Jigsaw: A Data Storage and Query Processing Engine for Irregular Table Partitioning

Donghe Kang
kang.1002@osu.edu
The Ohio State University
Columbus, Ohio, USA

Ruochen Jiang
jiang.2091@osu.edu
The Ohio State University
Columbus, Ohio, USA

Spyros Blanas
blanas.2@osu.edu
The Ohio State University
Columbus, Ohio, USA

ABSTRACT

The physical data layout significantly impacts performance when database systems access cold data. In addition to the traditional row store and column store designs, recent research proposes to partition tables hierarchically, starting from either horizontal or vertical partitions and then determining the best partitioning strategy on the other dimension independently for each partition. All these partitioning strategies naturally produce *rectangular* partitions. Coarse-grained rectangular partitioning reads unnecessary data when a table cannot be partitioned along one dimension for all queries. Fine-grained rectangular partitioning produces many small partitions which negatively impacts I/O performance and possibly introduces a high tuple reconstruction overhead.

This paper introduces Jigsaw, a system that employs a novel partitioning strategy that creates partitions with arbitrary shapes, which we refer to as *irregular* partitions. The traditional tuple-at-a-time or operator-at-a-time query processing models cannot fully leverage the advantages of irregular partitioning, because they may repeatedly read a partition due to its irregular shape. Jigsaw introduces a partition-at-a-time evaluation strategy to avoid repeated accesses to an irregular partition. We implement and evaluate Jigsaw on the HAP and TPC-H benchmarks and find that irregular partitioning is up to 4.2× faster than a columnar layout for moderately selective queries. Compared with the columnar layout, irregular partitioning only transfers 21% of the data to complete the same query.

CCS CONCEPTS

• **Information systems → Data layout**.

KEYWORDS

Irregular partitioning

ACM Reference Format:
Donghe Kang, Ruochen Jiang, and Spyros Blanas. 2021. Jigsaw: A Data Storage and Query Processing Engine for Irregular Table Partitioning. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457547

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457547

1 INTRODUCTION

Optimizing the physical layout of a database can significantly accelerate the performance of query processing. Prior work has extensively evaluated the trade-offs between row store and column store designs [1, 15, 16, 37]. The benefit of column stores for OLAP workloads is higher I/O efficiency: a row store reads unnecessary data when a query does not access all the attributes in a tuple.

I/O efficiency can further improve by partitioning the table and determining the best storage option independently for each partition. The partitioning strategies in prior work focus on partitioning either tuples (horizontal partitioning [8, 11, 43]), attributes (vertical partitioning [5, 13, 42]), or first on one dimension and then on the other (hierarchical partitioning [7, 40]). By co-locating tuples (attributes) that are often accessed together, horizontal (vertical) partitioning improves I/O efficiency by not accessing redundant data when evaluating queries. Careful partitioning is particularly effective in queries with low selectivity (projectivity).

The partitioning strategies in prior work produce rectangular partitions, which have some disadvantages. First, queries will still read redundant data when the access pattern cannot be naturally partitioned in one dimension for all queries. (For example, when partitioning vertically and the most recently added tuples in the table are accessed differently than the rest of the table.) Second, even when queries access disjoint parts of a table, rectangular partitioning will inevitably read some redundant data when accesses to one column are conditional on the value in another column. Hierarchical partitioning can reduce redundancy if it creates small partitions. However, small partitions can lead to worse I/O performance and possibly higher overhead to reconstruct each tuple if data is scattered in different blocks in cold storage.

This paper describes Jigsaw, a system that uses a novel table partitioning strategy, *irregular* partitioning, that creates and manages partitions of arbitrary shapes. Jigsaw first creates fine-grained segments to match the access pattern of the query workload and then reorders and merges these segments into partitions to optimize I/O performance. The partitioning algorithm is a top-down method, which recursively partitions the table into segments. Smaller segments are created by splitting either horizontally or vertically, based on the expected I/O benefit. The algorithm splits segments until finer partitioning will not reduce the need to read unnecessary data any further. After partitioning into segments, the algorithm merges segments with a similar access pattern into partitions. Merging segments with different schemas produces the distinctive irregular shapes of partitions in Jigsaw.

To fully leverage irregular partitioning, Jigsaw uses a partition-at-a-time query evaluation strategy. This evaluation strategy avoids

repeatedly accessing partitions due to their irregular shape, which is an issue when employing traditional query processing methods, such as the *tuple-at-a-time* [12] and *operator-at-a-time* [18] strategies which assume that the tuple or attribute order is the same across partitions. Jigsaw uses two parallelization strategies for the partition-at-a-time method: one based on locking (Jigsaw-L) and the other based on shared scans (Jigsaw-S).

Experiments with Jigsaw on the HAP [8] and TPC-H [41] workloads show that Jigsaw speeds up query processing by up to 4.2× compared to columnar partitioning for moderately selective queries. The speedup is due to the reduced I/O volume: columnar partitioning needs to read 4.7× more data than irregular partitioning to answer the same query.

The main contributions of this paper are:

(1) Introducing irregular partitioning and formulating the irregular partitioning problem as an optimization problem.
(2) Designing a top-down partitioning algorithm that simultaneously considers horizontal and vertical partitioning and then merges segments based on the similarity of their access pattern to eliminate small partitions.
(3) Developing a query processing strategy, *partition-at-a-time* evaluation, which is tailored to irregular partitioning and avoids repeated accesses to partitions. Two parallelization strategies process irregular partitions with multiple CPU cores.
(4) Implementing irregular partitioning in a system prototype, Jigsaw, and conducting an extensive experimental evaluation using the HAP and TPC-H benchmarks.

The remainder of the paper is structured as follows. Section 2 presents the table partitioning and query processing in previous works and analyzes the limitations. Section 3 describes the Jigsaw architecture. Section 4 formulates the partitioning problem and proposes a top-down partitioning algorithm which generates irregular partitions. Section 5 follows with more details about the data storage and partition-at-a-time query processing. Section 6 describes the experimental setup and evaluates the query performance and the partitioning algorithm performance. Section 7 presents related work and Section 8 concludes.

2 BACKGROUND AND MOTIVATION

2.1 Storage Models

The principle of data independence gives relational database systems flexibility in determining how data will be physically stored on the disk. Two orthogonal questions arise when determining how to best store a table: what is the data *order* and the data *layout*. Table 1 summarizes different storage choices from prior work.

Layout	Row order	Column order
Natural	PostgreSQL [32]	Sybase IQ [27], MonetDB [9]
Horizontal	Schism [11]	Casper [8], Parquet [6]
Vertical	Hyrise [13]	–
	H_2O [5]	
Hierarchical	Peloton [7], GSOP [40]	

Table 1: Data order and layout in prior work.

The **data order** determines which values will be stored contiguously in a page. Data is either stored in *row order* that serializes a table row by row or in *column order* such that one attribute is stored contiguously. Row store designs are superior for OLTP systems such as Microsoft Hekaton [24] and column store designs are superior for data warehouses such as MonetDB [9].

The **data layout** question determines how data is split into different pages. A number of strategies have been explored:

(1) Natural order layout: Data is sequentially stored in their natural order (row or column) and segmented into pages.
(2) Horizontal layout: Partition the table horizontally. Horizontal partitioning can be done by applying a hash or range function on a specific attribute [17] or can be based on access frequencies of the query workload [11]. Schism [11] partitions horizontally and stores partitions in row order, so that tuples that are frequently accessed together will be retrieved in a single I/O. Apache Parquet [6] partitions horizontally to *row groups* and stores partitions in column order.
(3) Vertical layout: Partition the table vertically by grouping frequently co-accessed columns together in a *column group* [5] (also *container* [13] and *projection* [23]). Columns in the same vertical partition are retrieved in a single I/O operation. H_2O determines vertical partitions and picks the optimal data order independently for each partition [5].
(4) Hierarchical layout: These designs first partition on one dimension and then on the other. Peloton implements a hierarchical storage engine [7] that horizontally partitions the table first, and then vertically partitions each horizontal partition. This allows different vertical partitioning strategies for each horizontal partition. Individual tiles in Peloton store data in either row or column order. GSOP [40] takes the opposite approach: it first determines the column grouping (vertical partitions) and then determines how to partition column groups horizontally to maximize the benefit of data skipping.

Limitations. Consider the table T shown in Figure 1a and the queries Q_1, Q_2, Q_3 shown in Table 2. Assume that evaluating predicates requires a full scan without index. Different shades of gray in Figure 1 indicate which values a query accesses.

The horizontal layout (shown in Figure 1b) places tuples accessed by the same query in one partition, such as t_1, t_2, t_4 to benefit Q_1. Redundant attributes are retrieved if partitions are stored in row-major order. Storing a partition in column-major order avoids reading redundant attributes but a partition may need to be visited multiple times to access additional columns: Q_1 will need to read a_1 in its entirety to evaluate the predicate in the query. Finally, note that grouping tuples in a certain way favors some queries at the expense of others: in the example shown in Figure 1b, although both Q_1 and Q_2 can skip one redundant partition when accessing a_2 and a_3, Q_3 needs to access both partitions to return the relevant values of a_5. In summary, the core limitation of the horizontal layout is that it cannot place attributes of one tuple to different partitions.

