# ReCache: Reactive Caching for Fast Analytics over Heterogeneous Data

Tahir Azim⋆, Manos Karpathiotakis⋆ and Anastasia Ailamaki⋆†

⋆École Polytechnique Fédérale de Lausanne    †RAW Labs SA
{tahir.azim, manos.karpathiotakis, anastasia.ailamaki}@epfl.ch

## ABSTRACT

As data continues to be generated at exponentially growing rates in heterogeneous formats, fast analytics to extract meaningful information is becoming increasingly important. Systems widely use in-memory caching as one of their primary techniques to speed up data analytics. However, caches in data analytics systems cannot rely on simple caching policies and a fixed data layout to achieve good performance. Different datasets and workloads require different layouts and policies to achieve optimal performance.

This paper presents ReCache, a cache-based performance accelerator that is reactive to the cost and heterogeneity of diverse raw data formats. Using timing measurements of caching operations and selection operators in a query plan, ReCache accounts for the widely varying costs of reading, parsing, and caching data in nested and tabular formats. Combining these measurements with information about frequently accessed data fields in the workload, ReCache automatically decides whether a nested or relational column-oriented layout would lead to better query performance. Furthermore, ReCache keeps track of commonly utilized operators to make informed cache admission and eviction decisions. Experiments on synthetic and real-world datasets show that our caching techniques decrease caching overhead for individual queries by an average of 59%. Furthermore, over the entire workload, ReCache reduces execution time by 19-75% compared to existing techniques.

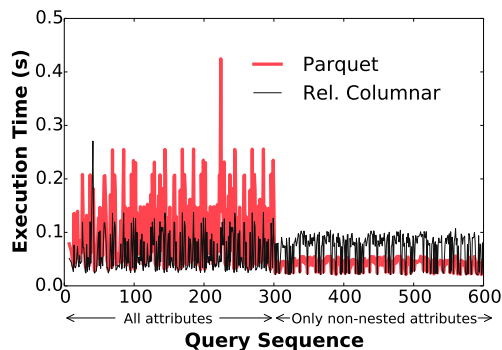**Figure 1:** Execution times for a sequence of queries on nested data, cached using Parquet and relational columnar layouts. Depending on the workload, different layouts result in significantly different query execution times.

# 1. INTRODUCTION

The volume of raw data generated everyday, ranging from sensor readings to Web logs to images and video, continues to grow at an exponential rate. The volume, variety and velocity of data generation makes pre-processing and loading of raw data extremely expensive. As a consequence, an entire industry now addresses the need to analyze raw data directly and glean useful insights from it. Nevertheless, the sheer volume of data makes analysis a time-consuming process. In addition, the raw data generated by sensors and Web-based systems often has widely varying, heterogeneous formats. Data in some formats, such as nested JSON data, can be much more expensive to parse than its relational equivalent [28, 35]. This makes performant analytics on heterogeneous data significantly more challenging.

Data analytics systems commonly use on-the-fly caching of previously parsed data and intermediate operator results to improve performance [9, 6, 28, 37, 26]. The basic idea is that when raw data is queried the first time, the system caches in memory either all of the data after parsing, or just the results of intermediate query operators. Since the data resides in an efficient binary format in memory, the cache enables faster responses to future queries on the same data.

Caching is, of course, a well-studied technique in computer systems. Unlike CPU caches where data is loaded into the cache for free before being used, database caches incur additional overhead reading, parsing and storing intermediate results into an in-memory cache. For the sake of simplicity, however, commercial DBMS and analytics systems use relatively straightforward caching policies [14, 33]. First, they cache data in memory using a fixed, pre-defined layout [8, 14, 33]. Moreover, they use relatively naive cache admission and eviction policies. The usual approach is to admit everything into the cache and evict the least-recently-used item (LRU). Another common approach is based on the five-minute rule [21] and its follow-up [20], which suggest caching disk pages that are reused every 5 minutes or less. Recent work on Vectorwise and MonetDB [37, 26] has shown promising results using cost-based

caching algorithms, but is designed for binary relational data stored in an in-memory database.

This paper focuses on efficient caching mechanisms for heterogeneous raw data formats that can contain both relational and hierarchical data. Existing caching policies result in significantly slower query performance on such diverse data formats.

First, the cost of reading and parsing raw data varies widely across different data formats. As a result, cost-oblivious caching algorithms like LRU under-perform on this data. On the other hand, cost-based algorithms need to be redesigned to capture the costs involved in processing queries over raw data.

Second, and to the best of our knowledge, there is no existing way to automatically determine the data layout for the cache that delivers the best query performance over richer data formats, such as nested data. Purely relational data layouts are not always well-suited for efficiently querying nested data. Instead, nested columnar layouts (e.g. Dremel/Apache Parquet [35, 30]) are considered a more suitable option than relational row-oriented and column-oriented layouts. Figure 1 illustrates how nested and relational column-oriented layouts for nested data differ in performance in two separate scenarios. The first 300 queries in the query sequence access attributes chosen at random from the list of all attributes. The last 300 queries, on the other hand, only choose from the set of non-nested attributes. The figure shows that neither a relational columnar layout nor a Parquet-based layout provides optimal performance in all cases. Instead, the optimal layout depends on the characteristics of the workload and the dataset. Furthermore, the nested nature of the data renders existing layout selection techniques for relational data [7, 22] inapplicable.

Third, caching intermediate results adds overhead to a query [37]. The amount of overhead depends on the type of data stored in the cache. Caching only the file offsets of satisfying tuples is cheaper, but also reduces the potential benefits of the cache. Parsing and storing complete tuples in memory has higher overhead, but enables better cache performance. Existing systems [28] provide support for statically choosing one of these caching modes, but leave open the problem of dynamically navigating this tradeoff.

This paper presents ReCache, a cache-based performance accelerator for read-only analytical queries over raw heterogeneous data. ReCache improves cache decision-making by using empirical estimates of the time cost of executing selection operators and caching their results. Additionally, it monitors the current query workload to automatically choose the optimal memory layout for caching results from relational and nested data sources. All of these statistics also enable ReCache to make intelligent cost-based cache admission and eviction decisions. Finally, if the overhead of caching an operator's results is too high, it reacts quickly by switching to a less costly caching scheme. Together, these optimizations enable ReCache to reduce overhead for individual queries, and achieve high cache performance over a query workload.

**Contributions**. This paper makes the following contributions:

- We present a cost-based approach towards caching that uses timing measurements and workload monitoring to make automatic decisions about caching policy. Using this information, our approach automatically switches to the best performing in-memory layout for caching nested data. These measurements further enable it to make more informed cache eviction decisions than LRU. Finally, our proposed approach avoids high caching overhead by choosing dynamically between a low and high overhead caching scheme for previously unseen queries.

- Based on this approach, we implement ReCache, a cache-based performance accelerator that optimizes query response time by

caching the results of intermediate selection operators in queries over CSV and JSON files. ReCache uses just-in-time code generation both to compute query results as well as to generate cache creation code tailored to the chosen caching policy.

- We show that ReCache outperforms existing analytics techniques on synthetic and real-world datasets. We present evaluations of ReCache with synthetic TPC-H data using multiple data formats, showing that ReCache's layout selection strategy results in execution time that is 53% closer to the optimal than Parquet and 43% closer than the relational columnar layout. Compared to LRU, its cost-based cache eviction policy reduces the execution time for TPC-H based workloads by 6-24% (which is 9-60% closer to that of a system with infinite cache size). Its cache admission policy reduces the response time for individual queries by an average of 59% compared to one that simply caches entire tuples in memory. Finally, on a real-world dataset, it improves query performance by 75% compared to a system that uses relational columnar layout and LRU eviction.

The next section overviews related work in more detail. Section 3 introduces ReCache's overall architecture. Section 4 then dives into ReCache's automatic layout selection algorithm and the fundamental insights driving the algorithm. Next, Section 5 describes ReCache's cache admission and eviction algorithms. Finally, Section 6 evaluates the performance benefits of ReCache on synthetic and real-world datasets, and Section 7 concludes.

## 2. RELATED WORK

Caching is a well-studied problem in computer systems in general. In the area of databases, a large body of work exists on the problem of caching the results of previous query executions and reusing them to enhance performance. Table 1 provides a summarized comparison of ReCache relevant existing work. The rest of this section surveys related work in more detail and highlights how ReCache pushes the state-of-the-art further.

