, we dynamically partition a table-like data structure storing the overall data by sets of columns/attributes into smaller tables based on two criteria: awareness of workload access patterns and data sparseness awareness.

Previous performance studies of in-memory data processing [5], [6] have emphasized the importance of table layout for good cache and TLB behavior. Such works focus on either building accurate cache models for query processing [6], or retrofitting on-disk data layouts for adequate fit, in terms of data alignment and packing, to memory pages [5]. These approaches implicitly assume that deciding the data layout is either done apriori at design time [5], or is an expensive, one-time run-time operation [6]. In particular, the Hyrise vertical partitioning algorithm for relational workloads running on in-memory databases [6] builds an accurate cache model based on exhaustively computing performance estimates for all combinations of attribute aggregation versus segregation which results in a partitioning algorithm with exponential complexity in terms of the number of attributes.

In contrast, our DVP partitioning technique groups attributes that are used together in queries into the same partition (smaller table) based on a simpler, faster algorithm, with polynomial complexity. We group attributes that are accessed together in queries close together in main memory within smaller tables thus improving locality of access. DVP is also the first partitioning technique which takes into account data sparseness. DVP segregates sparse and non-sparse attributes into different partitions. Thus, objects with null values for all attributes in a smaller table can be omitted, improving cache behavior by reducing the overall memory footprint.

Towards the above optimization goals, our algorithm evaluates layout costs for partitioned tables based on query execution-time and frequency, selectivity, attribute sparseness i.e., attributes with many null values, and attributes accessed together, for all queries in the workload mix. Moreover, DVP dynamically adapts the relational data layout for JSON data based on dynamic changes in both workload and data structure, on the fly.

We provide an experimental evaluation comparing the performance of our in-memory database engine against the performance of a representative system for relational data layouts for JSON data called the Argo system [1] in two areas: i) data layout restructuring to match database workload characteristics and ii) providing relational support for real-time data analytics on JSON data. Specifically, we compare against the performance of data layouts and relational capabilities for JSON data, which were previously introduced in the representative Argo study [1], and also against a representative of the state of the art in vertical partitioning for traditional relational data.
in in-memory databases, the Hyrise database engine [6]. We also compare against the performance of standard row-based and column-based data layouts. We measure and document the resulting performance and space improvements, including in terms of cache and TLB behavior.

In our experimental evaluation we use the query set from the NoBench benchmark introduced in the Argo study. Our results show that our DVP algorithm can provide a recommendation for partitioning a workload with queries accessing the 1000+ attributes used in Nodench within a few seconds. At the same time our engine outperformed the average query performance of the Argo layouts by factors between 15 and 30, and the Hyrise partitioned layouts by a factor of 2.4, on average, in steady state. Our experiments also show that our incremental partitioning algorithm is flexible enough to allow for adaptations to workload changes, within seconds, on the fly.

II. BACKGROUND

The Java Script Object Notation (JSON) data model is geared towards flexible, lightweight data exchange among Web services. JSON has a simple, yet powerful data format and was introduced recently as an alternative to the more traditional Extension Mark-up Language (XML) data model for semi-structured (Web) data.

Semi-structured data is schema-less, meaning that the users do not have to define a predetermined schema upfront. This lack of schema provides more flexibility for users, because different objects can have different structures and properties; semi-structured data is self-describing; it contains the structure of the data along with the actual values, and attributes can dynamically appear/disappear in different objects. However, due to the traditional advantages of the relational model, i.e., querying, indexing, benchmarking, recent studies have focused on adapting JSON data to relational table-like representations. In this paper, we use the Argo system [1], the first system to develop a mapping of JSON to relational representation, as one of our baselines for comparison.

A. JSON Characteristics

Each JSON object is a set of structured key, value pairs for attribute name and value. JSON supports four primitive types for its attributes: String, Number, Boolean, and Null. Moreover, the JSON model includes dynamic typing, sparse data, nested objects and nested arrays with an arbitrary number of elements, as explained next.

Dynamic and Nested Data Types: In Figure 1, we show an example of a JSON object, in order to introduce JSON characteristics that make mapping JSON to relational representation difficult; we also show the Argo system for representing the JSON data of the sample object in relational format.

As we can see from the example in the Figure, JSON objects can be hierarchical, which means that there can be any number of levels of nested arrays and objects. This is shown in our example for the "employees" array. We see that the "employees" attribute is nested. While we expect that the records of the employees named "Mary" and "Sam" would be included separately as top-level objects, we see that the record of one of the employees named "Jim" is nested in the top level object/record named "John".

Moreover, we can see flexibility of the lack of a predetermined schema in our example i.e., that attributes can appear or disappear dynamically. The values for three of the attributes for the employee named "Jim" are explicitly included in the nested record, while five attributes are included in the top-level record.

Another example of flexibility of the lack of schema is that, in JSON, the values of an attribute can have different data types in different records. For instance, the attribute called "salary" has an integer value of 100K in the top level object/record, and a string value of "tier-1" in the nested record.

Data Sparseness: Due to the lack of an upfront schema, we may encounter attributes that only exist in some objects/records rather than in all of them; this is known as sparse attributes. Data sparseness may occur due to both sparse attributes and nested data types.

We define the attribute sparseness of an attribute as the percentage of the records (documents) that have non-null values for that attribute. We define the data sparseness of a data set as the average attribute sparseness of all the attributes of the data set. For example, a data set with 1% sparseness is a data set with an average 1% non-null values for its sparse attributes.

B. Argo System for Mapping JSON to RDBMS

All of the above characteristics make mapping from JSON to relational databases challenging. Due to data sparseness, flattening JSON data into a relational table can be extremely inefficient, because of the huge number of null values that must be stored. The Argo system was introduced as a mapping layer that transforms JSON objects to a relational format to enable storage of JSON objects in traditional relational databases. Argo provides two mapping methods: Argo1 and Argo3.

As shown in Table I, in the Argo1 design, each entry in the table uses a composite key consisting of a unique object identifier and the attribute key from the JSON object. Each record in the single Argo1 table has the following format: object_id, attribute name, long (numeric) attribute value, string attribute value, and bool attribute value (5 columns in total). The value stored in each table entry is the attribute value of the given type from the record, thus storing two null values for each attribute-value pair. Arrays are flattened by using one indexed array element per record in the key structure.