The vertical layout (shown in Figure 1c) places attributes accessed by the same query in one partition. However, a query may still need to read some redundant cells: for example, Q_1 reads a_2, a_3 of t_3, t_5, t_6. Second, queries may read multiple partitions because

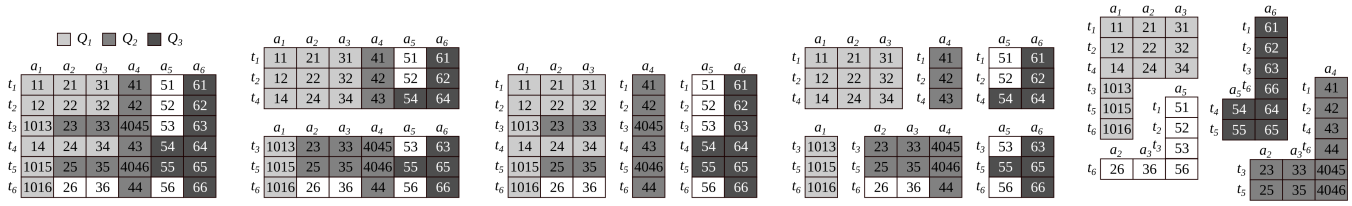


Figure 1: Example table T with different storage layouts.

Q_1	SELECT a_2, a_3 FROM T WHERE $a_1 \leq 1000$
Q_2	SELECT a_2, a_3 FROM T WHERE $4000 \leq a_4$
Q_3	SELECT a_5 FROM T WHERE $64 \leq a_6 \leq 65$

Table 2: Example queries on T.

vertical partitioning cannot always store all attributes of a query together: in the example shown in Figure 1c, Q_2 accesses two partitions to retrieve a_2, a_3, a_4. Producing the final output in this case requires reconstructing rows from multiple partitions during query evaluation (see Section 2.2 for more details).

The hierarchical layout splits tuples and attributes into different partitions. Figure 1d first horizontally splits tuples into two partitions (same as Figure 1b) and then vertically splits attributes in each partition independently. This two-dimensional partitioning approach improves I/O efficiency by minimizing the redundant data each query reads. The main weakness of hierarchical partitioning is that it generates too many small partitions. Whereas the horizontal layout (see Figure 1b) issues large sequential I/O requests, the hierarchical layout issues much smaller I/O requests and accesses different partitions a lot more frequently. Generating the hierarchical layout by splitting along one dimension at a time means that the partitions are still regular (that is, either horizontally or vertically aligned), which inherits the limitations of horizontal or vertical layouts albeit at a smaller scale.

Irregular partitioning. In order to overcome the limitations of the existing layouts, we propose to partition a table into irregularly-shaped fragments. Figure 1e shows the layout of table T with irregular partitioning. Irregular partitioning can store different schemas in the same partition: the top left partition in Figure 1e stores attributes a_1, a_2, a_3 for t_4 but only a_1 for t_3. Note how Jigsaw reorders tuples to avoid unnecessary schema changes within a partition: for example, the top left partition stores t_4 before t_3. As a result, the irregular layout matches the efficiency of the hierarchical layout in terms of data redundancy, but without creating the small partitions of the hierarchical layout which underutilizes I/O bandwidth.

Within an irregular partition, Jigsaw stores data in row-major order to avoid materializing intermediate data when queries access a single partition. (The irregular partitioning algorithm that is described in detail in Section 4 strives to make accesses to multiple partitions a rare occurrence.) Column-major ordering would require processing data attribute by attribute and materializing intermediate results: computing $a_1 + a_2 + a_3$ in the column-major MonetDB system, for example, materializes the result of $a_1 + a_2$ and then adds it to a_3. Section 6.3.2 experimentally shows the benefit of row-major ordering when processing in-memory data.

2.2 Query Evaluation

The *tuple-at-a-time* and *operator-at-a-time* are two query evaluation strategies that are tailored for row stores and column stores, respectively. The tuple-at-a-time strategy evaluates predicates and projects attributes as a tuple passes through different operators [12]. Processing a batch of tuples at a time can amortize the cache misses and branch mispredictions associated with switching between operators [9]. The operator-at-a-time model in column stores completes one operation at a time: the selection operator only tracks which tuples satisfy the predicate and the projection operator materializes the result [18].

Limitations. With irregular partitioning, both the tuple-at-a-time and operator-at-a-time query evaluation strategies may read a partition by multiple times; in the worst case, a partition may need to be revisited as many times as the number of tuples in the partition. The fundamental reason is that irregular partitioning may naturally change the ordering of tuples across attributes in different partitions. For example, the irregular layout shown in Figure 1e stores attribute a_6 in one partition, while the attribute a_2 is located in two different partitions in different tuple orders. If a query plan accesses tuples in the storage order of attribute a_6, adjacent attributes in the same order may be scattered in other partitions. As an example, consider the evaluation of the query SELECT a_2 FROM T WHERE $62 \leq a_6 \leq 65$ on the irregular layout in Figure 1e. Evaluating the predicate on a_6 would require accessing the tuples in the order of t_2, t_3, t_4, t_5 for attribute a_2, which would require four I/O operations each for a single value. Both the tuple-at-a-time and operator-at-a-time evaluation strategies would require repeated random accesses to the other partitions, which would be prohibitively expensive if the memory is not large enough to hold all accessed partitions in their entirety.

Partition-at-a-time evaluation. To avoid repeated accesses to a partition, Jigsaw adopts a partition-centric evaluation strategy that exhausts one partition before moving to the next partition. This evaluation strategy is crucial to fully realize the benefits of irregular partitioning. We refer to this strategy as a *partition-at-a-time* strategy, and describe it in detail in Section 5.2.

3 SYSTEM ARCHITECTURE

We design a prototype engine, Jigsaw, that partitions a table irregularly and evaluates queries by reading one partition at a time. Figure 2 shows the system architecture. Jigsaw has three main components, the *query processor*, the *partition tuner* and the *partition manager*. Given a set of queries, the partition tuner generates a

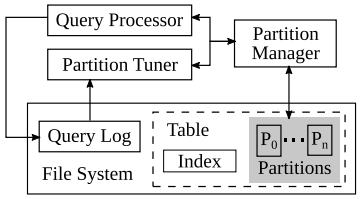


Figure 2: The Jigsaw system architecture.

partitioning plan $\mathbb{P}$ that minimizes the estimated execution time of a query workload $\mathbb{Q}$. The partition manager physically partitions the table and generates indexes to locate tuples and attributes in the physical partitions. The query processor evaluates a query by accessing partitions the partition manager has already loaded. The description of the algorithms assumes the evaluation of the conjunction of predicates p_1, ..., p_n on table T. This query pattern is very common: 20 out of 22 queries in the TPC-H workload include a `WHERE` clause of the form p_1 `AND ... AND` p_n.

The *partition manager* stores a partition in one file. It constructs two indexes, a *tuple-level* index and an *attribute-level* index, to identify partitions storing a tuple or an attribute respectively. The query processor can access a partition by either specifying the ID of a tuple or an attribute; the partition manager will load the relevant partition in memory, if necessary.

The *query processor* evaluates queries on the irregularly partitioned table partition-by-partition. The query processor implements a *select* operation and a *project* operation. The select operation reads a partition, evaluates the relevant predicates and extracts the attributes projected by the query. The project operation then reconstructs tuples, possibly from multiple partitions.

4 PARTITIONING PROBLEM

4.1 Preliminary Definitions

Table. We represent a table T by its metadata: (i) $T.A$, the set of attributes in T; (ii) $T.t$, the number of tuples in T; (iii) $T.range$, the minimal and maximal values $[min_a, max_a]$ for each attribute $a \in T.A$. Storing this metadata avoids accessing the partition for tuple-level information.

Segment. A segment S contains a subset of tuples and attributes of the table. We use the same notation to represent a segment, i.e. the attributes in the segment are $S.A$, the number of tuples in the segment are $S.t$ and the value ranges of tuples in S are $S.range$. It is worth noting that $S.range$ contains the value ranges of all attributes in the table, including the attributes not in $S.A$. $S.Q$ is the set of training queries that access the segment. Algorithm 1 shows the data structure of a segment.

Partition. A partition is a set of segments of a table. Each segment is contained entirely in one partition.

Query. Algorithm 1 shows the metadata of a query q. A_σ is the set of attributes in the predicates of q. A_π is the set of attributes that are projected by q. (For example, query Q_1 in Table 2 has $A_\sigma = \{a_1\}$ and $A_\pi = \{a_2, a_3\}$.) The $range$ contains the $[min_a, max_a]$ for each attribute a in the table. If a is in A_σ, the min_a and max_a are the boundaries specified in predicates; otherwise min_a and max_a are the range of a in the table. (For example, $Q_1.range$ = {a_1:[11, 1000], a_2:[21, 26], a_3:[31, 36], a_4:[41, 4046], a_5:[51, 56], a_6:[61, 66]}.