**Caching Disk Pages.** All database systems retain disk pages in memory buffers for a period of time after they have been read in from disk and accessed by a particular application. The key question while caching is to decide which pages to keep in memory to get the best performance. Based on contemporary performance and price characteristics of memory and disk storage, the 5-minute rule [21] proposed caching randomly accessed disk pages that are re-used every 5 minutes or less. A revised version of the paper ten years later found the results to be mostly unchanged [20].

When caching disk pages or other data items with approximately identical costs, the provably optimal eviction algorithm is to remove the element that is accessed farthest in the future [10]. However, since future query patterns cannot be predicted with perfect accuracy, past history is often used to guide cache behavior. The assumption is that future behavior of the system will be similar to past history. History-based policies such as least-recently-used (LRU), least-frequently-used (LFU) and others attempt to approximate optimal behavior in this way and are widely implemented in computer systems.

**Cost-based Caching.** Cost-based caching algorithms, such as the family of online Greedy-Dual algorithms [46, 24, 12], improve cache performance in cases where the cost of reading different kinds of data can vary widely. These algorithms prioritize the eviction of items with lower cost in order to keep expensive items in cache. This is especially important on the Web where the latency to access different pages can vary significantly. At the same time, these algorithms account for recency of access: recently accessed items are less likely to be evicted than those accessed farther

**Table 1:** Comparison with Related Work: a checkmark indicates if an area of related work addresses a given requirement.

| Research Area | Low Overhead | Optimizes For Heterogeneous Data | Improved Net Performance |
|---|:---:|:---:|:---:|
| Caching Disk Pages | ✓ | | ✓ |
| Cost-based Caching | ✓ | | ✓ |
| Caching Intermediate Query Results | | | ✓ |
| Caching Raw Data | | ✓ | ✓ |
| Automatic Layout Selection | | | ✓ |
| Reactive Cache (ReCache) | ✓ | ✓ | ✓ |

in the past. In addition, Greedy-Dual algorithms have a theoretical guarantee that they are k-competitive with an optimal offline algorithm, i.e. they perform within a constant factor k of the optimal offline algorithm. More recently, cost-based algorithms have been used for adaptive caching in Hadoop-based analytics systems [19].

Unlike algorithms for data items with identical costs, there is no provably optimal, polynomial time cache eviction algorithm [12]. However, offline algorithms exist which can approximate the optimal to within a logarithmic factor of the cache size [24].

**Caching Intermediate Query Results.** While caching intermediate results of query execution has been frequently studied [43, 31] it has not been widely deployed due to its potential overhead and additional complexity. Recent work towards caching intermediate results of queries over relational data [37, 26] shows further promise in this direction. This work uses a cost-based approach to decide which tuples to admit to and evict from the cache. The costs are computed based on fine-grained timing measurements and sizes of the cached data. In addition to caching operator results, [16] also proposes caching internal data structures created as a side-effect of join operations. ReCache adds to this body of work by taking into account the cost of reusing a cached result, adding support for lower-overhead caches that only cache offsets of satisfying tuples, and automatically choosing the fastest data layout for the cache.

**Querying Raw Data.** Multiple systems address the need to perform queries over data that do not reside in databases. The traditional approach of fully loading raw data in a DBMS before launching queries is a blocking point for use cases where deriving insights rapidly is necessary. Apart from full loads, DBMS offer an *external tables* functionality (e.g.,via a custom storage engine [36]), in which case every query pays to parse the raw dataset from scratch. Hadoop-oriented solutions [45, 41, 9] rely on file adapters per file format. For example, when a dataset is stored in CSV files, a CSV-specific adaptor will convert all values in a binary representation as part of the querying process, thus paying for the entire conversion upfront, even if parts of the dataset are not touched by queries. Other systems [15, 5] operate in a "hybrid" mode, loading data in a DBMS on-demand based on the workload patterns or when there are available system cycles [13]. Vertical partitioning for raw data [47] assumes knowledge of the dataset and workload in advance in order to optimize workload execution time by loading the optimal subset of raw data into a DBMS.

Finally, numerous systems advocate processing raw data *in situ*, and propose techniques to mask the costs of repeatedly accessing raw data [6, 29, 28, 25, 27, 11, 40], such as indexing structures, caching, code-generated custom access paths, metadata-driven data skipping, etc.

**Caching Raw Data.** The NoDB [6] and Proteus [28] query engines for raw heterogeneous data both use caching to improve query performance. While caching enables them to optimize their query response time, their caching policies are relatively adhoc and straightforward. For example, they admit everything into the cache and use LRU for eviction with the caveat that JSON caching is assumed to be always costlier than CSV. Proteus also introduces the concept of *degree of eagerness*: caching only offsets of satisfying tuples constitutes a low degree of eagerness, while caching the entire tuple represents a high degree of eagerness. However, it does not suggest automatic policies to determine the degree of eagerness for a cached data item.

**Automatic Data Layout Selection.** Columnar databases offer substantial performance speedups on workloads that access a small subset of attributes of a table [4, 3, 23, 42]. Row-oriented databases, in contrast, perform better when queries access entire tuples instead of a small subset of attributes. Recent work extensively studies automatic selection of row-oriented or column-oriented layout for a database. HyRise [22] makes use of advance workload information to decide the best layout for an in-memory database. H2O [7] supplements this work to automatically choose a row-based, column-based or hybrid layout that minimizes response time for a dynamically changing workload. However, both H2O and HyRise only consider the space of relational queries over relational data, an assumption that allows them to compute cost of a query by estimating the number of CPU cache misses it would incur. Instead, ReCache considers the cross product of flat / nested queries over flat / nested data, which is significantly more complex. In particular, data cache misses are insufficient for estimating query cost because the Parquet format also has significant computational cost. On the other hand, the relational columnar layout often provides sub-optimal performance, as shown by Figure 1.

ReCache builds on and advances the state-of-the-art in automatic layout selection by proposing a novel algorithm for automatically choosing the fastest layout for caches of nested data in the presence of a dynamic query workload. ReCache handles the complexity of nested data using a high-level query algebra [17] and runtime profiling. The algebra enables ReCache to reason about nested data and unnest hierarchies via an explicit operator. Combined with runtime profiling, the algebra enables ReCache to gather statistics about the data in nested and flattened formats, and thus make informed layout selection decisions.

ReCache also proposes new cache admission and eviction techniques which account for the heterogeneity of previously unseen raw data. All of these decisions are made using heuristics informed by measurements of time overhead and workload characteristics.

Finally, ReCache uses existing established layout schemes (i.e., columnar and Parquet) to maximize compatibility with existing state-of-the-art and reduce engineering effort for developers that potentially want to incorporate it into existing systems.

## 3. ReCache ARCHITECTURE

Following the efficient code-generation techniques of modern research databases like HyPer [38, 39], ReCache is built on top of a just-in-time, code generation based query engine. The code generation infrastructure enables ReCache to generate the code that is

responsible for cache creation, and also generate the code that enables caches of arbitrary contents and layouts to be reused by subsequent queries. These code generation capabilities enable the overall system to fine-tune itself and, instead of acting like a static system (e.g., column store **or** row store), operate over custom storage that has been put together just-in-time based on the query workload.

## 3.1 The Proteus Query Engine Generator

We have implemented ReCache as an extension of the Proteus query engine generator [28]. Proteus relies on a high-level query algebra [17] to express and optimize queries over heterogeneous datasets, such as CSV and JSON files, as well as binary data. In addition, Proteus uses code generation techniques [32, 38] to produce a specialized query engine for each combination of incoming query and dataset instance.
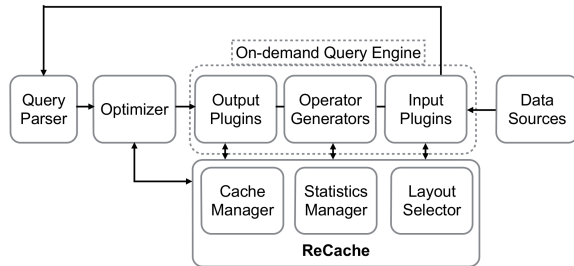


**Figure 2:** System Architecture.

Figure 2 shows the overall architecture of the system. After each query is parsed by the query parser, the optimizer uses information from ReCache to find any matches in the cache, replace operators with cached items and generate a query plan. The optimizer also updates ReCache with statistics about the data sources and attributes in the current query workload. Proteus then generates the query engine code, using ReCache to inject code that performs cache creation, timing measurements and layout switching.