In order to optimize the storage space, Argo3 stores different data types in three separate tables, one for each data type, (long) number, string, and boolean, respectively. We show an example of one of these tables, the one for storing the string data types, in Table II. Similar to Argo1, the first two columns store the object IDs and attribute keys while the last column stores the attribute values. However, this format allows us to omit the null values stored in the Argo1 table format.
C. Data Layouts: Column-based, Row-based, and Hybrid Vertical Partitioning

Data layout is the information that determines how data must be placed in different tables.

We call a vertical partitioning technique a technique that splits the attributes of a single table into multiple narrower tables, named partitions. By reducing the number of attributes in each table, a specific query may potentially skip accessing some of the tables. This skipping leads to lower I/O costs in disk-based databases, as well as better cache locality in main memory databases.

A partition can contain 1 to N attributes, where N is the total number of attributes in the data set. Two special cases of vertical partitioning are i) row-based layout, where we have a single partition with all (N) attributes and ii) column-based layout, where we have N partitions, each with one attribute. We will refer to a layout with any number of partitions in between as Hybrid layout.

D. Trade-offs between Column and Row-based Layouts and Opportunities for Hybrid Layouts

In a row-based layout, attributes are stored adjacent to each other in memory, and several attributes would thus fit within a cache line. Hence, a row-based layout is suitable for data analytics workloads that access many attributes while selecting only a few records.

Conversely, in a column-based layout, the attributes representing a record would be spread out far apart in memory. Therefore, a column-based layout is typically used for data analytics workloads that select and aggregate a large volume of records while accessing a small number of attributes.

We show these trade-offs in terms of cache behavior in the following examples.

From the perspective of data sparseness, row-based formats introduce substantial storage waste for storing null values of sparse attributes. Column-based formats may omit null values at the cost of replicating the object identifier for each column stored, resulting in a more compact representation overall.

Most of the current commercial main-memory database systems are using one of these two simple layouts: column-based or row-based. For example, Scuba [2] from Facebook uses a row-based approach and systems, such as, Dremel [7] and PowerDrill [8], from Google, all use a column-based approach.

To exploit the advantages of both layouts, some systems let the system administrator specify one of the column-based and row-based layouts [9], while others maintain copies of one table with both formats. For instance, the Oracle main memory system database [10] has a dual-format architecture that enables tables to be represented in main memory using both layouts simultaneously, and automatically sends each query to one of the copies to improve performance.

However, for workloads with mixed characteristics, either extreme would miss opportunities for optimization thus motivating Hybrid approaches, such as our Dynamic Vertical Partitioning which we introduce next.

III. Dynamic Vertical Partitioning (DVP)

Given a set of attributes $A$ and a query set $Q = \{q_1, \ldots q_n \}$, our goal is to derive a partitioning of the attributes $A, P$ such that the cost of executing $Q$ on $P$ is minimized. Each query $q$ in $Q$ can have associated frequency $f(q)$ with which it participates in the workload. We propose a partitioning algorithm that evaluates a range of possible partitions (table layouts), and identifies the layout with the minimum workload cost. Our goal is to reduce workload execution time, while also reducing storage space as a secondary factor. Our proposed algorithm evaluates workload costs on different layouts based on both query and data characteristics such as query execution times, selectivity, and frequencies, attribute sparseness ratios (relative frequency of non-null versus NULL attribute values), and access patterns (which attributes are accessed together). Such statistics are commonly present in commercial relational database systems.

<table>
<thead>
<tr>
<th>Object ID</th>
<th>Key</th>
<th>String</th>
<th>Num</th>
<th>Bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>name</td>
<td>John</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>manager</td>
<td>null</td>
<td>null</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>salary</td>
<td>null</td>
<td>100K</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>institution</td>
<td>IBM</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>employees[0]</td>
<td>Mary</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>employees[1]</td>
<td>Sam</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>employees[2].name</td>
<td>Jim</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>employees[2].salary</td>
<td>tier-1</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>employees[2].employees[0]</td>
<td>Jack</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

TABLE I: Relational Table Layout in Argo1, one column per data type.

<table>
<thead>
<tr>
<th>Object ID</th>
<th>Key</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>name</td>
<td>John</td>
</tr>
<tr>
<td>1</td>
<td>institution</td>
<td>IBM</td>
</tr>
<tr>
<td>1</td>
<td>employees[0]</td>
<td>Mary</td>
</tr>
<tr>
<td>1</td>
<td>employees[1]</td>
<td>Sam</td>
</tr>
<tr>
<td>1</td>
<td>employees[2].name</td>
<td>Jim</td>
</tr>
<tr>
<td>1</td>
<td>employees[2].salary</td>
<td>tier-1</td>
</tr>
<tr>
<td>1</td>
<td>employees[2].employees[0]</td>
<td>Jack</td>
</tr>
</tbody>
</table>

TABLE II: Relational Table Layout in Argo3. Argo3 stores different data types in separate tables. We show an example for the string data type only.
database management systems for other purposes i.e., optimization of indexes or query plans.

Our partitioning algorithm is based on two main ideas. First, by grouping attributes that are accessed together in the queries of the workload and placing them close together in main memory, within the same smaller table, we improve locality of access and thus cache and TLB performance. Second, we partition sparse from non-sparse attributes in different smaller tables; if the attribute values in a smaller table are all null, we omit storing them, thus reducing memory footprint.

Let \( c(q) \) be the cost of executing query \( q \) given partitioning \( \mathcal{P} \) of \( A \). We seek to identify the partitioning \( \mathcal{P} \) such that \( \min_{\mathcal{P}} \sum_{q \in \mathcal{Q}} c(q) \). Assuming that each attribute is a node in a graph and the set of attributes accessed by a query is a hyper-edge, it is easy to see that such a setting defines an instance of a hypergraph partitioning problem [11], which is known to be NP-Hard. For this reason, we employ effective empirical methods with the design objective to scale to very large number of attributes \( |A| \) and also derive fast the new cost effective partitioning \( \mathcal{P} \) under real-time workload changes.

The algorithm has two main components: an iterative search procedure and a cost model. Our iterative search starts with a current layout or an informed initial partitioning based on query characteristics and attribute access patterns (see section III-D), and repeatedly modifies the partitions to generate the final layout. Our cost model evaluates candidate layouts in each iteration of the search. We describe our search algorithm in subsection III-B and present our cost model in subsection III-C.