Algorithm 1: Structure of Segment and Query

```
1 Struct Segment S contains
2     A // attributes in S
3     t // number of tuples in S
4     range ← {[min_a, max_a] | ∀ attribute a in the table}
5     Q // set of queries that access S
6 Struct Query q contains
7     A_σ // attributes in the WHERE clause
8     A_π // attributes in the SELECT clause
9     range ← {[min_a, max_a] | ∀ attribute a in the table}
```

Previous horizontal partitioning algorithms [8, 11, 39] formulate the partitioning problem at the tuple level. (For example, a node in the graph of Schism [11] is a tuple.) Instead, we use the value ranges to represent partitions and queries. The space consumption is proportional to the number of segments, which is significantly smaller than the tuple-level representation.

4.2 Problem Definition

Given a set of partitions $\mathbb{P} = \{P_1, \cdots, P_n\}$ and a set of queries $\mathbb{Q} = \{q_1, \cdots, q_m\}$ on a table T, we define cost function $cost(\mathbb{P}, \mathbb{Q})$ to estimate the I/O time to evaluate these queries on the given partitions. The partitioning problem is finding a valid partitioning $\mathbb{P}$ over T that minimizes the cost function. The query processor only accesses a partition once for each query, thus the I/O time of a query is the sum of the I/O time of partitions accessed by the query. The total cost is the sum of the I/O time for each query. Hence, the cost function that estimates the total I/O time is:

$$cost(\mathbb{P}, \mathbb{Q}) = \sum_{q \in \mathbb{Q}} \sum_{P \in \mathbb{P}} io(sizeof(P)) \times access(P, q) \tag{1}$$

The function $sizeof(P)$ returns the size of partition P in bytes:

$$sizeof(P) = \sum_{S \in P} S.t \times (B_{ID} + \sum_{a \in S.A} B_a) \tag{2}$$

where B_{ID} and B_a is the size of the tuple ID and attribute a.

The $io(x)$ function in Formula 1 estimates the I/O time $\hat{y}$ to read a partition that is x bytes big, and is a linear prediction of the form $\hat{y} = \alpha x + \beta$. Jigsaw derives the constants by profiling the file system: it measures the time (y) to read files of different sizes (x) and calculates the α and β parameters through linear regression. In general, we have observed that linear prediction is a reasonable estimation of the actual I/O time for both hard drives and SSDs as long as the partition sizes are at least a few MBs big to amortize the overhead of issuing one I/O request. Jigsaw uses a configurable lower limit on the partition size, referred to as *MIN_SIZE*, to ensure that the generated partitions are not too small.

The $access(P, q)$ function in Formula 1 returns 1 if the query q reads any segment in partition P or 0 if not. Query q accesses segment S if S contains any attributes in A_σ, or if S contains any attribute in A_π and $S.range_a$ intersects with $q.range_a$ for each attribute a. Hence:

$$access(P, q) = \begin{cases} 1, & \exists_{S \in P} access(S, q) = 1 \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

$$access(S, q) = \begin{cases} 1, & S.A \cap q.A_\sigma \neq \emptyset \\ 1, & S.A \cap q.A_\pi \neq \emptyset \\ & \bigwedge \forall_{a \in T.A} S.range_a \cap q.range_a \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

The partitioning problem is to find a valid partitioning $\mathbb{P}$ on a table T to minimize the $cost(\mathbb{P}, \mathbb{Q})$ for a given query set $\mathbb{Q}$. A valid partitioning must meet certain constraints. First, the segments must be partitions of T, that is each cell must belong to one segment. Mathematically, this means that any two segments in $\mathbb{P}$ do not intersect and that the union of all the segments is the table. The final constraint is that a segment only belongs to one partition of $\mathbb{P}$.

Irregular Partitioning Problem. *Given a set of queries $\mathbb{Q}$ on a table T, find a valid partitioning $\mathbb{P}$ on T that minimizes $cost(\mathbb{P}, \mathbb{Q})$:*

$$\begin{aligned} & \underset{\mathbb{P}}{\operatorname{argmin}}\ cost(\mathbb{P}, \mathbb{Q}) \\ \text{subject to} \quad & S_i.A \cap S_j.A = \emptyset \vee S_i.range \cap S_j.range = \emptyset \\ & \cup_{a \in S.A}\ S.range = T.range, \quad \forall a \in T.A \\ & S \notin P_j, \quad \forall S \in P_i \wedge i \neq j \end{aligned} \tag{4}$$

4.3 Partitioning Algorithm

4.3.1 Algorithmic framework. Enumerating all the possible partitions of a table is infeasible. Algorithm 2 shows a hill-climbing algorithm to find a partitioning that locally minimizes the cost function. The inputs of the algorithm are the table T and training query set $\mathbb{Q}$ on the table. The algorithm is composed of the *partitioning* phase (lines 1–12), *resizing* phase (lines 13–25) and *selection* phase (lines 26–27). The first two phases minimize the $cost(\cdot)$ function that was defined in Formula 1.

The partitioning phase runs a top-down algorithm to partition the table into segments. Initially, the first segment is the entire table which is marked as *active*. In each step, the algorithm partitions an *active* segment by the function $partitionSegment(\cdot)$ which returns the resulting segments after partitioning ($children$) and an estimation of the I/O savings from this split ($benefit$). (Segment partitioning is described in Section 4.3.2.) If the split is not projected to reduce I/O time further, the segment is *frozen*. The partitioning phase stops when finer partitioning will not reduce I/O time further ($benefit \leq 0$) and all segments are *frozen*.

The resizing phase splits or merges segments such that all partitions have sizes in a user-defined range $[MIN\_SIZE, MAX\_SIZE]$. The partitioning phase will split segments larger than MAX_SIZE to reduce redundant I/O when queries are encountered which do not look like the training queries. Conversely, segments that are smaller than MIN_SIZE will be merged accordingly to their access pattern similarity. Merging segments with a similar access pattern increases I/O performance by reducing I/Os to small partitions.

The final selection phase picks the more efficient layout between the irregular and the columnar layout based on the estimated tuple reconstruction cost. The irregular partitioning algorithm stores tuples that appear in multiple partitions in an in-memory hash table during tuple reconstruction (see Section 5.2). The algorithm checks whether the I/O cost of the chosen irregular partitioning

Algorithm 2: Partitioning algorithm

input : T, a table
$\mathbb{Q}$, training query set on T
output: $\mathbb{P}$, the partitions of T

1 $s_0 \leftarrow$ initialize a segment to be the entire table T
2 $s_0.Q \leftarrow \mathbb{Q}$
3 $active, frozen \leftarrow$ two empty lists to store segments
4 Add s_0 to the end of $active$
5 **repeat**
6 $\quad s \leftarrow$ Remove the first segment in $active$
7 $\quad children, benefit \leftarrow partitionSegment(s)$
8 $\quad$ **if** *benefit > 0* **then**
9 $\quad\quad$ Push $children$ to $active$
10 $\quad$ **else**
11 $\quad\quad$ Push s to $frozen$
12 **until** *active is empty*
13 **repeat**
14 $\quad s \leftarrow$ Remove the first segment in $frozen$
15 $\quad$ **if** $sizeof(s) > MAX\_SIZE$ **then**
16 $\quad\quad a \leftarrow$ the most frequent attribute in the predicates of queries in $s.Q$
17 $\quad\quad children \leftarrow horizontal\left(s, a, \frac{s.max_a + s.min_a}{2}\right)$
18 $\quad\quad$ Add $children$ to the end of $frozen$
19 $\quad$ **else if** $sizeof(s) < MIN\_SIZE$ **then**
20 $\quad\quad S \leftarrow \{s' | s' \in frozen$ **and** $s.Q = s'.Q\}$
21 $\quad\quad p \leftarrow$ merge s in S until $sizeof(p) \geq MIN\_SIZE$
22 $\quad\quad$ Add p to $\mathbb{P}$
23 $\quad$ **else**
24 $\quad\quad$ Create a partition for s and add to $\mathbb{P}$
25 **until** *frozen is empty*
26 **if** $cost(\mathbb{P}, \mathbb{Q}) + cost_{recons}(\mathbb{P}, \mathbb{Q}) > cost_{column}(\mathbb{Q})$ **then**
27 $\quad \mathbb{P} \leftarrow$ columnar layout
28 **return** $\mathbb{P}$

plan ($cost$) including the anticipated overhead of maintaining the in-memory hash table for irregular partitioning ($cost_{recons}$) is greater than the I/O cost of the simpler columnar layout ($cost_{column}$) that does not require maintaining a hash table. The total reconstruction cost $cost_{recons}$ for the query set $\mathbb{Q}$ with irregular partitioning is:

$$cost_{recons}(\mathbb{P}, \mathbb{Q}) = \sum_{q \in \mathbb{Q}} mem\left(\sum_{S \in \mathbb{P}} survived\_tuple\_num(S, q)\right) \tag{5}$$

where $survived\_tuple\_num(\cdot)$ is the number of tuples in a segment S that satisfy the query q, as estimated by `S.range` ∩ `q.range`. The $mem(x)$ function estimates the time to insert x tuples to the hash table based on the memory throughput (writes per second) of a microbenchmark that writes to random memory locations. The function $cost_{column}$ estimates the I/O cost of the columnar layout as follows:

$$cost_{column}(\mathbb{Q}) = \sum_{q \in \mathbb{Q}} \sum_{a \in q.A_\sigma \cup q.A_\pi} io(page\ size) \times \frac{sizeof(a)}{page\ size} \tag{6}$$