Proteus handles data and query heterogeneity in two steps which end up complementing each other. To deal with input data heterogeneity, Proteus exposes a different input plugin for each type of data format [29]. The plugin is responsible for generating specialized code that considers each file format's characteristics when accessing data. For textual formats such as CSV and JSON, the corresponding input plugin populates a positional map [6, 28]; an index that acts as the "skeleton" of a file, capturing its internal structure and binary offsets of some data fields. The positional index then facilitates navigation in the file in subsequent queries, and reduces the cost of repeatedly parsing already accessed raw data.

Regarding query heterogeneity, Proteus ensures efficient query execution over relational and/or hierarchical data by generating the engine most suitable for the query at hand. Specifically, using static query operators that must process both relational and hierarchical data would result in these operators being too abstract, and introduce significant interpretation overhead [32, 38, 28]. Thus, each operator of Proteus generates code to "hard-code" itself to the current query and the model of the incoming data.

The final result of the code generation process is a custom piece of code. Proteus stitches together the code stubs generated by its input plug-ins and its query operators, in a manner that enables data pipelining through the code.

## 3.2 Cache Building and Matching

On receiving a query, ReCache uses a cost-based optimizer to generate an efficient physical query plan. For each select operator

in the plan, it inserts a "materializer" as the parent of the operator (Figure 3a). A "materializer" creates and populates a cache of the tuples that satisfy its child operator. Finally, ReCache generates, stitches together and compiles the code for the entire query using the LLVM compiler infrastructure [1, 34].

In case ReCache encounters an operator in a query whose results it has previously cached, it reuses the cache in order to speed up the query. A cached operator exactly matches an operator in the current query if both i) perform the same operation (e.g., selection), ii) have the same arguments (i.e., evaluate the same algebraic expressions), and iii) their children also match each other respectively. To do this, ReCache changes the query plan by removing the subtree under the operator. It then replaces the subtree with an in-memory scan over the previously created cache instead (Figure 3b).
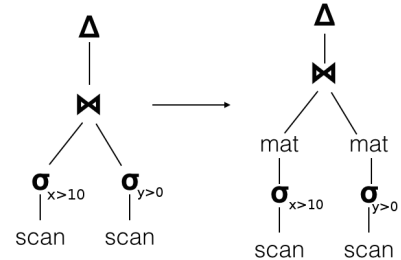


**(a)** A materializer added as the parent of an operator caches its results.
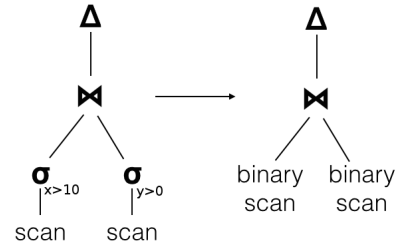


**(b)** If an exactly matching cache is found, a scan over the cache containing pre-parsed binary data replaces the operator executed over the original file scan.

**Figure 3:** Creating and reusing a cache.

## 3.3 Query Subsumption Support

Exact matches across multiple queries are relatively uncommon. More commonly, the cache contains data that is a superset of the data required by a new query. In this case, it is much faster to apply the query on the cached superset than on the raw data. The data required by the query is said to be subsumed by the data available in the cache [18, 44].

ReCache currently provides support for query subsumption on range predicates in select operators. ReCache reuses a cache if it is the result of a range predicate and if the range of a new predicate is completely covered by that of the cache. It also reuses the results of a range predicate if the predicate subsumes a clause in a conjunctive SELECT operation.

A naive approach to implement support for this kind of subsumption would be to iterate over every cached data item, and check if its range predicate covers the predicate of the new query. This approach runs in linear time to the number of objects in the cache. ReCache makes the lookup process faster by using a spatial index based on a balanced R-tree. For every numeric field in every

relation, ReCache maintains a separate spatial index. It adds the bounding box for every cached range predicate into the index. On encountering a new range predicate, ReCache looks up the corresponding spatial index to find all existing caches that fully cover the new predicate. Since the R-tree is balanced, subsuming caches can be discovered in logarithmic time. While inserting into the R-tree is more expensive than appending to an array, our experiments show that this overhead is negligible over a variety of workloads, ranging from 2-15$\mu$s.
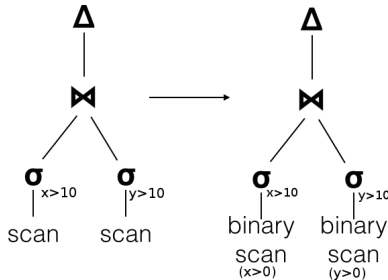


**Figure 4:** For query subsumption, a scan over the original data source is replaced by a scan over an in-memory binary cache containing a superset of the data requested.

If a subsuming cache is found, a scan over an underlying data source is replaced by a scan over the in-memory cache. Since the cache is still just a subsuming cache (as opposed to an exact match), the operator still needs to be applied on top of the scan in order to obtain the correct results from the subsuming cache. The required query rewrite in this case is illustrated in Figure 4.

Note that the techniques and insights of ReCache are applicable to all query types and operators. This paper focuses on subsumption of range predicates because select operators are a typical bottleneck of in-situ data accesses [6, 29, 28]: queries over raw data pay a high cost to access the data and filter out unwanted values before forwarding the data to operators higher in the query tree. In addition, ReCache focuses on cases of subsumption and reuse for select operators that are low in the query tree to increase the probability of future queries being able to re-use a cached result.

## 4. AUTOMATIC CACHE LAYOUT

One of the key questions for caching heterogeneous data is choosing the cache layout that yields the best performance. To maximize the performance of an in-memory cache, ReCache monitors the query workload to decide the best layout for the cache.

Automatically determining in-memory or on-disk data layout is a deeply studied area of database research, as described in Section 2. Heterogeneous data adds a further dimension to ReCache's layout selection strategy, since it does not fit the relational model directly. Specifically, nested data needs to first be flattened in order to make it fit into the relational model. One approach is to "shred" the nested data into multiple tables with foreign keys used to connect the tables together. The more prevalent technique, however, is to simply flatten the data into a single relational table and then separate out its columns. For instance, the JSON entry {"a":1,"b":4, "c":[4,6,9]} would be flattened into three rows to match the corresponding relational format: {"a":1,"b":4,"c":4}, {"a":1, "b":4,"c":6} and {"a":1,"b":4,"c":9}. Naturally, this leads to data duplication if any field in the JSON entry is an array. Data duplication in the cache is problematic because each query has to process more data, resulting in lower performance.

The column striping approach used by Dremel/Parquet [35, 30] addresses this problem by storing each field in a separate column without duplication, and attaching a few bits of additional metadata with each column entry. In order to reconstruct the original nested structure, Parquet initializes a finite state machine (FSM) which points to the start of each column. The state of the FSM corresponds to its current position in each column. Before each state transition, the FSM outputs the data stored in the current position for each column. The FSM then transitions to a new state based on the metadata values associated with each column's current position. The FSM terminates when all columns have been completely read. By eliminating duplication, this technique reduces memory accesses. This also makes Parquet an ideal layout for accessing data on disk. On the other hand, the FSM-based reconstruction algorithm requires significantly more computation and adds more CPU pipeline-breaking branches into the instruction stream [35].

### 4.1 Characterizing the Tradeoffs

**Accessing Shorter Columns.** A pair of experiments help explain the factors affecting cache performance using Parquet or a relational columnar layout. One of the key benefits claimed by Parquet is that some columns can be significantly shorter than others. So, in theory, if an analytics query only accesses the shorter columns, it can be executed much more quickly.

Our first experiment tests this claim by running two different kinds of queries on nested JSON data. The experiment uses a 2.5 GB JSON data file named **orderLineitems**, generated from the TPC-H SF1 **lineitem** and **orders** tables. Each line in the file is a JSON object mapping an **order** record to a list of **lineitem** records. Each **lineitem** record is itself a JSON object with a set of nested fields describing the **lineitem**. On average, about four lineitem records are associated with each order record. For this experiment, the queries in the workload are a sequence of 600 select-project-aggregate queries with the following format:

```
SELECT agg(attr_1), ..., agg(attr_n)
FROM orderLineitems
WHERE <range predicates with random selectivity
over randomly chosen numeric attributes>
```

For the first 300 queries, the queries access random attributes from the list of all attributes in each record. For the next 300, the queries only choose from the set of non-nested fields describing an **order** record. We populate the caches beforehand in order to isolate the performance of the cache from the cost of populating them.