To identify an optimal layout, we map the problem to a graph partitioning problem. Each attribute of the data set is a node in the graph and each partition in the layout is a partition of the graph. Attributes that are accessed in the same queries are connected to each other in the graph by a weighted edge. Weight of edges are assigned based on workload and data characteristics such as selectivity and frequencies of queries as well as data sparseness. Once this mapping is completed, our goal is to identify a graph partitioning that minimizes our cost model. Typically in partitioning problems the number of partitions is fixed (k-way partitioning) and the cost model that is adopted and optimized is the summation of weights of the edges that lie between two partitions. In our setting we do not know the number of partitions in advance and the number of partitions can change depending on workload changes. Such a cost model will not suit our purposes as it would trivially place all attributes in the same partition. We adopt a different approach, assign penalties to partitions based on the number of attributes that belong in the partition but not accessed by the same query; we detail our cost model in section III-C.

A. Intuitions behind Partitioning

In this section we describe scenarios that motivate our partitioning algorithm. Our algorithm takes three aspects of a query into consideration for partitioning: the attribute set and the selectivity set of the query and the sparseness of the attributes accessed by the query.

In a given query generally there are two dimensions for selection. On the vertical domain, we have the attribute set that represents the attributes being accessed in the query. On the horizontal domain, we have the selectivity set that represents the records (rows) being accessed. Depending on the size of these two sets and the positions of the set elements in the database, we develop methodologies for partitioning that result in better cache utilization.
1) **Sparse Attributes:** If the table under consideration has sparse attributes, chunking those attributes into smaller partitions results in better performance because the objects that do not have any value for the attributes of a table are not inserted into it. Thus, the number of null values is lower in tables with fewer attributes. Hence, the minimum number of null values appears in layouts with very small partition sizes (1-6). However, the performance of layouts with these small partition sizes suffers from the overhead of accessing too many partitions.

Figure 3 shows the performance impact of different partition sizes for a query that has a large attribute set (all attributes) and also has 25% of attributes in the selectivity set resulting from a where condition. The sweet spot for this data set is somewhere between 6-12 attributes. Our algorithm automatically finds this sweet spot using the cost model described in the next section.

![Execution Times of Query](image)

Fig. 3: Execution times of query "Select * Where", with 25% selectivity, over layouts with increasing partition sizes of up to 120 attributes.

**B. Search Algorithm**

We aim to design a working solution that supports a very large number of attributes and derives the partitioning, in real time, under dynamic workload changes. Thus, we propose algorithms that start from a current layout, map it to the graph partitioning problem, and refine the partitions, deriving an updated layout.

To dynamically adapt the layout to data or workload changes, each time the partitioning algorithm is triggered, it starts with the current layout and optimizes it rather than generating a new set of partitions from scratch. Generating the next layout by refining the current one has the advantage of both faster convergence of the algorithm and lower cost in recreating tables.

Our approach is incremental and cost based; Our DVP algorithm starts from a current layout (or an initial partitioning) and at each iteration examines all existing attributes and partitions; for each attribute-partition pair, we calculate the gain of migrating the attribute to that specific partition. We iterate until no further cost improvement with a cap in the total number of iterations.

Let \( P_1 \) and \( P_2 \) be two partitions differing only in the placement of a single attribute. We define the gain between these two partitions as \( |\sum_{q \in Q} c^{P_1}(q) - \sum_{q \in Q} c^{P_2}(q)| \). At the end of the iteration, the algorithm chooses the attribute that would provide the maximum gain, and actually migrates this attribute to its new partition. The algorithm completes when the maximum gain during an iteration is less than or equal to zero, or it exceeds maximum number of iterations. (Algorithm 1).

**Algorithm 1:** Partitioning Algorithm

```plaintext
Input: attributes, queries, [initial_layout]
Output: new_layout: list of attributes in each partition
if initial_layout is not available then
   Call Initial_Partitioning to get initial_layout
foreach attribute pair \((a_1, a_2)\) in attributes do
   Calculate their connecting edge's weight (Equation 7)
   current_layout = initial_layout
   do
      max_gain = -1
      Calculate clc (current_layout_cost) by Equation 9
      foreach attribute \(a\) in attributes do
         foreach partition \(p\) in layout do
            Temporarily change \(a\)'s layout to \(p\)
            Calculate nlc (new_layout_cost) by Equation 9
            if clc - nlc > max_gain then
               max_gain = clc - nlc
               moving_attribute = a
               target_partition = p
            if max_gain > 0 then
               Migrate
               the moving_attribute to its target_partition
               Update current_layout
         if max_gain > 0 & num_iterations < max_num_iterations;
         new_layout = current_layout
   while
```

**C. Cost Model**

In this subsection, we detail the cost function we adopt and describe all its parameters. During query execution, a query accesses different attributes from various tables. But not all attributes of all tables accessed are required. For the remainder of this document, we will refer to these as redundant attributes and the cost of accessing them as redundant access cost. Since redundant attributes are part of the accessed tables, they will be brought into the cache by the query while actually processing necessary attributes. Redundant attributes are wasting cache space and reduce our ability to fully benefit from cache locality; it’s important for a partitioning approach to avoid these redundant attributes as much as possible.

By placing each attribute in a separate partition, one can avoid redundant access cost completely. However, the resulting layout (column-based) penalises queries that involve ad-hoc subsets of attributes for analytical purposes. Since such queries have to access several tables, the number of cache misses is higher than for the row-based layout for low selectivity (highly
selective) queries. As an example consider a query that selects 8 attributes with 10% selectivity: with a column-based layout, the query requires access to 8 different tables and since the selectivity value of the query is low, the column value accesses will be spread apart and not fit in the same cache line. Further reducing the cache efficiency, it is unlikely for consecutive records to be selected because of the low selectivity (meaning the query is highly selective). In contrast, with a row-based layout, if the query condition holds for a record, all 8 attribute will be selected with a lower cost because they are next to each other in memory.

We refer to this overhead as cross partition cost, because it is the price that a query pays while accessing different partitions. As a result an efficient partitioning algorithm should solve the trade off between redundant cost of accessing some attributes of wide partitions and cross partition cost of going through many narrow partitions.

Our cost function evaluates a layout by estimating redundant access cost and cross partitioning costs for executing a given workload. To estimate these costs, we consider query frequency, and selectivity as well as attribute access patterns and sparseness ratio. Below are the key terms used throughout to quantify the estimation.

- \( sel(q,a) \) is the selectivity of query \( q \) for attribute \( a \) (Equation 1).
- \( sel(q,p) \) is defined as the maximum selectivity of query \( q \) for all attributes of partition \( p \) (Equation 2).
- \( spa(a) \) is the sparseness ratio of attribute \( a \) (Equation 3).
- \( spa(p) \) is the maximum sparseness ratio of all attributes in partition \( p \) (Equation 3).
- \( rac(q,p) \) is the redundant access cost of query \( q \) over partition \( p \) (Equation 4).
- \( condition\_part(q) \) represents the attributes in the condition part of the query \( q \).
- \( selection\_part(q) \) represents the attributes in the selection part of the query \( q \).