Algorithm 3: *partitionSegment()*

input : S, a segment
output: *children*, the children segments of S
benefit, the saved I/O time after partitioning

1 $C_{initial} \leftarrow cost(S, S.Q)$
2 **foreach** $q \in S.Q$ **do**
3 $S_\sigma \leftarrow Segment(S.A \cap q.A_\sigma, S.t, S.range)$
4 $S_\pi \leftarrow Segment\big((S.A \cap q.A_\pi) - q.A_\sigma, S.t, S.range\big)$
5 $S_r \leftarrow Segment(S.A - S_\sigma.A - S_\pi.A, S.t, S.range)$
6 **foreach** $a \in q.A_\sigma$ **and** $v \in \{q.min_a, q.max_a\}$ **do**
7 $S_{\pi 1}, S_{\pi 2} \leftarrow horizontal(S_\pi, a, v)$
8 **foreach** $q' \in S.Q$ **do**
9 **foreach** $s' \in \{S_\sigma, S_{\pi 1}, S_{\pi 2}, S_r\}$ **do**
10 **if** $access(s', q')$ **then**
11 $s'.Q.add(q')$
12 $children_{q,a,v} \leftarrow \{S_\sigma, S_{\pi 1}, S_{\pi 2}, S_r\}$
13 $children \leftarrow \text{argmin}_{children_{q,a,v}}\ cost(children_{q,a,v}, S.Q)$
14 **return** $children, C_{initial} - cost(children, S.Q)$

4.3.2 Segment partitioning algorithm. Algorithm 3 defines the function $partitionSegment(\cdot)$ which horizontally and vertically partitions a segment S. It outputs the resulting segments and the expected I/O time benefit after partitioning (which may be negative if evaluating the queries on the resulting segments is estimated to take longer). Given a query $q \in S.Q$, the algorithm first vertically partitions S into three segments, S_σ, S_π, and S_r (lines 3–5). S_σ stores the attributes of S which are in predicates of q, S_π stores the attributes projected by q but not in S_σ, and S_r stores the remaining attributes of S. A query q reads all tuples in S_σ, some tuples in S_π and nothing in S_r. The segment S_π is further partitioned horizontally into two segments (lines 6–7) on the boundaries of each attribute. Algorithm 3 updates the queries that access each segment according to Formula 3.2 (lines 8–11). Finally, line 13 returns the partitioning with the minimal I/O cost.

Algorithm 4 describes the $horizontal(\cdot)$ function. The function horizontally partitions a segment on an attribute a by setting the min and max values of a in the children segments. The function estimates the size of children segments, assuming that the distribution of each attribute is uniform and independent. Other cardinality estimation techniques can be used for more accurate results.

Algorithm 4: *horizontal()*

input : S, a segment
a, an attribute
v, the partitioning point on a
output: S_1, S_2, two children segments of S

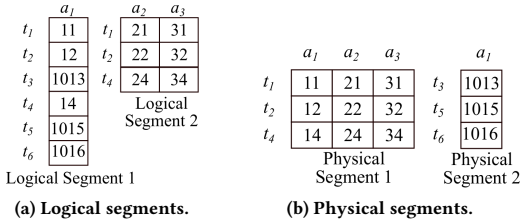
1 $t_1 \leftarrow S.t \times \frac{v - S.min_a}{S.max_a - S.min_a}$
2 $S_1 \leftarrow Segment(S.A, t_1, S.range)$
3 $S_2 \leftarrow Segment(S.A, S.t - t_1, S.range)$
4 $S_1.max_a, S_2.min_a \leftarrow v$
5 **return** S_1, S_2

5 SYSTEM PROTOTYPE

The *Partition Manager* physically reorganizes the table according to the logical partitioning generated by the *Partition Tuner*. The *Query Processor* evaluates queries on the partitions. Section 5.1 describes the physical storage organization in *Partition Manager* and Section 5.2 describes the query evaluation algorithm.

5.1 Partition Manager

The partition manager stores each partition in a separate file. A partition is composed of multiple *physical segments*. A physical segment stores tuples with the same attributes in this partition. Figure 3a shows the two *logical segments*, as generated by the partitioning algorithm, for the upper left partition in Figure 1e. Figure 3b shows the physical storage for the same partition. Note that tuples t_1, t_2, t_4 are stored sequentially in the same physical segment because they contain the same attributes.



(a) Logical segments. **(b) Physical segments.**

Figure 3: Logical and physical representation of a partition.

Cells in a physical segment are serialized in row-major order. A physical segment also stores the tuple IDs and a bitmap to record the attributes in the segment. (A tuple ID is stored once in a partition.) Figure 4 shows the disk layout of the partition.

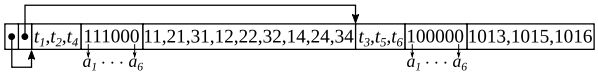


Figure 4: Physical layout of a partition on disk.

Finally, the partition manager builds two indexes, the *attribute-level* index and *tuple-level* index to find the partitions storing an attribute or a tuple. The indexes are hash tables, where the keys are the attribute name and tuple IDs, respectively. Given an attribute name and a tuple ID, the partition manager consults the indexes and returns the partition storing the specific cell.

5.2 Query Processor

As discussed in Section 2.2, the *tuple-at-a-time* and *operator-at-a-time* methods may read a partition repeatedly when the table is irregularly partitioned due to the different tuple orders in different partitions. Jigsaw proposes the *partition-at-a-time* method (Algorithm 5) to process a query. Jigsaw exhausts a partition before moving to the next one to avoid repeated accesses.

The method consists of the *selection* phase (lines 3–16) and the *projection* phase (lines 17–23). The inputs of the algorithm are a query and the indexes described in Section 5.1, and the output is a hash table storing the query result. The hash table stores the projected attribute values indexed by their tuple ID.

Algorithm 5: Partition-at-a-time query evaluation

```
   input : A_proj, the projected attributes in the SELECT clause
           Preds, the query predicates
           Index_t, the tuple-level index
           Index_a, the attribute-level index
   output: ret, a hash table storing results
1  ret ← an empty hash table
2  status ← set NOT_CHECKED for each tuple in the table
3  A_pred ← the attributes in Preds
4  P_pred ← retrieve partitions containing A_pred from Index_a
5  foreach partition p ∈ P_pred do
6      foreach tuple t ∈ p such that status[t] ≠ INVALID do
7          A_t ← the attributes of t stored in p
8          if A_t cells in t do not satisfy Preds then
9              if status[t] = VALID then
10                 delete ret[t]
11             status[t] ← INVALID
12         else
13             if status[t] = NOT_CHECKED then
14                 ret[t] ← allocate one row for t
15                 status[t] ← VALID
16             add A_t ∩ A_proj cells of t to ret[t]
17 foreach tuple t ∈ ret do
18     A_miss ← A_proj attributes missing in ret[t]
19     P_proj ← P_proj ∪ partitions containing A_miss of t
20 foreach partition p ∈ P_proj do
21     foreach tuple t ∈ p such that status[t] = VALID do
22         A_t ← the attributes of t stored in p
23         add A_t ∩ A_proj cells of t to ret[t]
24 return ret
```

The selection phase scans the partitions that contain attributes that match the predicates of the query. During processing, a tuple is either NOT_CHECKED, VALID or INVALID. The status is initialized as NOT_CHECKED (line 2). The selection phase starts by reading a partition that contains an attribute that appears in the query predicates. Each tuple is evaluated against the predicate and becomes VALID if it satisfies the predicates (lines 13–15). Any projected attributes of VALID tuples that are stored in this partition are added to the hash table (line 16) to avoid accessing this partition again in the projection phase that follows. A tuple becomes INVALID if it does not satisfy a predicate, in which case it is removed from the hash table (lines 8–11). The algorithm then proceeds to the next partition in P_{pred}. At the end of the selection phase, the hash table stores the projected attributes that are stored in the partitions that were accessed when evaluating the WHERE clause of the query, indexed by their tuple IDs.

The projection phase scans the hash table to fill missing attributes from the selection phase. The algorithm first identifies the partitions containing missing cells (lines 18–19) and loads them using the tuple-level index $Index_t$. (Note that A_{miss} may be the empty set, in which case P_{Proj} will not grow.) For each partition, the algorithm iterates over the VALID tuples of the partition and fills the missing cells in the hash table (lines 20–23).

Algorithm 6: Parallelization based on locking

```
1  Phase Selection
2      foreach thread th do in parallel
3          repeat
4              p ← pop P_pred and load the partition
5              foreach tuple t ∈ p do
6                  lock the bucket with bucket ID hash(t)
7                  process t // lines 6–16 in Algorithm 5
8                  unlock the bucket
9          until P_pred = ∅
10     wait for all threads to complete
```

5.2.1 Parallelization. There are two ways to parallelize Algorithm 5, one based on locking and another based on shared scans [22]. Threads in the lock-based strategy process different partitions. The shared scan strategy uses all threads to process one partition but assigns a part of the hash table to each thread using range partitioning. The two strategies differ in how they resolve conflicts across multiple threads.