As shown in Figure 1, we find that a Parquet-based layout is indeed better suited for a workload where only the non-nested attributes are accessed frequently. On the other hand, a relational columnar format performs better when both nested and non-nested attributes are accessed.

**Querying Data with Large Nested Fields.** The second benefit that Parquet claims is that by eliminating duplicated data, it enables a more compact data representation. Therefore, queries on data using Parquet representation should require fewer memory references and fewer CPU cache misses, resulting in faster performance.

We test this hypothesis by executing scans on in-memory caches of synthetic data with increasingly large nested arrays. The dataset for this experiment has the same format as **orderLineitems**. Instead of generating it from the **lineitem** and **orders** tables, we populate it with synthetic data from a uniform distribution. We then populate the **lineitem** array associated with each **order** with an increasing number of elements and test how cache performance using a Parquet-based layout differs from a relational columnar layout.
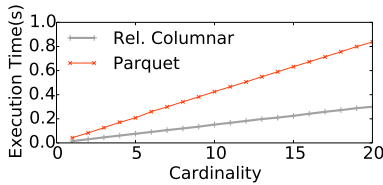
**Figure 5:** Execution times for full scans over nested data cached using Parquet and relational columnar layouts. Each data record is associated with a nested array of varying cardinality. Even with large cardinality, Parquet performs about 3x slower than the relational columnar layout.
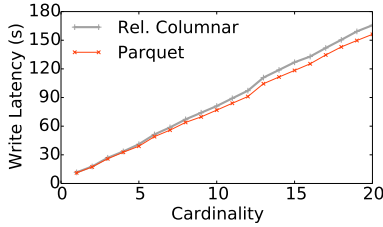


**Figure 6:** Time required to write nested data into an in-memory cache using Parquet and relational columnar layouts. Each data record is associated with a nested array of varying cardinality.

Figure 5 shows the results. Surprisingly, Parquet continues to perform about 2.8x slower than a relational columnar layout as the size of the nested array increases. Profiling the process shows that while Parquet incurs far fewer memory references, it does not incur significantly fewer cache misses. Prefetching allows the CPU to effectively avoid cache misses in both layouts, so the difference in data size due to duplication does not matter significantly. Instead, Parquet's extra computational cost remains the dominant factor affecting query performance.

These experiments lead to the following two insights. First, the presence of large collections or high cardinality keys in nested data does not automatically make it better suited for a Parquet-based layout. Second, when queries only access columns with fewer elements, both data size and instruction count is reduced, making Parquet a far better choice for this scenario. Our heuristic for choosing between nested and relational columnar layouts (described next) draws directly from these basic insights.

## 4.2 Choosing between nested and relational columnar layouts

By default, ReCache caches nested data in the Parquet layout because it generally requires substantially fewer writes to memory. We confirm this observation in Figure 6: the smaller memory footprint of the Parquet layout makes it faster to write to memory.

To determine if a relational columnar layout is a better option for a cached item, ReCache profiles the performance of each query that touches the item. For each query $Q_i$, it measures the data access cost $D_i$ and the computational cost $C_i$ for scanning each cached data item. The data access cost is measured as the time spent loading data from the cache. The computational cost is measured as the time spent evaluating branch conditions and any other additional processing. In addition, ReCache tracks the number of columns $c_i$ and rows $r_i$ accessed by query $Q_i$, and the total number of rows in the cache if it were flattened into a relational columnar layout, $R$.

In order to decide whether to switch the cache from Parquet to
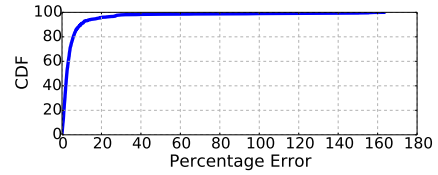


**Figure 7:** Percentage error in the actual cost of scanning a cache versus the cost predicted by ReCache's cost model on the **order-Lineitems** dataset.

a relational columnar layout, ReCache computes the total cost of answering queries using Parquet and compares it to the estimated cost of answering queries using a relational columnar layout. In addition, it estimates the transformation cost $T$ of switching from one layout to another. The estimates are computed by extrapolating the cost of accessing $r_i$ rows to the case where all $R$ rows are scanned, as given below:

$$Cost_{parquet} = \sum (D_i + C_i) \tag{1}$$

$$Cost_{relational} = \sum D_i \frac{R}{r_i} \tag{2}$$

$$T = max((D_i + C_i)\frac{R}{r_i}) \tag{3}$$

ReCache switches to a relational columnar layout if $Cost_{parquet}$ is larger than $(Cost_{relational} + T)$ over past queries accessing the cache. If a switch is performed, it moves forward the window for further query tracking to look at new incoming queries. ReCache does not use a fixed window size for query tracking in order to avoid frequent switching overhead on a rapidly changing workload. Instead, it considers all queries from the start of the window to the last observed query.

As an example, consider a scenario where ReCache is serving queries over the **orderLineitems** table and sees 5 queries over a cached item. Each query has approximately the same data and compute cost, with $\sum D_i = 1000$ and $\sum C_i = 2000$. Each order in the cache is associated with 4 lineitems. Therefore, given the initial Parquet layout, if these queries access only the non-nested **order** fields, $r_i = \frac{R}{4} = 100$ and $R = 400$. Then $Cost_{parquet} = 3000$, $Cost_{relational} = 4000$ and $T = 2400$. In this case, ReCache keeps the layout unchanged. On the other hand, if the queries also access the **lineitem** fields, $r_i = R = 400$. Then $Cost_{parquet} = 3000$, $Cost_{relational} = 1000$ and $T = 600$. As a result, ReCache switches to the relational layout. After switching, ReCache starts tracking queries after the fifth query for future decision-making.

For the opposite case of switching from a relational columnar layout to Parquet, ReCache uses a similar algorithm. The only complication is that the relational columnar layout has negligible computational cost. This makes it difficult to estimate Parquet's computational cost by extrapolation. Instead, ReCache estimates it as the computational cost of a query in the cache's history that used the Parquet layout, and is closest to the current query in terms of rows and columns accessed. Denoting the function computing this estimate as $ComputeCost(rows, cols)$, the cost equations are:

$$Cost_{relational} = \sum D_i \tag{4}$$

$$Cost_{parquet} = \sum (D_i + ComputeCost(r_i, c_i))\frac{r_i}{R} \tag{5}$$

The transformation cost $T$ is computed in the same way as before. In this case, ReCache switches to a Parquet-based layout if
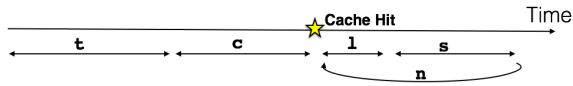
**Figure 8:** Given time cost of operator execution $t$, caching overhead $c$, looking up a matching operator $l$, scanning the cache $s$, number of times operator re-invoked $n$ and cache size $B$, the cache benefit metric is given by $(n*(t+c-s-l))/log(B)$.

$Cost_{relational}$ is larger than $(Cost_{parquet} + T)$ over the last $n$ queries accessing the cache.

We validate this cost model using the same set of queries as in Figure 1. We first run each query using data cached in the Parquet layout during which we measure the cost of scanning the cache and estimate the cost if it were in the relational columnar layout. We then repeat this test with the layouts interchanged. The results in Figure 7 show that the cost model accurately estimates the actual cost of using either Parquet or the relational columnar layout. The error is within 10% of the actual cost for 90% of queries and within 30% for 98% of queries.

### 4.3 Choosing between relational row and column oriented layouts

In order to choose between column-oriented and row-oriented relational formats, ReCache monitors the set of queries in the workload and the columns of R accessed together by the queries. Using a minor variation of the algorithm described in H2O [7], ReCache computes the number of data cache misses for a row-based layout as well as a column-based layout. The number of data cache misses act as an estimator for CPU performance: the more the estimated number of misses, the worse the performance. This enables ReCache to decide the best performing layout for the cache.

## 5. CACHE ADMISSION AND EVICTION

Section 4 addressed the question of choosing the best performing in-memory layout for nested and tabular data. We now consider the more long-standing fundamental problem in cache design of choosing which data items to keep in the cache and which to evict. Most systems simply evict the least-recently-used items (LRU) to reclaim space in the cache. While LRU is widely deployed due to its simplicity, it is known to underperform when the cached items have widely varying costs [12]. Instead, ReCache uses a cost-based eviction algorithm which overcomes LRU's limitations. For data whose cost is not known beforehand, ReCache uses a reactive, cost-aware cache admission algorithm to reduce caching overhead.