Let \( f(q) \) be the frequency of the query in the workload and \( has\_Attr(q,p) \) be a boolean function that is true if and only if query \( q \) accesses any attribute from partition \( p \). \( rac(q,p) \) shows the redundant access cost of query \( q \) over partition \( p \) and \( RAC^P \) in Equation 5 represents the total cost of accessing redundant attributes, for all queries, over all partitions in \( P \). \( has\_Attr(q,p) \) in Equation 4 shows that \( rac(q,p) \) is zero if query \( q \) does not access any of the attributes in partition \( p \).

Equation 4 further indicates for query \( q \), there are two types of attributes belonging to partition \( p \) that increase \( rac(q,p) \): attributes that are never accessed by the query, and attributes that are accessed by the query, but at a lower frequency in comparison to other attributes of the partition. For example, if attribute \( a \) \( \in \) \( selection\_part(q) \), then query \( q \) accesses attribute \( a \) with frequency \( sel(q,a) \times f(q) \), while if \( a \) \( \in \) \( condition\_part(q) \), then query \( q \) accesses attribute \( a \) with frequency \( f(q) \).

The summation in Equation 4 shows that the impact that these types of attributes have on \( RAC^P \) is proportional to \( spa(p) \times sel(q,p) \) minus \( spa(a) \times sel(q,a) \). Specifically, larger differences imply more null values or more values that are non accessed as part of the query, which lead to poorer cache locality.

\[
sel(q,a) = \begin{cases} 
1 & a \in condition\_part(q) \\
\min\{sel(q,a) \} & a \not\in \{q\} 
\end{cases} 
\]

\[
sel(q,p) = \max \{ sel(q,a) \mid a \in p \} 
\]

\[
spa(p) = \max \{ spa(a) \mid a \in p \} 
\]

\[
rac(q,p) = has\_Attr(q,p) \times f(q) 
\]

\[
RAC^P = \sum_{q \in \mathbb{Q}} \sum_{p \in \mathbb{P}} rac(q,p) 
\]

Placing each attribute separately in a different partition eliminates the \( RAC^P \). However, the overhead of accessing several tables for queries that need accesses to different attributes can be unacceptably high. Therefore we need to quantify the cost of accessing different tables and attributes.

Let \( w(a,b) \) represent the benefit of keeping attributes \( a \) and \( b \) in the same partition. We assign values to \( w(a,b) \) based on workload information \( (spa(a), spa(b), sel(q,a), sel(q,b)) \). We then map the problem to a graph partitioning problem, where attributes are graph vertices, and the weight of an edge between an attribute pair is \( w(a,b) \).

Equation 7 shows the formula for calculating \( w(a,b) \) of an edge between two arbitrary attributes \( a \) and \( b \), where \( Q_{ab} \) is the set of queries that are accessing both \( a \) and \( b \) together. Note that \( w(a,b) \) is the penalty cost that a layout would pay if it decides to map \( a \) and \( b \) to different partitions. Thus, the \( w(a,b) \) is higher when \( spa(a) \) and \( spa(b) \) are close and \( \sum_{q \in Q_{ab}} f(q) \times \min\{sel(q,a), sel(q,b)\} \) is also higher (\( a \) and \( b \) are accessed by many queries like \( q \) with large \( f(q) \) and similar \( sel(q,a) \) and \( sel(q,b) \)). Moreover, placing sparse and common attributes in the same partition introduces null values in the equivalent table which waste cache space while running queries.

Equation 8 presents the total Cross Partition Cost (\( CPC^P \)) for the entire workload over all partitions in \( P \) as the summation of the \( w(a,b) \) values of all attribute pairs that belong to different partitions.

\[
Q_{ab} = \{ q \in \mathbb{Q} \mid a \in q \land b \in q \} 
\]

\[
w(a,b) = \frac{w(a,b)}{\max(spa(a),spa(b))} \times \sum_{q \in Q_{ab}} f(q) \times \frac{\min\{sel(q,a), sel(q,b)\}}{\max(spa(a),spa(b))} 
\]
\[ CPC_P = \sum_{a, b \in A} \{ w(a, b) \mid p_a \neq p_b \} \] (8)

So far, we have introduced \( RAC_P \) and \( CPC_P \) as the overhead of having redundant attributes while accessing partitions in \( P \) and the overhead of accessing several number of partitions from \( P \) during query executions. Equation 9 presents the total Cost, \( C_P \), of running the workload over \( P \). We define \( C_P \) as the average of the normalized values of both \( CPC_P \) and \( RAC_P \). Normalization is necessary for mapping the absolute values of \( CPC_P \) and \( RAC_P \) to the \([0 \ldots 1]\) interval for a meaningful summation. The \( \alpha \) coefficient is a workload-dependent parameter that represents the relative importance of \( CPC_P \) and \( RAC_P \) for the specific workload. Note that \( CPC_P \) and \( RAC_P \) get their maximum possible values in column-based and row-based layouts respectively.

\[ C_P = \alpha \times \frac{CPC_P}{CPC_{max}} + (1 - \alpha) \times \frac{RAC_P}{RAC_{max}} \] (9)

D. Initial Partitioning

When generating the partitioning (table layout) for the first time one needs to determine the initial partitioning to start from, which is the one the algorithm will start operating on. In such cases two obvious candidates are the row-based and column-based layouts. However, since the number of attributes in our case is in the order of thousands, having a single table or more than one thousand tables will not lead to an ideal layout. This is because, when a query accesses only a few attributes in a table with say one thousand attributes, the attributes accessed have a higher probability of being far from each other, which prevents the query from taking advantage of spatial cache locality. On the other hand, when a query needs to access many attributes for a few records, in a row-based format, attributes of the selected records are grouped together in memory, while in a column-based format, the selected attributes within each table will be far from each other.

Therefore, either of those extremes, is far from the ideal number of tables; convergence would be slow and more likely lead to a local minimum near the inefficient extreme initial layout. For these reasons, we designed a new technique to generate the initial layout that is better suited for a wide variety of use cases including the two extremes.