The lock-based strategy assigns locks to hash table buckets so that two threads will not concurrently access the same bucket. Algorithm 6 describes the selection phase in the lock-based strategy. A thread reads a partition in P_{pred} and iterates over the tuples in the partition. The thread locks the target bucket when it processes each tuple. All threads wait at the end of the selection phase in a barrier and then proceed to the projection phase. The projection phase does not need hash table locks, because threads do not update the *status* array and only fill missing cells of tuples that have already been allocated in the hash table (see Alg. 5, line 14).

The strategy based on shared scans is a lock-free algorithm, where a thread scans all the partitions and only processes tuples for the buckets this thread is responsible for. Algorithm 7 describes the selection and projection phases in this method. In the selection phase, threads load partitions in P_{pred} and wait for loading to complete. Each thread then iterates over all the loaded partitions and processes tuples which should be stored in the buckets assigned

Algorithm 7: Parallelization based on shared scans

```
1  Phase Selection
2      foreach thread th do in parallel
3          B_th ← the buckets assigned to th
4          p_th ← pop P_pred and load the partition
5          wait for all threads to complete
6          foreach thread i do
7              foreach tuple t ∈ p_i do
8                  if hash(t) ∈ B_th then
9                      process t // lines 6–16 in Algorithm 5
10 Phase Projection
11     foreach thread th do in parallel
12         foreach partition p ∈ P_proj do
13             foreach tuple t ∈ p do
14                 if hash(t) ∈ B_th and status[t] = VALID then
15                     process t // lines 22–23 in Algorithm 5
```

to this thread (line 7). The selection phase stops when all partitions in P_{pred} have been scanned by all threads. In the projection phase, each thread scans all the partitions in P_{proj} and processes the assigned tuples (lines 12–13). In this strategy, threads do not conflict because they process distinct bucket ranges in the hash table.

6 EXPERIMENTAL EVALUATION

This section evaluates the partitioning algorithm and the query processor in Jigsaw. We perform all experiments on three servers with different compute and I/O capabilities: an on-premises server named **balos**, and the **t2.2xlarge** and **c5.9xlarge** instances on Amazon AWS. The on-premises server stores data in a locally attached HDD; the t2.2xlarge and c5.9xlarge instances store data on two EBS SSD volumes, **gp2** and **io1**, respectively. (The io1 volume is the highest-performing EBS volume and is recommended for database workloads.) The typical throughput is 75 MB/s for the local HDD, 125 MB/s for the gp2 volume and 1 GB/s for the io1 volume. Table 3 shows the configuration of the three servers. The experiments use all the CPU cores on each server. Because irregular partitioning is an I/O optimization that is designed for cold storage, we clear the OS cache before evaluating each query to force the query processor to read all data from the underlying storage device. We repeat each experiment at least three times and report the median response time. The evaluation considers the following questions:

- **Parallelization performance:** When and why is shared scan-based parallelization faster than lock-based parallelization, and vice versa?
- **Query evaluation performance:** What is the relative performance of Jigsaw compared with the state-of-art baselines? Does irregular partitioning read less redundant data?
- **Partitioning time:** How does the partitioning algorithm in Jigsaw compare with the existing partitioning algorithms in terms of the partitioning time?

6.1 Benchmarks and Baselines

6.1.1 Benchmarks. We use two benchmarks: HAP and TPC-H.

HAP: The benchmark has two tables, a narrow table with 16 columns and a wide table with 160 columns. Our experiments use the wide table with 100M tuples. Each attribute is a 4-byte uniformly distributed integer. We use the read-only queries in the HAP benchmark for evaluation, which are of the form `SELECT` a_i, ... , a_j, ... , a_k `FROM` T `WHERE` $C_1 < a_j < C_2$. The query workload is configured by 4 parameters: selectivity, projectivity, the number of query templates and the number of queries.

We create a workload by first generating query templates. Given a desired projectivity, a query template projects a randomly selected set of attributes and uses one of the projected attributes, a_j, in predicates. We then randomly pick a query template to populate a query. The constants C_1 and C_2 are set randomly but meet the selectivity requirement. Random queries are not a realistic benchmark but represent a difficult case for Jigsaw because the workload does not have frequently co-accessed columns or tuples, hence it is more challenging to find an optimal partitioning across all queries. Previous work, such as Mosaic [42], follows the same procedure to create and run a query workload for evaluation purposes.

Name	**balos**	**t2.2xlarge**	**c5.9xlarge**
CPU	Xeon E-2246G (6 CPU cores)	Xeon E5-2676v3 (8 vCPUs)	Xeon P-8124M (36 vCPUs)
Memory	62GB	32GB	72GB
Storage	HDD (75 MB/s)	gp2 (125 MB/s)	io1 (1 GB/s)

Table 3: Experimental environment.

TPC-H: We generate a TPC-H database with a scale factor of 30. We adopt the evaluation strategy of GSOP [40]: we denormalize the database and evaluate 5 TPC-H query templates, Q_3, Q_6, Q_8, Q_{10} and Q_{14}, on the denormalized LINEITEM table. (In total, we materialize 19 attributes in the denormalized table.) We generate 500 training queries and 10 evaluation queries at random, equally distributed among the 5 query templates.

6.1.2 Baselines. We compare Jigsaw with six partitioning baselines: Row, Row-H, Row-V, Column, Column-H, and Hierarchical.

- **Row**. The Row baseline stores the table in row order, tuple by tuple, in multiple file segments. The Row baseline stores tuples in their natural order.
- **Row-H**. The Row-H baseline horizontally partitions a table by the graph partitioning algorithm in Schism [11]. A tuple is a node in the graph; two nodes are connected if the two tuples are accessed by the same query. We sample 160K tuples to build the graph and assign the remaining tuples to the partitions. The algorithmic complexity is $O(N^2 \cdot Q)$ where N is the number of tuples in the table and Q is the number of training queries. Figure 1b shows a possible partitioning by the Row-H baseline. After partitioning, Row-H serializes tuples in row order. Horizontal partitions in Row-H are sized to fill an entire file segment.
- **Row-V**. The Row-V baseline uses the greedy algorithm in Peloton [7] to partition columns. The algorithm sorts the query templates by descending order of evaluation time, iterates over the templates, and groups columns in each template as one vertical partition. The algorithmic complexity is $O(Q \cdot A)$ where A is the number of attributes. Figure 1c shows a possible partitioning by the Row-V baseline. Within a vertical partition, Row-V stores a partition tuple by tuple in the natural order the tuples appear in the table. A partition in Row-V spans multiple file segments.
- **Column**. The Column baseline serializes a table in column order, attribute by attribute. Within each column, tuples are stored in their natural order. A column spans multiple file segments.
- **Column-H**. The Column-H baseline horizontally partitions tuples by the same algorithm as Row-H. Columns in the resulting partitions are then stored as separate file segments. The partitioning in Column-H is coarser than in Row-H: horizontal partitions are sized such that each column fills an entire file segment.
- **Hierarchical**. The Hierarchical baseline horizontally partitions the table by the partitioning algorithm in Row-H and then independently splits each horizontal partition vertically by the partitioning algorithm in Row-V, as shown in Figure 1d. In the Hierarchical baseline, each partition becomes a file segment.

We do not consider the Column-V variant, which first vertically partitions a table and then stores each partition in columnar order, because it would have the same disk layout as Column. For all baselines, a file segment is at least 4MB big and is accessed in its entirety. For irregular partitioning, the `MIN_SIZE` and `MAX_SIZE` in

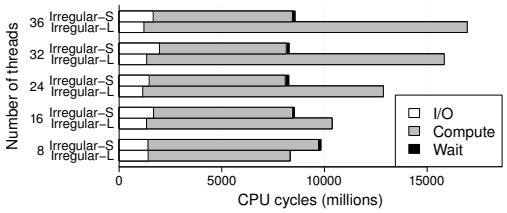


Figure 5: Breakdown of CPU cycles with shared scan-based and lock-based parallelization.

Algorithm 2 are set to 4MB and 32MB respectively. Therefore, the file segments that are irregular partitioning reads are no smaller than the file segments of the other partitioning strategies. We implement all partitioning strategies in a multi-threaded data storage and query processing engine that is written in C++ and is based on an earlier prototype [25, 26].

6.2 Microbenchmarking

In this section, we first compare the *lock-based* and *shared scan-based* parallelization methods. Then we report the query evaluation performance and how much data is read from disk with Jigsaw and the baselines on the HAP benchmark when varying query workload parameters, specifically selectivity, projectivity and the number of query templates.

6.2.1 Parallelization strategies. We compare the *lock-based* parallelization (Irregular-L) and *shared scan-based* parallelization (Irregular-S) by running a HAP query on the c5.9xlarge node. We vary the number of threads from 8 to 36. Figure 5 shows the breakdown of CPU cycles in the select operator for the two parallelization strategies. The CPU cycles are decomposed to I/O, computation and waiting cycles, and are averaged over the active threads.

Looking at the computation cycles, Irregular-L is faster than Irregular-S when there are 8 threads. This is because a thread in Irregular-S has to process all partitions while a thread in Irregular-L only processes a subset of the partitions. The second reason is that Irregular-S threads concurrently process the partitions after the barrier, while threads in Irregular-L process partitions independently. With more threads, Irregular-S becomes faster but Irregular-L becomes slower. An Irregular-L thread may access any bucket in the hash table to process a partition, leading to false sharing. Irregular-S does not have the false sharing problem because threads are responsible for different bucket ranges in the hash table.