### 5.1 Cache Eviction

In case of data for which some history of cost measurements is available, ReCache is able to make more informed caching decisions. To this end, ReCache computes a benefit metric for each cached item and uses it to make cache admission and eviction decisions. To decide on cache admission for a data item, if the benefit metric is higher than existing items in the cache, we proceed to add the data item to cache. Otherwise, we revert to the admission policy for unseen data. To decide on which item to evict, ReCache removes the item with the minimum value of the benefit metric. Computing the benefit metric requires ReCache to measure the costs of reading and parsing heterogeneous data files, executing database operators, caching operator results in memory, and subsequently scanning over the in-memory cache.

ReCache uses a novel instance of a Greedy-Dual algorithm [46] for making eviction decisions. Algorithm 1 outlines the overall

---

**Algorithm 1** Cost-based Eviction in ReCache

Initialize $L \leftarrow 0$
**function** EVICT
    $Items \leftarrow$ set sorted (ascending) by key
    **for** each cached item $p$ **do**
        Compute the benefit metric $b(p)$
        $H(p) \leftarrow L(p) + b(p)$
        Add $p$ to $Items$ with key $H(p)$
    **end for**

    $Diff \leftarrow TotalCacheSize - CacheSizeLimit$
    $C \leftarrow$ set sorted (descending) by key
    **while** $Diff \geq 0$ **do**
        $p \leftarrow$ cached item with $H(p) = min_{p \in Items}(H(p))$
        $Diff \leftarrow Diff - Size(p)$
        Pop $p$ from $Items$ and add to $C$ with key $Size(p)$
        **if** $L \leq H(p)$ **then** $L \leftarrow H(p)$
        **end if**
    **end while**

    $Diff \leftarrow TotalCacheSize - CacheSizeLimit$
    **while** $Diff \geq 0$ **do**
        $p \leftarrow$ largest item in $C$
        $Diff \leftarrow Diff - Size(p)$
        Evict $p$ and remove from $C$
        $p \leftarrow$ smallest item in $C$ with size $\geq Diff$
        **if** $p$ exists **then** Evict $p$ and return
        **end if**
    **end while**
**end function**

**function** ONINSERTORACCESS(Item $p$)
    **if** cache size exceeded **then** EVICT();
    **end if**
    $L(p) \leftarrow L$
    Add $p$ to the cache
**end function**

---

algorithm. The Greedy-Dual algorithm requires the definition of a benefit metric $b(p)$ which specifies the benefit of keeping item $p$ in cache. As illustrated in Figure 8, the benefit metric used by ReCache is based on the following set of measurements:

1. $n$, how many times an operator's cache has been reused.

2. $t$, the time incurred executing the operator. This includes the time spent parsing the fields involved in executing the operator. It also includes any additional time spent scanning the file to determine offsets of various fields, creating indexes, etc.

3. $c$, the time incurred caching the results of the operator in memory. This includes the time spent parsing the cached fields of each record.

4. $s$, the time spent scanning the in-memory cache when it is reused.

5. $l$, the time spent finding a matching operator cache

6. $B$, the size of an operator cache in bytes

Based on these parameters, ReCache uses a benefit metric which considers the number of times a cached item is reused, its size, the time saved by using it, and the time needed to reconstruct it in case it is evicted. Note that if a cached item is evicted, it will take

time $t + c$ to reconstruct it. This is also the time that is saved when the item is found in the cache. On the other hand, the savings are tempered by the costs involved in looking up and reusing a cached item given by $s + l$. So the overall benefit of reusing a cached item is $t + c - (s + l)$. Multiplying this by $n$ gives the total benefit accrued so far. Finally, in order to give slightly higher preference to smaller cached items, the benefit metric is divided by $log(B)$. The resulting benefit metric, $b(p) = n * (t + c - (s + l))/log(B)$, is always non-negative assuming the cost of lookup and the cost of scanning the in-memory cache are small.

In addition to the benefit metric $b(p)$, the basic Greedy-Dual algorithm associates a value, $H(p)$, with each cached item $p$, and a global baseline value, $L$. When an item $p$ is brought into or accessed from the cache, $H(p)$ is set to $(L + b(p))$. If the cache does not have enough capacity to add a new item, it evicts items in ascending order of $H(p)$. $L$ is then set to the $H(p)$ value of the last item evicted. The basic idea is that the $H(p)$ values of recently accessed items retain a larger portion of the original cost than those of items that have not been accessed for a long time.

Implementing this algorithm unchanged leads to more items being evicted than necessary. For example, if 1 GB of space needs to be reclaimed, the unchanged algorithm might evict items of size 100 MB, 200 MB, 300 MB and 800 MB in that order. In fact, evicting only two of the last three items is enough to reclaim the required amount of space. Unfortunately, finding the optimal set of elements to evict in this scenario reduces to the NP-complete Knapsack problem. Therefore, ReCache uses a heuristic approach to find an efficient, approximate solution. It iterates through the items the original algorithm would evict and considers them in descending order of their byte size. This enables the required space to be reclaimed after evicting many fewer items. Furthermore, since these objects were going to be evicted by the original algorithm anyway, it does not affect its correctness.

Furthermore, ReCache does not update $H(p)$ only when an item $p$ is accessed. Instead, for each cached item $p$, it recomputes the value of $H(p)$ from its individual components whenever an eviction decision needs to be made. This obviously adds overhead to the eviction process, but this additional overhead of a few milliseconds is negligible compared to the typical execution time of a query. Continuously reconstructing the benefit metric for each cached item in this way has a major additional benefit. If the underlying query engine changes how it reads a file, the new costs are immediately reflected in the benefit metric. For instance, the query engine might create an index on the file to allow faster traversal. In this case, the cost of any cached data originating from this file changes and the benefit metric immediately reflects this change. We find that in our workloads, using an unchanging benefit metric for each cached item causes execution time to increase by up to 6%.

**Minimizing Cost Monitoring Overhead**. A naive way to measure the costs of various operations during a query is to invoke timing system calls, such as clock_gettime(), before and after every operator is invoked during the query. However, this approach adds a significant amount of overhead. Our measurements on a set of select-project-join queries over TPC-H data show that this approach adds a runtime overhead of 5-10% to each query.

Instead, ReCache reduces this overhead by executing timing system calls on less than 1% of records selected uniformly at random. While this adds approximation error to the timing measurements, it still yields sufficiently accurate estimates for the cache to make informed decisions. Moreover, since the system calls are invoked very rarely, CPU branch prediction makes the overhead negligible over large datasets.

## 5.2 Cache Admission

Making intelligent caching decisions requires information on the cost and benefit of data items cached in the past. If a query refers to data that has not been previously accessed, this information is not available. An eager default policy, which fully parses and stores every tuple, can result in high caching overhead, but this overhead may not be amortized by future queries. On the other hand, a lazy caching policy, which only caches the file offsets of satisfying tuples, has a lower overhead but also a lower benefit if the cache is reused for a future query. Thus, an inherent tradeoff exists between the overhead of caching and subsequent cache performance.

ReCache navigates this tradeoff by using a more reactive cache admission policy to reduce the overhead for individual queries compared to eager caching. Initially, it starts caching a small sample of data at the beginning of the file, both eagerly and lazily. As it scans over these records, it tracks both the time spent caching the records and the time spent executing other instructions. Assuming that the sample is representative, ReCache then compares the percentage overhead added by the caching operation against a user-specified threshold value. If the threshold is exceeded, it switches to a lazy policy. Otherwise, it continues in eager mode. If a lazy cached item is accessed again, it is replaced by an eager cache for faster subsequent access. Moreover, as long as all cached items from the same file are not evicted, ReCache assumes that it is still part of the working set and eagerly caches further accesses to the same file.

The presence of operators that break the CPU pipeline, such as joins, complicates the effectiveness of this approach. A join can itself be computationally expensive, so when the time cost of a subsequent caching operator is measured over a small sample, the percentage overhead can seem very small. For example, suppose $(R \bowtie S \bowtie \sigma_{key>0}(T))$ is the query being executed over three data sources with millions of records. Suppose the first join operation over relations R and S takes 10 seconds. The next step would be to scan through the relation $T$, and cache the result of $\sigma_{key>0}(T)$. Suppose it takes 100 ms to cache the first $n = 1000$ results of $\sigma_{key>0}(T)$. This would represent an overhead of just 1%. However, since many tuples of $T$ are yet to be processed, the percentage overhead of caching all of $\sigma_{key>0}(T)$ will eventually be much higher.