Our approach starts by sorting queries in \( Q \) according to their workload frequency, namely \( f(q_i), \forall i \). For each query \( q \), all of its accessed attributes that are not already assigned to any partitions are placed together into one new partition. After iterating once through all the queries, the only unassigned attributes are those that are not accessed by any queries. Storing these attributes separately, in any format, has no impact on workload execution time. We choose to store them in column-based format because, if any of these unused attributes starts to be accessed by queries and we need to take it out of the unused partition, no change in layout is required for the attributes remaining unused.

IV. Prototype Implementation Details

Our system is implemented in C++. Our main memory tables contain fixed-size attributes. To store a string value, we keep the actual value in a hash dictionary and store its equivalent hashed integer in the memory table. Nested objects and arrays are flattened before being stored.

Cache Collision Prevention: the starting addresses of data tables are aligned to page size addresses in order to exploit TLB entries as much as possible. Since the number of sets in the L1 cache is a divisor of page size, this alignment causes similar offsets of different tables to be placed into the same cache line [6]. For example, the first 64 bytes of all tables will be placed into set 0 of the L1 cache. This means that only a limited number of tables can be accessed at the same time (up to a number equal to the associativity of the cache). To overcome this problem, the Data Manager allocates a data table in page size-aligned addresses that are shifted by one cache line size compared to the previous data table. By doing so, we can concurrently access a number of tables equal to the associativity of the cache \( \times \) the number of sets in the cache, without introducing any cache collisions between different attributes of the same record that are in different tables.

Narrow Padding: In some cases, it is beneficial to add padding to the end of the records of a table in order to make their size cache-friendly; in other cases, padding will introduce extra cache misses in addition to its memory overhead. In order to figure out whether padding would be beneficial or not for a specific table, we predict the total number of cache misses for all possible simple projection queries over the table. If the average cache miss per record is lower for the padded version, then we add the padding; otherwise, we leave the table unpadded. We use the projection miss formulas from Hyrise [6] to predict the number of cache misses for the projection. The padding size is calculated by using Equation 10, where \( CLS \) and \( RS \) stand for \( cache \_line \_size \) and \( record \_size \), respectively.

\[ Padding \_Size = CLS - (RS \% CLS) \] (10)

Indexing, Scanning, Insert: To keep track of attributes from the same object in different tables, an object ID is assigned to each object that is stored in the engine. The object ID is replicated in every table. Since the object ID is assigned to objects during the insertion, records in the tables are sorted by object IDs. If an object doesn’t have non-null values for any attributes of a given table, the object is not inserted into the table. Therefore, tables with sparse attributes will have missing object IDs. Indexing is provided for the object IDs as primary keys. To avoid extra cost of joins while retrieving attributes of an object from different tables, the tables are scanned simultaneously using the sorted object ID column. When dynamic repartitioning is in progress, we use batching and bulk insert on the main table. The repartitioned table catches up with the main table quickly after the bulk operations on the main table. Data aging aspects are left for future work.
V. EXPERIMENTAL SETUP

All of our experiments are performed on an Intel(R) Xeon(R) machine with CPU E5-2650, running at 2.00GHz, equipped with 32KB L1D cache, 256KB L2 cache, a shared 20MB LLC (last level cache) cache, and a 32GB main memory. Caches are 8-way set associative and cache line size is 64B.

A. NoBench

In our experiments we generate query workloads based on Nobench, a benchmark for JSON data introduced in the Argo study [1], with two modifications to the NoBench query set in order to represent queries that access both sparse and non-sparse attributes at the same time.

Each document (object) used by NoBench has 19-25 attributes; there are 1019 attributes in total. Each object has a few attributes of various types, e.g., string, boolean and long, followed by a nested object with two attributes, a nested array with length 0-7, two dynamic type attributes, and 10 sparse string attributes. We use the 1% data sparseness configuration which conforms to NoBench data generation [1]. NoBench generates 100 groups of sparse attributes, with each group containing 10 attributes, hence 1000 sparse attributes total. A group is randomly assigned to each JSON object in the data set and all 10 attributes of the group get non-null values in the object. This corresponds to a 1% data sparseness for the overall NoBench data set.

To generate a workload with uniform query distribution based on the NoBench query set, equal frequencies are assigned to all queries. Then a query log of 1000 queries is randomly populated based on the query set.

While results in other configurations are not included, due to space concerns, we have also experimented with 5% sparseness and with other query workloads in the query mix e.g., random distribution. The results for all configurations we experimented with are similar. In general, our scheme will benefit more from higher sparseness degrees compared to schemes that do not consider sparseness.

Table III shows the set of queries used by NoBench to generate workloads of various types that cover a plethora of real world scenarios, as follows.

Q1-Q4 are projections accessing different attributes, including sparse, non-sparse, and flattened nested objects. We replace the nested_obj.num in the original Q2 with sparse_300 to present the case of projecting non-sparse and sparse attributes together. Q5 is a selection of a single record based on a specific value. Q6-Q9 are selections with 0.1% selectivity. All cases of having sparse, non-sparse, dynamic type, arrays and common attributes for the condition values are covered in these queries. In Q8 we replaced * with num, sparse_330, to examine the case of selecting sparse and non-sparse attributes without selecting all attributes. Q10 and Q11 are aggregation and join queries, and Q12 is a bulk insert query, that inserts objects from a given file into tables.

All reported results are an average of 5 independent runs. The time to retrieve actual string values from our dictionary table is not reported in any of the results since it is the same for all of the layouts.

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>SELECT str1, num FROM nobench_main;</td>
</tr>
<tr>
<td>Q2</td>
<td>SELECT nested_obj.str1, sparse_300 FROM nobench_main;</td>
</tr>
<tr>
<td>Q3</td>
<td>SELECT sparse_110, sparse_119 FROM nobench_main;</td>
</tr>
<tr>
<td>Q4</td>
<td>SELECT sparse_110, sparse_220 FROM nobench_main;</td>
</tr>
<tr>
<td>Q5</td>
<td>SELECT * FROM nobench_main WHERE str1 = XXXXX;</td>
</tr>
<tr>
<td>Q6</td>
<td>SELECT * FROM nobench_main WHERE num BETWEEN XXXXX AND YYYYY;</td>
</tr>
<tr>
<td>Q7</td>
<td>SELECT * FROM nobench_main WHERE dyn1 BETWEEN XXXXX AND YYYYY;</td>
</tr>
<tr>
<td>Q8</td>
<td>SELECT sparse_330, num FROM nobench_main WHERE XXXXX = ANY nested.arr;</td>
</tr>
<tr>
<td>Q9</td>
<td>SELECT * FROM nobench_main WHERE sparse_300 = YYYYY;</td>
</tr>
<tr>
<td>Q10</td>
<td>SELECT COUNT(*) FROM nobench_main WHERE num BETWEEN XXXXX AND YYYYY GROUP BY thousandth;</td>
</tr>
<tr>
<td>Q11</td>
<td>SELECT * FROM nobench_main AS left INNER JOIN nobench_main AS right ON left.nested_obj.str = right.str1 WHERE left.num BETWEEN XXXXX AND YYYYY;</td>
</tr>
<tr>
<td>Q12</td>
<td>LOAD DATA LOCAL INFILE file REPLACE INTO TABLE table;</td>
</tr>
</tbody>
</table>