Irregular-S spends more cycles doing I/O when we increase the number of threads because more threads concurrently read from the disk. The I/O cycles do not change significantly for Irregular-L because the Irregular-L threads read partitions independently and few threads concurrently read from the disk. In the following experiments, we use Irregular-S to represent irregular partitioning due to its good and stable performance with many threads.

6.2.2 Selectivity. In this experiment, the query workload uses 2 query templates, each of which projects 16 out of the 160 attributes. We tune the selectivity of the query workload from 1% (1M selected tuples) to 100% (100M selected tuples). We do not consider queries that are more selective because a query would likely access the data through an index, which our current prototype does not support.

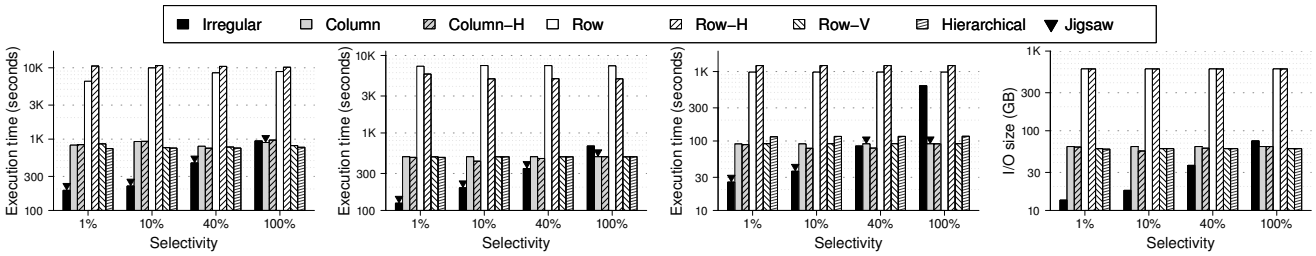
Figure 6 shows the result when we vary the query selectivity for each server. The Jigsaw mark (a triangle) shows the layout that Algorithm 2 picks (between Irregular or Column) after considering the tuple reconstruction cost. Figure 6d shows how much data was accessed to complete the query (same for all servers).

Evaluating queries on irregular partitions is 4.2× faster than Column when the selectivity is low. Row and Row-H are the slowest because they store all attributes together so that they have to scan the entire table for each query to evaluate predicates. As the selectivity increases, the performance gap between Irregular and the baselines shrinks, because the data Irregular partitioning accesses increases from 13.5GB to 74.5GB. (Recall that in addition to the data, Irregular partitioning needs to read the tuple IDs that are stored with the projected attributes.) When the selectivity is 100%, Irregular has the same performance as Column on t2.2xlarge and balos because of the similar I/O volume. The Row-V and Hierarchical baselines read the same amount of data because the vertical partitioning algorithm stores the attributes of the `SELECT` and the `WHERE` clauses of the query in the same partition. Thus, Hierarchical reads redundant attributes when it evaluates predicates.

When the selectivity is 40%, Irregular is as fast as the baselines on c5.9xlarge but 1.7× faster than baselines on balos. Irregular has better speedup on the slower storage of balos and t2.2xlarge than the faster storage of c5.9xlarge because Irregular transfers less data but spends more time to reconstruct tuples in memory.

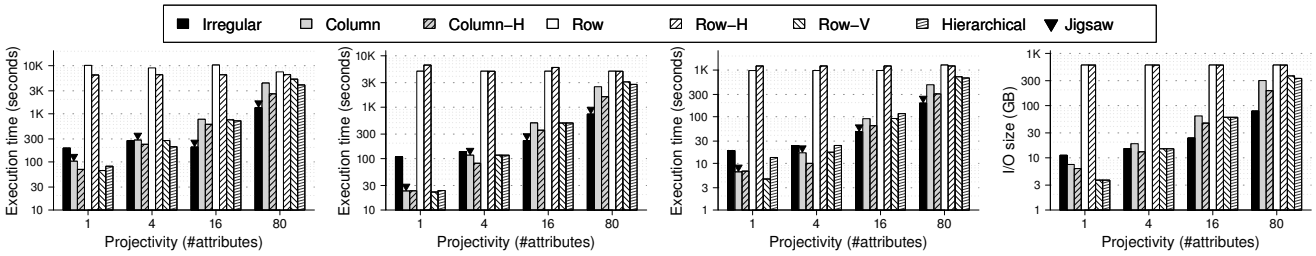
6.2.3 Projectivity. In this experiment, the workload consists of 2 query templates with a selectivity of 20%. The number of attributes each query projects varies from 1 to 80 out of the 160 attributes. Figure 7 shows the results. The execution time of Irregular increases from 184 seconds to 909 seconds on balos, while the time of Column increases from 103 seconds to 4337 seconds. Irregular reads 1.5× more data than Column when the query projects 1 attribute but 74% less data when the query projects 80 attributes. The I/O overhead in the baselines is that they either read redundant tuples because they do not horizontally partition the table (Column and Row-V), or the horizontal partitioning is not optimal (Column-H and Hierarchical). This I/O overhead increases as the query reads more attributes. Similar to the selectivity experiment, Irregular has better speedup on balos and t2.2xlarge than on c5.9xlarge.

6.2.4 Number of query templates. This experiment fixes the selectivity to 20% and projectivity to 16 out of the 160 attributes but varies the number of query templates in the query workload from 2 to 8. As the workload contains more templates that randomly pick what to read, the attributes are more finely fragmented into partitions. Figure 8 shows the results as we increase the number of query templates. The performance gap between Irregular and Column on t2.2xlarge shrinks from 257 seconds to 126 seconds, while the I/O volume for Irregular increases from 24.2GB to 40.5GB. This is because the table is vertically partitioned more finely and tuple IDs are replicated when Irregular vertically splits a partition, so Irregular reads more tuple IDs overall. Row-V reads more redundant attributes when there are more query templates because the greedy algorithm in Row-V scatters attributes in low-cost query templates into the vertical partitions generated by high-cost query templates. The speedup of Column-H over Column decreases from 1.5 to 1 on c5.9xlarge as templates filter tuples by different attributes.



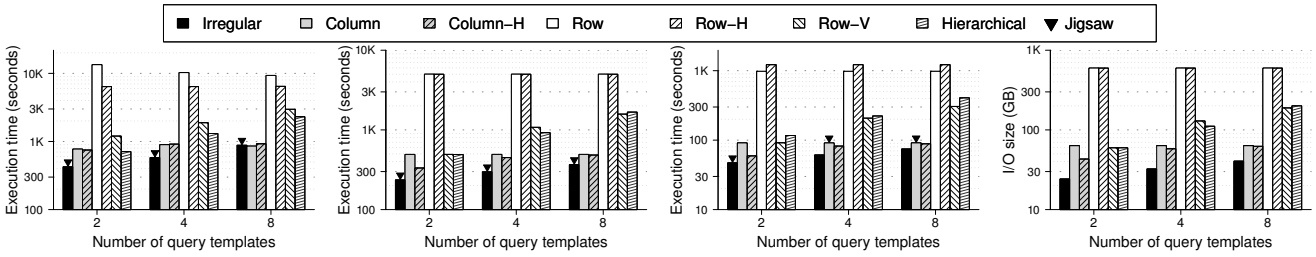
(a) balos (b) t2.2xlarge (c) c5.9xlarge (d) I/O size

Figure 6: Vary the selectivity from 1% to 100%. Irregular is at most 4.2× faster than Column and reads 79% less data.



(a) balos (b) t2.2xlarge (c) c5.9xlarge (d) I/O size

Figure 7: Vary the number of projected attributes. Irregular is at most 3.4× faster than Column and reads 74% less data.



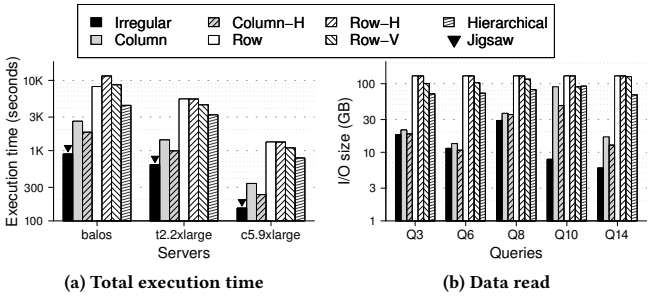
(a) balos (b) t2.2xlarge (c) c5.9xlarge (d) I/O size

Figure 8: Vary the number of query templates. Irregular is at most 2.1× faster than Column and reads 62% less data.

6.3 End-to-end Performance

6.3.1 TPC-H. This experiment evaluates Jigsaw with TPC-H queries that more realistically mimic real applications because they incorporate business logic instead of returning random ranges of tuples as in HAP. (For example, TPC-H Q_8 selects tuples with `O_ORDERDATE` between [1995-01-01, 1996-12-31].)