To deal with this problem, ReCache generates two timestamps $t_{o1}$ and $t_{c1}$ at the beginning of a sample. $t_{o1}$ stores the total overall time spent executing the query up to that point, while $t_{c1}$ stores the total time spent caching results. At the end of the sample, it recomputes these two timestamps storing them in $t_{o2}$ and $t_{c2}$. Assuming the entire file is $N$ times larger than the sample, it estimates $t_o$ up to the end of the file as $(t_{o1} + N(t_{o2} - t_{o1}))$ and $t_c$ as $(t_{c1} + N(t_{c2} - t_{c1}))$. Then $t_c/t_o$ yields a much more accurate estimate of the caching overhead, which ReCache then uses to decide whether to proceed with eager or lazy caching.

Experiments in Section 6.2 show that this technique allows the cost of caching to approach that of lazy caching, while retaining the benefit of an eager policy.

## 6. EVALUATION

In this section, we present a series of experiments to evaluate ReCache and validate its ability to increase cache efficiency. All experiments are conducted on a Intel Xeon E5-2660 processor with a clock speed of 2.20 Ghz and 8 cores per socket. The machine has two sockets with 16 hardware contexts per socket. Each core has a 32 KB L1 instruction cache, 32 KB L1 data cache, 256 KB L2 cache, and 20 MB L3 cache is shared across the cores. Finally the machine has 132 GB of RAM and a 450 GB, 15000 RPM disk with a 6 Gb/s throughput. ReCache uses LLVM 3.4 to generate custom

code with the compilation time being at most ~50 ms per query. We run all experiments in single-threaded mode. We focus on caching the outputs of scan and select operators, which are generally the main bottleneck when running queries over raw data.

We evaluate ReCache using three workloads. For the first workload, we use the TPC-H **lineitem**, **customer**, **partsupp**, **order** and **part** tables as input CSV files, using scale factor 10 (SF10 - 60M lineitem tuples, 15M order tuples, 8M partsupp tuples, 2M part tuples and 1.5M customer tuples). To test performance over JSON data, we convert the TPC-H SF10 **lineitem** and **orders** tables from CSV format to a 20GB JSON file for lineitems and a 3.5GB file for orders. All experiments operate over warm operating system caches. The data types are numeric fields (integers and floats). The queries in this workload are a sequence of 100 select-project-join queries over TPC-H SF-10 data with the following format:

```
SELECT agg(attr_1), ..., agg(attr_n)
FROM subset of {customer,orders,lineitem,partsupp,
part} of size n
WHERE <equijoin clauses on selected tables>
AND <range predicates on each selected table with
random selectivity>
```

For each query in this workload, each table is included with a probability of 50%. One attribute is randomly selected for aggregation from each of the selected tables. Each table is joined with the other table on their common key. The range predicate is applied to random numeric columns of the selected tables.

The second workload is a real-world dataset of spam email logs provided by Symantec. The Symantec dataset features a combination of i) numeric and variable-length fields, ii) "flat" and "nested" entries of various depths, as well as iii) fields that only exists in a subset of the JSON objects. Additionally, the dataset is the target of both relational and hierarchical queries. Symantec periodically collects this data as JSON files from spam traps distributed around the world. The JSON data contains information about spam e-mails, such as the mail body and its language, its HTTP content type and its origin (IP address, country). The JSON dataset used in this paper is 3.4 GB in size and contains 3 million objects. The JSON files are the input to Symantec's data mining engine, which produces CSV files containing an identifier for each email, summary information and various classes assigned to each email. For this paper, we use CSV files containing 40 million records and a total size of 2.2 GB.

The third workload uses the **business**, **user** and **review** JSON files of Yelp's open-source dataset [2]. In total, these files are 4.8GB in size and contain 144K, 1M and 4M records respectively. The query workload comprises select-project-aggregate queries on the three JSON files[1].

The experiments validate the following hypotheses:

- ReCache's algorithm for choosing between a relational columnar layout and Parquet achieves the best of both worlds (Section 6.1).

- ReCache's cache admission policy reduces caching overhead for individual queries compared to a policy of admitting everything as fully parsed tuples. Moreover, it does not affect query response time over the whole workload (Section 6.2).

- ReCache's cost-based cache eviction policy leads to improved workload execution time compared to LRU (Section 6.3).

- Integrating these features together, ReCache delivers improved response times for queries over a diverse data set (Section 6.4).

---

[1]For purposes of reproducibility, a detailed description of the query workload is available at: `https://github.com/tahirazim/recache-evaluation/raw/master/vldbQueryInfo.pdf`

## 6.1 Automatic Cache Layout

This section presents an evaluation of ReCache's automatic cache layout algorithm. First, we evaluate the benefits of using ReCache on workloads with arbitrary select-project-aggregate queries and nested JSON datasets. Next, we carry out a sensitivity analysis of the algorithm by varying the data attributes and data sources accessed by the query workload.

### 6.1.1 Performance Evaluation

We first evaluate the performance benefit of ReCache's mechanism for automatically choosing a relational or Parquet-based layout for column-oriented caching of nested data.

We build on the experiment described in Section 4.1 using the same 2.5GB **orderLineitems** JSON file and the same sequence of select-project-aggregate queries. For this experiment, we populate the caches beforehand in order to isolate the performance of the cache from the cost of populating them. We then test whether ReCache is able to adapt dynamically to this changing workload.

Figure 9a shows the results. The relational columnar layout clearly outperforms the Parquet layout for the first 300 queries because reading it incurs very little computational cost compared to Parquet. In contrast, Parquet outperforms the relational columnar format for the last 300 queries because it only has to access the non-nested columns which allows it to iterate over 4x fewer rows. ReCache's automatic layout selection algorithm monitors the workload and quickly switches to the caching layout that performs best for the workload. Over the entire workload, ReCache's execution time is 53% closer to the optimal than Parquet and 43% closer than the relational columnar layout.

We further validate these results on two additional query workloads over the same dataset. Figure 9b shows how ReCache reacts on a workload where the set of queried attributes switches every 100 queries. As before, ReCache switches the layout of each cached item to the most suitable one for the workload. However, since the cost model considers all queries that have utilized the cache since the previous switch, it prevents the layouts from switching after every 100 queries. This prevents excessive switching overhead. Overall, ReCache is 68% closer to the optimal than Parquet and 57% closer than the relational columnar layout.

Finally, Figure 9c shows ReCache's performance on a random workload where a query can arbitrarily choose to access any attribute. Without a clear pattern in the workload, ReCache is unable to optimize for the case of either nested or non-nested attribute access, and its performance resembles that of the Parquet layout.

We further confirm the effectiveness of ReCache's automatic layout strategy by executing two workloads of 2000 distinct queries each on Symantec's JSON-based spam data. The first workload consists of select-project-aggregate queries with only 10% randomly chosen queries accessing nested attributes and the remaining 90% accessing only non-nested attributes. The second workload inverts this ratio with 90% of queries accessing nested attributes and 10% accessing only non-nested attributes. The workloads run with no limit on cache size to isolate the effects of cache eviction from the automatic layout algorithm. Each experiment starts with an empty cache, so the initial cache creation cost is included in the results.

Figure 10 shows the results for these two workloads. The first workload is a better fit for the Parquet format because it rarely accesses nested attributes. As a result, ReCache's performance tracks closely with Parquet. In contrast, the relational columnar layout performs almost 29% slower than ReCache on this workload.