B. Hyrise: A Main Memory Database with Hybrid Vertically Partitioned Layout

Hyrise (2010) [6], [12] is a main memory database that supports hybrid layouts. It uses an accurate cache miss model to calculate the estimated runtime of all queries in the workload for a given layout. Hyrise uses an exhaustive search to find the layout with the minimum estimated execution time among all possible layouts. Even though the Hyrise layout generator performs an exhaustive search, which is of exponential complexity in number of attributes, it scales up to a few hundred number of attributes by implementing different optimizations [6]. Hyrise scales up to a few hundreds of attributes by pruning parts of the search space; however, it is still unable to generate a layout for a dataset with 1000 attributes as we have in our benchmark. We ran the Hyrise layout generator on the Nobench dataset and the program did not terminate even after several hours of execution. We eventually had to halt the program.

VI. EXPERIMENTAL RESULTS

In this Section, we compare the performance of the DVP-generated layout and five baseline layouts: row-based and column-based layouts as two common layouts that are being used in most commercial databases [2], [9], [13], [14], Argo1 and Argo3 layouts that are designed specifically for JSON data, and finally Hyrise, a state of the art partitioner for in-memory relational databases.

In the first set of experiments, we compare the execution time of populating in-memory tables of the engine using different layouts in Section VI-A, we show different measurements of the execution times in Section VI-B, and we present the cache and TLB misses for all engines in Section VI-C. In the second set of experiments, presented in Section VI-D, we...
showcase our model’s ability to adapt to a changing data set, by evaluating its performance with and without repartitioning in the presence of changing input data.

A. Layout Overview and Analysis

Table IV shows a detailed characterization of memory consumption for all layouts studied, in terms of number of tables, the total amount of occupied memory, the total amount of null values. The table also presents the total table building and populating time for each layout studied.

Overall, we can see that our DVP approach uses approximately 10% of the total tables used by the column-based layout, which uses 1019 tables (one for each attribute). All other approaches use very few tables, with the row-based layout and Argo 1 at the extreme of placing all attributes in a single table.

In terms of NULL values, the row-based layout stores the highest number of NULL values, resulting in the highest memory consumption overall. At the other extreme, the column-based layout stores no null values, but each object id is replicated in each table, which causes 50 percent storage overhead.

DVP uses the least total memory across all schemes because it takes data sparseness into account, and segregates sparse and non-sparse attributes, thus reducing the number of null values stored. Overall, DVP uses only 5%, 3% and 3.5% of the memory storage space that Argo3, Argo1 and Hyrise use, respectively.

As shown in section II, in Argo1, each record has 5 values, and for each record, exactly one of the columns has a non-null value and that’s why 40% of the values are null (2 out of 5 values per record). Other sources of storage overhead are replicating object ids for each attribute of each object, and storing the hashed form of attribute names in each record.

Argo3 uses three tables, one table for each type of data, and each record uses the following format: object id, attribute name, attribute value. Argo3 avoids storing null values by keeping different types of attributes in separate tables, but it still suffers from overhead due to replicating the object ids and the attribute names.

Hyrise uses 11 tables, which it customizes for attributes that are accessed outside of SELECT * queries. However, Hyrise doesn’t consider data sparseness, and consequently places all attributes that are only accessed in the * part of the queries in one single table, resulting in a very large number of null values (3.9 GB).

The time needed to build the layouts for the Argo engines is roughly 3.5x larger than for the other engines, because for the Argo implementations, each attribute is a record instead of each object, and the hashed values of attribute names are also inserted into tables.

The row-based and Hyrise layouts suffer from a large amount of null values (4 GB and 3.9 GB, respectively) and poor cache locality, due to writing the non-null values in far-apart locations within each record.

The column-based layout experiences overhead due to inserting data for each object into an average of 24 tables, while the hybrid DVP layout only inserts data into 7 or 8 tables despite having a total of 109 tables, and thus the hybrid approach has the best overall performance in terms of time needed to build the layout.

B. Execution Times

In this Section, we compare the execution times across different data models for our engine. Figure 4 shows the average query execution time for each query across all engines and Figure 5 shows a comparison of the total execution times for each engine.

Queries Q1 to Q4 are projection queries; for this type of query, the engine scans the entire table and retrieves the data from the desired columns. Argo 1 and Argo 3 are 4x-6x slower compared to all other layouts, mainly because, for Argo, each attribute is a row (record) in their table(s), so the tables have 20x-24x more records in comparison with tables with the other layouts. To retrieve attributes from tables in the Argo layouts, the engine scans the attribute-name column and, if the attribute name matches the name of the selected attribute in the query, it retrieves the attribute value. Scanning the entire attribute-name column for tables that are over 20 times larger is the reason for the poor performance of the Argo layouts.

For queries Q1 and Q2, the performance of hybrid, column-based, and Hyrise are the same. The row-based layout performs poorly for these queries because the retrieved values for an object are far (in terms of spatial memory locality) from those retrieved for the next object, and other attributes that are brought into the cache together with desired attributes, are not used.

Q3 and Q4 are projections on sparse attributes. The hybrid layout stores the three accessed attributes in one table, while Hyrise and the column-based layout store each attribute in a separate table. Bringing into the cache one object-id per each retrieved attribute results in poorer cache utilization for Hyrise and the column-based layout, which in turn causes 24% and 33% longer execution times for Q3 and Q4, respectively. Similar to Q1 and Q2, the row-based layout suffers from poor cache utilization.