Figure 9 shows the total execution time and the data transferred from the disk with the TPC-H query workload. Irregular is 2× faster than the fastest baseline Column-H on balos, and at least 86% of the execution time is I/O. Irregular and Column-H transfer 72.5GB and 125GB data, respectively, whereas the absolutely necessary data for query evaluation is 43.8GB for this experiment. Column-H horizontally partitions attributes the same way, regardless of the attribute access pattern, and produces rectangular partitions with an average file segment size of 10MB. Inadvertently, rectangular partitions require reading some unnecessary data. Irregular transfers less data because its partitions are neither horizontally nor vertically aligned: for this workload, Irregular generates 80 irregular partitions, each of which stores multiple segments of the table (the average file segment size is 22MB). The storage overhead of Irregular is 28.7GB, most of which (27.4GB) is reading tuple IDs.



(a) Total execution time (b) Data read

Figure 9: TPC-H. Irregular is 2.9× faster than Column.

Breaking I/O volume by query, the data read by Irregular and Column-H is similar for Q_3, but Irregular reads 84% less data for Q_{10}. This is because Q_3 filters by more attributes but projects less data from each tuple than Q_{10}. Specifically, Q_3 filters by 3 attributes, which Irregular stores in different partitions, and projects 36 bytes (5 attributes). Q_{10} filters by 2 attributes, which Irregular stores together, and projects 254 bytes (9 attributes). As a result, Irregular reads fewer tuple IDs for Q_{10} than for Q_3, and this overhead is amortized over more projected attributes. Hierarchical reads more data than Column-H because attributes that are stored together are not always co-accessed. For example, some partitions in Hierarchical store `C_COMMENT` and `L_EXTENDEDPRICE` together but 4 out of the 5 query templates only access `L_EXTENDEDPRICE`.

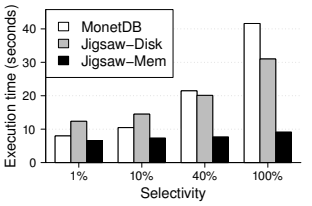


Figure 10: Comparison with in-memory data.

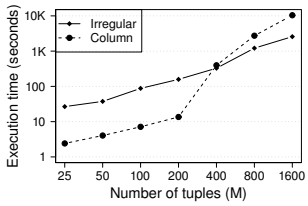


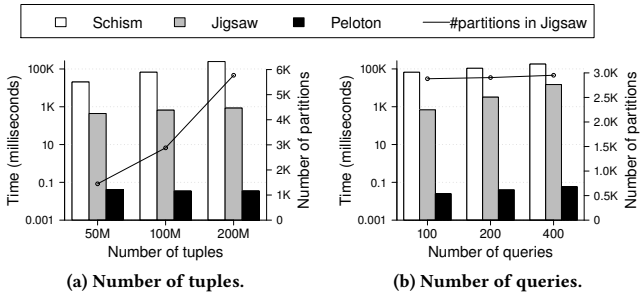
Figure 11: Varying the cardinality of the table.

6.3.2 Evaluation with in-memory data. This experiment compares Jigsaw and MonetDB, an open-source columnar database, when the database fits in memory. The experiment uses the HAP table and the arithmetic query `SELECT max(` $a_i + ... + a_j + ... + a_k$ `) FROM` T `WHERE` $C_1 < a_j < C2$. (Returning the maximum minimizes the communication overhead between the server and the client of MonetDB.) Figure 10 reports the execution time as we vary the selectivity. Algorithm 2 picks the columnar layout, denoted as Jigsaw-Mem, while Jigsaw-Disk denotes the irregular partitioning.

When the selectivity is 1%, Jigsaw-Disk is slower than Jigsaw-Mem and MonetDB, primarily because Jigsaw-Disk requires significant random memory accesses in the hash table to insert and update cells for tuple reconstruction. However, MonetDB becomes the slowest engine as selectivity increases: Because MonetDB evaluates arithmetic operators attribute by attribute, it materializes intermediate columns. (Adding attributes spends 94% of the execution time when the query selectivity is 100%.) Jigsaw-Mem does not materialize intermediate columns because it reconstructs tuples before evaluating arithmetic. Jigsaw-Disk stores a partition in row-major order and reconstructs tuples in the hash table before evaluating arithmetic. This result supports the design decision to store data in row-major order inside irregular partitions in Jigsaw.

6.3.3 Impact of database size. An important question is what is the performance of irregular partitioning when the data is not cold. This experiment does not flush the OS cache and excludes the execution time of the first query for each template, hence the reported results reflect the performance one can expect if queries access warm data. This experiment uses HAP with 2 query templates, fixes selectivity to 10% and projects 16 attributes out of 160. The experiment runs on balos that has 62GB of usable memory. Multiple tables are generated for this experiment with different cardinalities, ranging from 25M tuples (16GB of data) to 1,600M tuples (1TB of data). A cardinality of 100M tuples means that the table no longer fits in memory; a cardinality of 400M tuples means that the columns the query workload accesses no longer fit in memory.

Figure 11 plots the execution time of Irregular partitioning and the Column baseline. Column is about 11× faster than Irregular for the smallest table (25M tuples). When the table has up to 200M tuples, the execution time of Irregular and Column grows proportionally to the number of tuples. When the table has 200M tuples, Column reads 25GB of data in total, so the accessed columns comfortably fit in memory where the file segments have been cached by the OS. Although Irregular reads less data, the reconstruction cost dominates the execution time for small tables. When the table has 400M tuples, the data transferred by the query workload for the Column baseline reaches the memory capacity of balos. Accessing cold partitions then starts to dominate the execution time, and Irregular becomes faster as it reads less data: when the table has 1.6 billion tuples, Irregular partitioning is 3.5× faster than the Column baseline.



(a) Number of tuples. **(b) Number of queries.**

Figure 12: Partitioning time for different cardinalities and number of queries. Jigsaw is up to 290× faster than the graph-based partitioning in Schism.

6.4 Partitioning performance

This section evaluates the partitioning performance of different algorithms. Assume that a table has N tuples, A attributes and the workload has Q queries. The Row-V baseline uses the greedy algorithm in Peloton [7] to partition attributes that has a time complexity of $O(Q \cdot A)$. Column-H uses the graph partitioning algorithm in Schism [11] to partition tuples. The time complexity of the Schism algorithm is $O(N^2 \cdot Q)$.

The experiments in this section use the HAP table and vary the number of tuples and the number of queries in the workload. The query selectivity is 20%, projectivity is 16 attributes (out of 160) and 2 query templates are used. Figure 12 reports the partitioning time for Jigsaw, and our implementations of the Schism (Row-H and Column-H) and Peloton (Row-V) partitioning algorithms. The figure also shows the number of partitions generated by Jigsaw. The partitioning time does not include the time to load the data or write the partitions, as the performance of these phases is determined by the I/O bandwidth and not the partitioning algorithm.

6.4.1 Sensitivity to cardinality. Figure 12a shows the performance as we increase the number of tuples in the table. Jigsaw partitions the table up to 290× faster than Schism while Peloton is 25K× faster than Jigsaw. Jigsaw and Peloton are faster than Schism because they are tuple-agnostic: While Schism partitions the table tuple by tuple, Peloton only partitions vertically and Jigsaw horizontally partitions based on the value ranges of the attributes.

The performance gap between Jigsaw and Schism increases quadratically with the number of tuples. The partitioning time of Jigsaw increases linearly as we double the number of partitions because Jigsaw partitions the space, whereas the graph-centric partitioning in Schism takes almost 4× as the number of tuples doubles.

Jigsaw maps tuples to logical partitions and then materializes partitions on the disk. Jigsaw writes more data than the baselines, as it needs to store the tuple IDs and the index. For the 100M table, Jigsaw spends 78 seconds to write data while Column-H spends 90 seconds because Jigsaw generates fewer, larger partitions.

6.4.2 Sensitivity to the number of queries. Figure 12b shows the performance as we increase the number of queries. Jigsaw partitions the table 97× faster than Schism when there are 100 queries and 12× faster with 400 queries. The partitioning time of Schism is linearly proportional to the number of queries, while the partitioning time of Jigsaw increases by 4.6× as we double the number of queries. This is because Jigsaw generates a partitioning candidate for each query and then compares all candidates. As a result, the Jigsaw partitioning time is quadratic to the number of queries.

7 RELATED WORK

Storage Models. Row- and column-major storage models optimize for different workloads. The row-major storage model, the *de facto* standard for many commercial and open-source databases, serializes a table tuple by tuple. The column-major storage model, in systems such as C-Store [37], MonetDB [9] and DB2 BLU [34], serializes a table column by column. Prior work [1, 15, 16, 37] compares the two storage models on various workloads. Prior work has also explored ways to benefit from both storage options in one DBMS. Fractured mirrors [33] store replicas of a table in different models. Peloton [7] uses the hybrid storage model that partitions a table into rectangular tiles. Each tile contains a subset of tuples and attributes and can be stored in any storage layout.

Partitioning. Partitioning is a technique to physically split a table or a database. It helps maximize intra-query parallelism for OLAP workloads [30, 35] and minimize the number of distributed transactions for OLTP workloads [11, 29, 31]. The SQL Server Autoadmin [3, 10, 29] and DB2 Database Advisor [35, 47] are two industrial tools to partition databases. Prior work horizontally partitions tuples, vertically partitions attributes, or partitions both.