On the other hand, the second workload is better suited for the relational columnar layout. In this case, ReCache reacts by switching quickly to the relational columnar layout and closely matches its
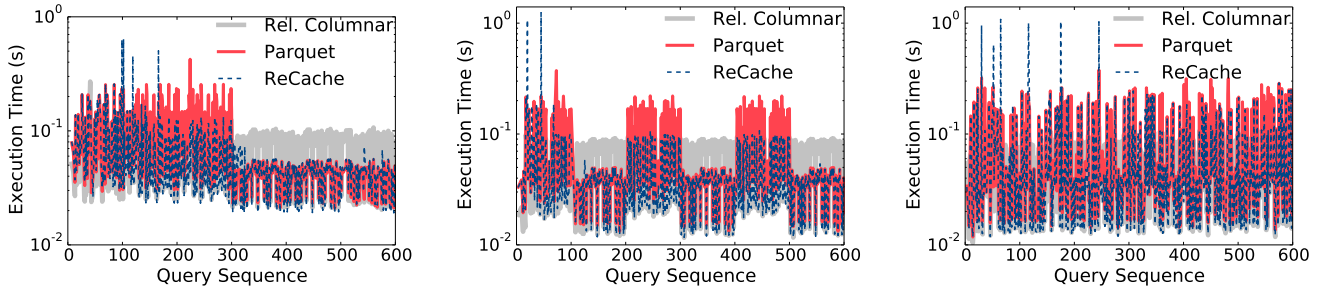
**(a)** First 300 queries access attributes chosen randomly from all attributes. Last 300 only choose from non-nested attributes.

**(b)** Queries 1-100, 201-300 and 401-500 access attributes chosen randomly from all attributes. The rest choose only from non-nested attributes.

**(c)** 50% of queries choose only from non-nested attributes, while the remaining choose from all attributes.

**Figure 9:** Execution times for a sequence of queries on nested data, cached using Parquet, relational columnar and ReCache's automatic layout strategies. The spikes indicate the overhead of switching from Parquet to relational columnar layout.
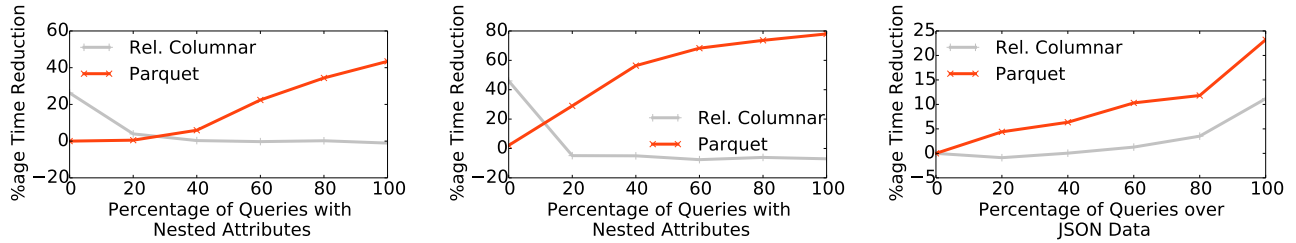


**(a)** Percentage reduction in execution time relative to Parquet and relational columnar layouts depends on how many queries access nested attributes in the Symantec data.

**(b)** Percentage reduction in execution time relative to Parquet and relational columnar layouts on the Yelp data.

**(c)** Percentage reduction in execution time increases if more queries access the nested JSON component of the Symantec data.

**Figure 11:** Sensitivity Analysis - Automatic Layout Selection.
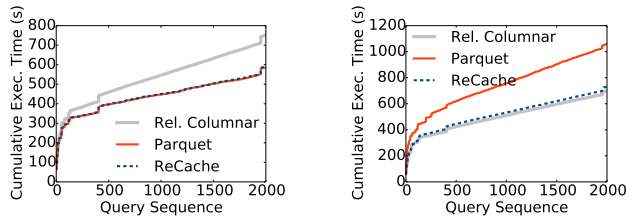


**(a)** Only 10% queries accessing nested attributes.

**(b)** 90% queries accessing nested attributes.

**Figure 10:** Cumulative execution time for two workloads of 2000 queries each, executed over the JSON component of the Symantec spam dataset. Intermediate operator results are cached using Parquet, relational columnar and ReCache's automatic layout strategy.

performance. The small 4% difference in performance is because ReCache has a default Parquet layout and switches layouts after tracking a few queries. The Parquet layout performs 44% slower than ReCache on this workload.

### 6.1.2 Sensitivity Analysis

Next, we carry out a sensitivity analysis of the layout selection algorithm to determine how it is affected by changes in the workload. To analyze this behavior, we run a workload containing queries on both the CSV and JSON components of the Symantec spam dataset. We use an unlimited cache size in order to isolate the effect of cache eviction from the cache layout selection algorithm.

Each experiment starts with an empty cache, so the initial cache creation cost is included in the results.

First, we measure how the relative performance of ReCache is affected by the percentage of queries that access nested attributes. We run a set of select-project-join and select-project-aggregate queries and vary how many of those queries access randomly chosen nested attributes from the JSON file. Overall, 90% of the queries access JSON data, 90% are select-project-aggregate queries, and 10% are select-project-join queries across CSV and JSON data.

The results are shown in Figure 11a. As more nested attributes are accessed, ReCache performs increasingly better than a Parquet-based layout. This is consistent with our observations in Section 4.1 that Parquet is an inefficient cache layout when queries access nested attributes. The relative performance is inverted in the case of the relational columnar layout which performs worse when queries mostly access non-nested attributes.

We validate these results further by running the experiment on the Yelp dataset using a sequence of select-project-aggregate queries with unlimited cache size. The results in Figure 11b show a similar trend. ReCache performs increasingly better than Parquet when more queries access nested attributes. On the other hand, it achieves higher performance improvement over the relational columnar layout when fewer queries access nested attributes. In fact, with more nested attributes, the graph shows a small 4-7% increase in execution time due to performance monitoring overhead and the initial delay in switching to a relational columnar layout.

Second, we confirm our hypothesis that ReCache is more beneficial when queries in a workload access JSON data more frequently than CSV. This is based on the intuition that on narrow relational

data like ours, switching between row-based and column-based layouts provides little gain. To test this, we use a workload consisting of select-project-aggregate queries on Symantec data and vary the percentage of queries querying JSON. For this experiment, only the last 50% of queries access nested data. Figure 11c confirms our hypothesis: ReCache outperforms Parquet and relational columnar layouts by a wider margin when more queries access JSON data.

**Discussion**. The results in Figures 9 and 11 show that ReCache automatically switches cached data to the higher performance layout whenever the queries in the workload have properties favoring one layout over another (e.g. if accesses to nested or non-nested attributes dominate). Figure 10 further shows that, on such workloads, as the size of the query workload increases, ReCache provides increasingly greater improvement in execution time compared to the unfavorable layout.

## 6.2 Cache Admission

We evaluate the cache admission policy for previously unseen queries using the select-project-join workload described in Section 6 over TPC-H SF-10 data. ReCache is configured to cache and, where possible, re-use the outputs of the selection operators in each query.

Figure 12a shows the caching overhead incurred by ReCache as well as by the lazy and eager strategies. The cost of lazy caching is substantially smaller, with a mean overhead of 2.5% compared to 20% for eager caching. Of course, as we will see, the benefit of lazy caching is also much smaller. On the other hand, using its cost-adaptive caching mechanism, ReCache is able to switch automatically to lazy for queries with high cost of caching eagerly. This enables ReCache to achieve a much lower mean overhead of 8.2% compared to eager caching for individual queries, a reduction of 59%. This overhead comes from the extra monitoring cost, the cost of eagerly caching the initial sample, and occasionally underestimating the caching cost and continuing with eager caching.

Figure 12b further quantifies the tradeoffs involved in choosing an appropriate overhead threshold for switching from lazy to eager caching. On this workload, a maximum overhead threshold of 10% enables ReCache to approach the overhead of lazy caching.

Besides reducing caching overhead for individual queries, we also show that ReCache improves query response when the queries are run in sequence as a single workload. In this case, ReCache is able to leverage its query subsumption capability to answer these non-identical queries using previously cached results. Figure 13 shows the response time of ReCache compared to a system using lazy caching, eager caching and no caching. Overall, the query response time improves by 62% compared to a system with no cache and 47% compared to a lazy cache.

Furthermore, the graph for ReCache flattens out near the end of the workload, while the cumulative execution time for lazy caching continues to increase. This indicates that the improvement in performance will tend to increase even more as further queries run over the same dataset.

Compared to an eager cache, response time for the cost-adaptive cache over the entire workload is virtually the same, with a difference of just 3%. Eager caching performs well over a long query sequence touching a small set of tables due its aggressive caching strategy, which may be expensive in the short term but eventually pays off over the long run. ReCache's admission policy adds overhead on such a workload because it uses lazy caching if the short-term overhead is large. The cache then only switches to eager mode when it is first reused. Despite this overhead, ReCache performs almost identically to the eager caching strategy in the long term while reducing caching overhead for individual queries by 59%.