For query Q5, as it retrieves a single record, the execution time of the query mainly depends on scanning the condition column. The hybrid, column-based, and Hyrise layouts perform equally well because they all isolate the condition column in its own table. On the other hand, the row-based layout performs approximately 5 times worse because it pollutes the

<table>
<thead>
<tr>
<th>Layout</th>
<th>Tables</th>
<th>Size [MB]</th>
<th>Amount of NULLs [MB]</th>
<th>Build Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>1</td>
<td>4100</td>
<td>4000</td>
<td>86</td>
</tr>
<tr>
<td>Col</td>
<td>1019</td>
<td>168</td>
<td>0</td>
<td>98</td>
</tr>
<tr>
<td>Argo1</td>
<td>3</td>
<td>2700</td>
<td>1800</td>
<td>297</td>
</tr>
<tr>
<td>Argo3</td>
<td>3</td>
<td>2700</td>
<td>0</td>
<td>292</td>
</tr>
<tr>
<td>Hyrise</td>
<td>11</td>
<td>4000</td>
<td>3900</td>
<td>85</td>
</tr>
<tr>
<td>DVP</td>
<td>109</td>
<td>138</td>
<td>10</td>
<td>81</td>
</tr>
</tbody>
</table>

TABLE IV: Comparison of the characteristics of memory consumption across all layouts.
cache with unnecessary attributes while scanning the condition column.

Argo1 and Argo3 perform better for SELECT * queries than they do for projection queries, for the following reason:

For SELECT * queries, the engine scans the table(s) looking for the condition attribute name; once it finds the attribute name, it then reads the attribute value to evaluate the condition. If the condition is true, then the engine needs to read the remaining records of the object; however, because of the low selectivity of the queries, the condition is false 99.9% of the time. Since we support indexing for primary keys (object_ids), once the engine knows the condition is false, it no longer needs to scan the remainder of the object’s records, and instead it can just jump straight to the first record of the next object.

In contrast, for projection queries, the engine needs to find more than one attribute name, so it will continue scanning after it finds the first attribute name until either it finds all projected attributes, or it reaches the end of the current object. During this continued scanning, it will incur additional cache misses when compared to the behaviour for SELECT * queries.

Q6, Q7, and Q9 are SELECT * queries; for these queries, the Hyrise and DVP layouts outperform both the column-based and the row-based layout. The column-based layout has good cache utilization while scanning the condition column, but it suffers from a high number of TLB misses when the condition is true and the engine has to retrieve attributes from more than one thousand tables. On the other hand, the row-based layout has poor cache utilization while scanning the condition column, but it has very few TLB misses when it retrieves attributes of a record by simply reading a continuous chunk of memory. The Hyrise and DVP layouts experience the same good cache utilization as the column-based layout, without experiencing its high number of TLB misses caused by retrieving data from too many tables.

Q8 is a SELECT query that selects two columns. For this query, the row-based layout still suffers from poor cache utilization while scanning the condition column. Q8 evaluates up to 7 different condition attributes for each object (nested_array[0-6]). The column-based layout outperforms the Hyrise and DVP layouts by 28%, because it keeps each nested_array index in a separate table and avoids storing null values, while the other two layouts keep all the indexes of the nested_array in one table. Storing all of them in one table leads to storing some null values and to poorer cache utilization, because the length of the nested_array varies from 0 to 7 and not all objects have all of the indexes.

Q10 is an aggregation query; when running the query, the engine first executes the selection part of the query, and then it does the aggregation over the retrieved result of the selection part. The execution time for the selection part over different layouts is similar to the execution time of all other SELECT * queries. However, the execution time of the aggregation part is higher for the row-based and Hyrise layouts. This is because both layouts have many null values in the retrieved records of the SELECT * part, which in turn results in poor cache utilization when doing the aggregation part.

C. Cache and TLB Misses

In this Section, we compare the cache misses and the TLB misses experienced by all engines across all queries. Figure 6 shows the number of cache misses across all queries for each cache level (L1, L2 and L3 caches), while Figure 7 shows the number of TLB misses for all queries and across all engines.

1) Cache Performance: Figure 6 shows that both Argo1 and Argo3 experience a very high number of cache misses across all three cache levels. Argo3 has relatively fewer misses because it stores its data in three separate tables, while Argo1 stores all its data in a single table that has 40% null values. The poor cache utilization of these layouts is caused both by...
large table sizes and by the fact that they scan every attribute for each query (including projection queries).

The column-based layout has a poor overall cache utilization, even though it is similar to the Hybrid layout in terms of storage size. This is because in the column-based layout, for SELECT * queries, the engine has to retrieve each attribute from a separate table, and selected attributes are far apart within a table because of the low selectivity of the queries. Therefore, the column-based layout causes one cache miss for retrieved attribute for these queries. In particular, the L1 and L2 cache performance of column-based layout is as poor as that of the row-based layout, even though the latter has an order of magnitude more data stored. This is because of the thrashing that the L1 and L2 caches experience when a new line is brought into the cache every time an attribute is retrieved.

The row-based layout experiences the largest number of L3 misses. There are two reasons for this: first, the row-based layout has the largest total storage size across all layouts; second, this layout experiences one cache miss for each retrieved attribute for projection queries, and one cache miss for each scan of the condition column for SELECT * queries. Even though the Argo layouts also have large storage sizes, they have better L3 cache utilization because while running SELECT * queries, selected attributes are next to each other; in contrast, in the row-based layout, the selected attributes are spread out among more than 1000 attributes.

The Hyrise and Hybrid layouts have similar cache utilization for the L2 and L3 caches. They both experience a low number of cache misses for projection queries. Hyrise has a large single table holding all attributes that are only accessed in the selection part of SELECT * queries. Because of this large table, the Hyrise layout has a much larger amount of data than the Hybrid layout (by an order of magnitude), which is reflected in the significant number of L1 misses for Hyrise.

2) TLB Performance: Figure 7 shows that the column-based layout has the worst TLB miss performance among all layouts, as it needs to access the largest number of tables (1019) for all SELECT * queries. Argo1 and Argo3 experience the second-worst TLB performance, because for non-SELECT * queries, they cannot scan only the columns of desired attributes, but instead have to scan entire tables.

Even though both Argo1 and Argo3 store data in one or few continuous arrays, it’s difficult for the prefetcher to prefetch the TLB entries based on the page access patterns. The access pattern for these layouts is complex because once the value of the condition attribute is within the given range, it may be necessary to scan backward all the way until the beginning of the current object id, in order to retrieve the selected attributes of the object.

The row-based layout has the best TLB performance as it stores all its data in a single continuous array and, for all queries other than the JOIN query, the data access pattern is very simple. It’s straightforward to correctly prefetch the TLB entries.