The range and hash partitioning [35, 43, 44] partitions tuples by applying range or hash functions on a specific attribute. Yang et al. [43] use reinforcement learning to construct a tree, where each node is a "cut" on an attribute. Other work [11, 39] partitions tuples based on the access pattern. Schism [11] models the access pattern in a graph. A node in the graph is a tuple and two nodes are connected if the corresponding two tuples are accessed in the same query. Sun et al. [39] have also explored how to recursively merge tuples with similar access patterns.

Vertical partitioning splits attributes to avoid reading redundant attributes. Both bottom-up and top-down methods have been employed to automatically partition attributes. The bottom-up method [5, 13, 14, 21, 30] recursively merges small partitions with similar access patterns into larger partitions. The top-down method [28] recursively breaks large partitions into smaller partitions, starting from a complete table. Jindal et al. [20] compare the performance of vertical partitioning methods for a column store.

Partitioning a table in one dimension is suboptimal when a query only reads a subset of tuples and attributes. Prior work has proposed to partition on both dimensions in separate phases. Peloton [7] and Autopart [30] first horizontally partition tuples and then vertically partition attributes independently for each horizontal partition. However, this two-phase partitioning approach inherits the limitations of horizontal or vertical partitioning, albeit at a smaller scale. In contrast, Jigsaw generates irregular partitions by partitioning both dimensions simultaneously.

Comparing with materialized views. The partitioning algorithm in Jigsaw has intellectual similarities with materialized views, as Jigsaw aims to partition the input table such that it can be directly used to answer a query. One difference is that materialized views can encode additional computation, such as aggregation, whereas Jigsaw only aims to optimize the storage layout. This means that Jigsaw is limited to identifying common data access patterns, while materialized views target common sub-trees in query plans [19, 36]. Another key difference is that materialized views can replicate cells that are frequently accessed by several queries, whereas Jigsaw will store each cell in one partition. Neither decision is optimal: Jigsaw incurs a reconstruction cost that could have been avoided through replication, but replicating cells requires more disk space and complicates updates. Prior work in the materialized view literature [2, 4, 45] has proposed merging materialized views in a similar manner as the merge algorithm in Jigsaw.

Query processing. The *tuple-at-a-time* and *operator-at-a-time* are two query evaluation methods for row stores and column stores respectively. MySQL and PostgreSQL [38] read the needed attributes tuple by tuple [12] when they evaluate queries. In order to amortize the overhead of a function call per tuple, the block-iterator model [46] extends the tuple-at-a-time method by returning blocks of tuples to the upstream operators. The Row, Row-H, and Row-V baselines in the experiments use the block-iterator model. In the operator-at-a-time method, an operator must complete before its subsequent operators start [18]. The Column and Column-H baselines adopt the operator-at-a-time method.

8 CONCLUSION

The partitioning strategies in prior work produce rectangular partitions. We present Jigsaw, a prototype system that allows partitions with arbitrary shapes. Jigsaw recursively splits a table until finer partitioning will not transfer less data for a given query workload and then merges small segments into larger, irregular partitions. Jigsaw introduces a partition-at-a-time query processing strategy to avoid accessing partitions multiple times during query evaluation due to their irregular shape. Experimental results show that Jigsaw speeds up query processing by up to 4.2× compared to columnar partitioning for moderately selective queries. The speedup is because irregular partitioning transfers less data from cold storage: columnar partitioning reads 4.7× more data than irregular partitioning to answer the same query.

Jigsaw currently only considers how to split cells into different partitions. Allowing for limited replication of certain cells could reduce the tuple reconstruction cost when accessing multiple partitions. In addition, the partitioning algorithm in Jigsaw is currently single-threaded. Parallelizing the compute-intensive partitioning phase has the potential to significantly accelerate the algorithm. Furthermore, Jigsaw is currently limited to full scans and optimizes accesses independently for each table. Investigating how irregular partitioning can benefit from indexing and how queries with complex joins can better leverage irregular partitioning are promising avenues for future work.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation (grant CCF-1816577) and Oracle America, Inc.

REFERENCES

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 967–980. https://doi.org/10.1145/1376616.1376712

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 496–505.

[3] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 359–370. https://doi.org/10.1145/1007568.1007609

[4] Rafi Ahmed, Randall Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated Generation of Materialized Views in Oracle. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3046–3058. https://doi.org/10.14778/3415478.3415533

[5] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H_2O: A Hands-free Adaptive Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1103–1114. https://doi.org/10.1145/2588555.2610502

[6] Apache Parquet [n.d.]. . https://parquet.apache.org/documentation/latest/

[7] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 583–598. https://doi.org/10.1145/2882903.2915231

[8] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. 2019. Optimal Column Layout for Hybrid Workloads. *Proc. VLDB Endow.* 12, 13 (Sept. 2019), 2393–2407. https://doi.org/10.14778/3358701.3358707

[9] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*, Vol. 5. 225–237.

[10] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "What-If" Index Analysis Utility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 367–378. https://doi.org/10.1145/276304.276337

[11] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[12] G. Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (Feb. 1994), 120–135. https://doi.org/10.1109/69.273032

[13] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE: A Main Memory Hybrid Storage Engine. *Proc. VLDB Endow.* 4, 2 (Nov. 2010), 105–116. https://doi.org/10.14778/1921071.1921077

[14] Richard A. Hankins and Jignesh M. Patel. 2003. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 417–428.

[15] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. 2006. Performance Tradeoffs in Read-Optimized Databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 487–498.

[16] Allison L. Holloway and David J. DeWitt. 2008. Read-Optimized Databases, in Depth. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 502–513. https://doi.org/10.14778/1453856.1453912

[17] IBM DB2 Manual [n.d.]. . https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.partition.doc/doc/c0021557.html

[18] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45. http://sites.computer.org/debull/A12mar/monetdb.pdf

[19] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *Proc. VLDB Endow.* 11, 7 (March 2018), 800–812. https://doi.org/10.14778/3192965.3192971

[20] Alekh Jindal, Endre Palatinus, Vladimir Pavlov, and Jens Dittrich. 2013. A Comparison of Knives for Bread Slicing. *Proc. VLDB Endow.* 6, 6 (April 2013), 361–372. https://doi.org/10.14778/2536336.2536338

[21] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. 2011. Trojan Data Layouts: Right Shoes for a Running Elephant. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 21, 14 pages. https://doi.org/10.1145/2038916.2038937

[22] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 715–730. https://doi.org/10.1145/3035918.3064049

[23] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[24] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 298–309. https://doi.org/10.14778/2095686.2095689

[25] Feilong Liu and Spyros Blanas. 2015. Forecasting the Cost of Processing Multi-Join Queries via Hashing for Main-Memory Databases. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*. 153–166. https://doi.org/10.1145/2806777.2806944

[26] Feilong Liu, Ario Salmasi, Spyros Blanas, and Anastasios Sidiropoulos. 2018. Chasing Similarity: Distribution-aware Aggregation Scheduling. *PVLDB* 12, 3 (2018), 292–306. https://doi.org/10.14778/3291264.3291273

[27] Roger MacNicol and Blaine French. 2004. Sybase IQ Multiplex - Designed for Analytics. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 1227–1230.

[28] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. 1984. Vertical Partitioning Algorithms for Database Design. *ACM Trans. Database Syst.* 9, 4 (Dec. 1984), 680–710. https://doi.org/10.1145/1994.2209

[29] Rimma Nehme and Nicolas Bruno. 2011. Automated Partitioning Design in Parallel Database Systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1137–1148. https://doi.org/10.1145/1989323.1989444

[30] Stratos Papadomanolakis and Anastassia Ailamaki. 2004. Autopart: Automating schema design for large scientific databases using data partitioning. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management* (Santorini Island, Greece). IEEE, 383–392.

[31] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2213836.2213844

[32] PostgreSQL: Documentation: 68.6. Database Page Layout [n.d.]. . https://www.postgresql.org/docs/13/storage-page-layout.html

[33] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. 2003. A Case for Fractured Mirrors. *The VLDB Journal* 12, 2 (Aug. 2003), 89–101. https://doi.org/10.1007/s00778-003-0093-1

[34] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[35] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating Physical Database Design in a Parallel Database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 558–569. https://doi.org/10.1145/564691.564757

[36] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 249–260. https://doi.org/10.1145/342009.335419

[37] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 553–564.

[38] M. Stonebraker, L. A. Rowe, and M. Hirohama. 1990. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 125–142.

[39] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-Grained Partitioning for Aggressive Data Skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1115–1126. https://doi.org/10.1145/2588555.2610515

[40] Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu. 2016. Skipping-Oriented Partitioning for Columnar Layouts. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 421–432. https://doi.org/10.14778/3025111.3025123

[41] TPC Benchmark^{TM}H Standard Specification Revision 3.0.0 [n.d.]. . http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf

[42] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 14 pages.

[43] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 193–208. https://doi.org/10.1145/3318464.3389770

[44] Jingren Zhou, Nicolas Bruno, and Wei Lin. 2012. Advanced Partitioning Techniques for Massively Distributed Computation. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2213836.2213839

[45] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. 2007. Efficient Exploitation of Similar Subexpressions for Query Processing. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 533–544. https://doi.org/10.1145/1247480.1247540

[46] Jingren Zhou and Kenneth A. Ross. 2004. Buffering Databse Operations for Enhanced Instruction Cache Performance. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 191–202. https://doi.org/10.1145/1007568.1007592

[47] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 1087–1097.