Hyrise experiences 35% more TLB misses in comparison to Hybrid. This is because all attributes that are accessed only in the selection part of SELECT * queries, are kept in a large single table. The table is so wide that while running SELECT * queries, each selected row of the table falls in a different page and, due to the low selectivity of queries, the page addresses are not prefetched into the TLB. Note that this is not the case for the row-based layout because the condition attribute which has the fixed scanning pattern is kept within the same table as the selected attributes, and the page address is already prefetched for scanning the condition attribute.

D. Workload Adaptation

Figure 8 shows the moving average of query execution time for 1000 queries. At the moment indicated by the arrow in the figure, a change is injected into the workload (changing the accessed attributes and the conditions for some queries) to reflect the dynamic characteristics of the JSON data and workload. After the workload changes, repartitioning takes less than 3 seconds to refine the layout, define new table schemas and populate the new tables in DVP. Repartitioning is done by a background thread, with no downtime, and with thread binding on a different core, so we do not pollute the L1, L2 caches for query execution. The engine switches to using the
new set of tables through an atomic change. If the workload is stable for periods of time of more than a few seconds, then we benefit from the improved cache and TLB behavior. The graph shows that our DVP algorithm works 8-10% better by repartitioning for a small change in the workload. We expect to gain more when the changes are more substantial including changes in query frequencies in the workload or their selectivities or appearance of new queries, etc.

VII. RELATED WORK

Previous work related to our study falls in one of the following areas: 1) Partitioned and adaptive data representation in relational databases, 2) Main memory relational databases, 3) Optimizations for processing JSON data, including by relational mapping of the JSON data model, and 4) Main memory NoSQL databases.

A. Hybrid and Adaptive Data Representation for Relational Databases

Vertical partitioning has been studied extensively in the past. Traditionally, it was used mainly to minimize I/O cost in disk-based relational databases [15]–[17]. Past studies have addressed both horizontal and vertical partitioning [18], [19] and layout optimization in distributed databases [20], [21]. Recently, vertical partitioning has been studied in the context of achieving better cache utilization in main memory database systems [6], [22], [23].

Among other research areas directly related to our work is the area of data stores with adaptive representation [24]–[26].

In addition to vertical partitioning, the PAX [5] data layout groups together all values of each attribute within each page to improve cache utilization. However, it doesn’t perform well for queries that need to access many attributes and for tuple reconstruction [22], [23]. Data Morphing [23] and Trojan [22] are two extensions of the PAX model that use coarser-grain grouping to reduce the number of groups in each page/data block, which in turn leads to a lower tuple reconstruction cost [22], [23].

Alagiannis et al. [24] provide multiple storage layouts and data access patterns in a single engine. However, the authors employed redundancy of representation, which is not what we advocate in our work. Sun et al. [25] are concerned with skipping as an optimization, an improvement of previous work by the same authors, orthogonal to our approach. Arulraj et al. [26] employs a specific approach to segmenting, namely providing tiles and is concerned with transactional workloads and semantics. Our approach is suitable for non-transactional (JSON) data and is much more general, as it allows for greater flexibility in representation.

Most of the in-memory databases, which may support row-based and/or column-based layouts, are not designed to support semi-structured data, or they perform poorly on data-sets with many sparse attributes [2], [9], [13], [14], [27], [28].

B. Optimization of Processing JSON Data

There have been a few recent studies on optimization for processing JSON data, including by mapping of JSON data to relational databases [4], [29]–[31].

Some of these recent works focus on hardware acceleration for parsing JSON data [4], [29], [30]. This is orthogonal to our work. Other recent related work focuses on automatically discovering functional relationships across JSON objects towards building an overall schema [31]. The main focus of their optimization is on providing the user with the basis for a clear contextual understanding of the schema. In contrast, our primary focus is on grouping and restructuring of data for locality of access and documenting the architectural impact. Both approaches achieve a reduction in storage space by reducing redundancy in stored data.

The system we use for comparison, Argo [1], uses generic tables to store JSON data in a relational format such that every attribute of a JSON object is stored as a separate record. However, Argo exhibits poor performance when reconstructing JSON objects [32]. Our experimental results show that our system is 15x-30x faster than Argo for different workloads.

Oracle has recently published a paper on JSON data management in RDBMSs [32]. Each JSON object instance is stored as one aggregate object in a separate column, without any relational decomposition. In this design, a JSON object
collection is modelled as a table with one column storing JSON objects. Each row in this table holds a JSON object instance in the collection. The format for storing objects is adapted from document store databases. Therefore, OLAP queries will still have a poor performance when extracting few values from arbitrary complex objects at runtime.

Sinew [33] is an SQL system designed for support of multi-structured data. In Sinew’s data storage model, after flattening nested objects, each unique key becomes a logical column. To avoid having a table with hundreds or thousands of attributes, only a subset of attributes are transformed to physical columns while the remaining columns make a serialized binary column in the database. Users see a logical view of the database with all logical columns, and transforming queries from the logical view into queries over the physical layout is accomplished by using a mapping layer. The disadvantage of this scheme compared to our approach is that changing columns that are grouped into one serialized column can be very expensive.

C. NoSQL Databases

Many NoSQL databases have been designed for high-performance processing of semi-structured data, among them key-value and document (JSON) stores [34], [35]. However, key-value stores, such as Redis [36] and RAMcloud [37] do not support semi-structured data nor they can execute complex queries like joins and have poor performance for read-heavy queries [38]. In contrast, document stores can support more complex data than key-value stores [38]. But, they do not support nested documents.

VIII. CONCLUSIONS

In this paper, we present and evaluate a novel technique for partitioning of the database schema for optimizing performance of main memory databases.

Our partitioner considers workload characteristics, such as query frequency, selectivity and attributes accessed by each query towards generating a data layout to minimize TLB and cache misses during the workload execution. Our dynamic vertical partitioning algorithm (DVP) can be run with any kind of relational data. However, DVP is especially suitable for intelligently partitioning the large set of attributes commonly resulting by mapping JSON data into tables. Furthermore, our engine adapts to workload and data changes in real-time; if new queries or new attributes appear in the input, the DVP partitioner generates a new layout by incrementally refining the current layout.

Our detailed performance evaluation using query workloads adapted from the Nobench benchmark for JSON data shows that DVP outperforms unoptimized relational support for JSON by factors between 15 and 30, and a recent static vertical partitioning algorithm by a factor of 24% on average. Our algorithm is able to achieve around 40% better cache utilization and 35% better TLB utilization. Our adaptation of partitioning based on dynamic workload change results in 10% better query execution time.

REFERENCES