**(a)** Using a switching threshold of 10% .

**(b)** Effect of varying the switching threshold.
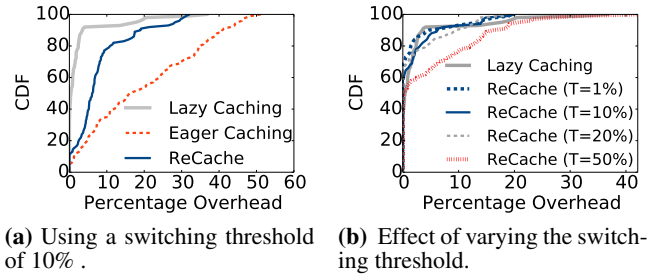
**Figure 12:** Percentage caching overhead (in ascending order) for 100 individual queries executed over the TPC-H SF-10 dataset.
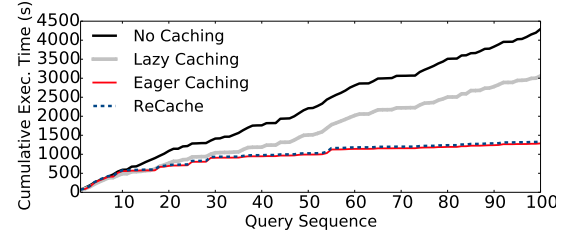


**Figure 13:** Cumulative execution time while running a workload of 100 queries over the TPC-H SF-10 dataset.
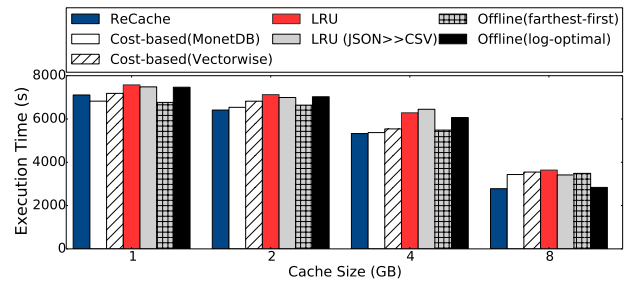


**Figure 14:** Cumulative execution time using various cache sizes for a workload of 100 queries over the TPC-H SF-10 dataset.

## 6.3 Cost-based Eviction

We evaluate the costs and benefits of the cost-based eviction policy (Section 5.1) using the workload described in Section 6, comprising 100 select-project-join queries over a TPC-H SF-10 dataset. In order to add more heterogeneity to the workload, we convert the lineitem table into JSON format and use the resulting file as our input. As before, ReCache is configured to cache and, if possible, re-use the outputs of the selection operators in each query.

Figure 14 plots the total execution time for the workload using ReCache's cost-aware eviction algorithm, two state-of-the-art cost-based eviction algorithms [37, 26], LRU, and Proteus' algorithm [28] of using LRU but with higher priority given to JSON than CSV. In addition, we compare against two offline algorithms: (i) farthest-first, which removes the item that is accessed farthest in the future and is provably optimal for unweighted caches, and (ii) log-optimal [24], which guarantees that the difference from the optimal is within a logarithmic factor of the cache size.
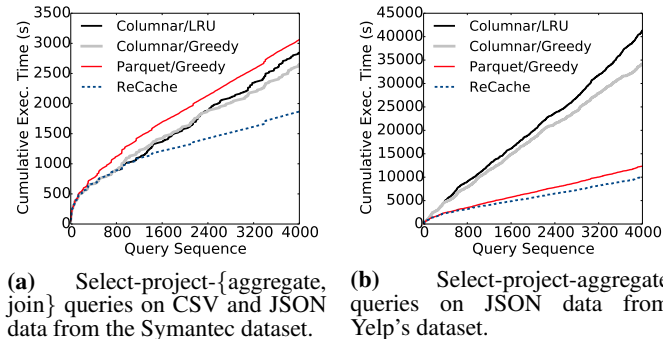
The results show that ReCache's cost-aware eviction algorithm performs better than LRU, Proteus and existing cost-based caching algorithms for every cache size (with the exception of MonetDB's algorithm and a 1 GB cache size). In particular, for a cache size of 8 GB, it reduces execution time by as much as 24% compared

to LRU. Compared to a best-case scenario where cache size is unlimited, ReCache is 60% closer to the baseline than LRU. Overall, compared to this best-case scenario, the reduction in execution time ranges from 8.7% to 66%. As an additional point of comparison, if we discount the time initially spent filling up the cache, the performance improvement over LRU ranges from 6.4% to 43%.

Compared to existing cost-based algorithms, ReCache outperforms the algorithm used in Vectorwise [37] for every cache size. However, MonetDB's algorithm [26] for cache eviction performs comparably with ReCache for most cache sizes. While MonetDB's benefit metric is based only on the frequency and weight of a cached object, its heuristic to put an upper bound on the worst-case allows it to perform comparably with ReCache. Nevertheless, on the larger 8GB cache, ReCache is able to improve performance by over 19% compared to MonetDB's algorithm.

Surprisingly, ReCache also performs better than or comparably close to the two offline algorithms. Since farthest-first does not account for object weights, it is not always able to make the optimal eviction decision. The log-optimal algorithm also provides only approximate optimality guarantees since weighted cache eviction is an NP-complete problem.

The main reason for this improvement is that ReCache tends to keep more costly data items in cache, whereas LRU does not take into account the cost of bringing in expensive data items. However, unlike Proteus, it does not assume that JSON data is always more expensive than CSV, and improves on it by basing its decisions on actual cost measurements. ReCache's behavior is particularly useful with a large cache size, since it allows more of the large, expensive objects to be kept in cache.



**(a)** Select-project-{aggregate, join} queries on CSV and JSON data from the Symantec dataset.

**(b)** Select-project-aggregate queries on JSON data from Yelp's dataset.

**Figure 15:** Cumulative execution times for a sequence of queries on two datasets.

## 6.4 Cache Performance over Diverse Workloads

Finally, we evaluate ReCache's performance on limited memory budgets over two workloads: (a) a sequence of 4000 select-project-aggregate and select-project-join queries on the Symantec dataset over both CSV and JSON data using a 24 GB cache size, and (b) a sequence of 4000 select-project-aggregate queries on the Yelp dataset using a 32 GB cache size. We run queries with four different configurations to highlight the contribution of the techniques described in this paper: (i) using a relational columnar layout and LRU eviction (Columnar/LRU), (ii) using a relational columnar layout and ReCache's cost-based eviction (Columnar/Greedy), (iii) using a Parquet-based layout and ReCache's cost-based eviction (Parquet/Greedy) and (iv) using ReCache's automatic layout selection and cost-based eviction algorithms (ReCache).

Figure 15 plots the results. Automatic layout selection in ReCache enables it to reduce execution time over the entire workload by 39% compared to a solely Parquet-based layout, and by 51% over the last 2000 queries. Furthermore, ReCache reduces execution time by 34% compared to a columnar layout with greedy cost-based caching. The reasons for this improvement in performance are three-fold. First, some cached items are best queried using the Parquet layout. Second, creating cached items in the relational columnar layout is slightly more expensive. Finally, the relational columnar layout results in larger cached items, which triggers more cache evictions and, hence, a lower cache hit rate.

On the Yelp dataset, the results for the relational columnar layout are significantly worse. Compared to Parquet, ReCache reduces execution time by 19%, but compared to the relational columnar layout, it is able to reduce execution time by over 70%. This is primarily because, on the average, the Yelp dataset associates larger collections with each record. Flattening this dataset into the relational columnar layout makes cache evictions more frequent and scans more expensive.

Lastly, we measure the benefit of cost-based eviction versus LRU on the relational columnar layout. As shown in Figure 15, the difference in performance is negligible initially, but increases over time. The eventual improvement over 4000 queries is approximately 8% for the Symantec workload and 20% for the Yelp workload. Compared to ReCache, LRU combined with the relational columnar layout leads to a 1.5-4x increase in execution time.

**Summary**. On real-world datasets with limited memory budgets, ReCache performs 19-39% faster than Parquet over the entire workload, and 22-51% faster over the second half of the workload. Compared to a relational columnar layout and an LRU-based cache eviction policy, ReCache reduces execution time by 34-75%. ReCache achieves these gains due to its cost-aware caching algorithm, and its ability to automatically adapt its data layout according to the current dataset and workload.

## 7. CONCLUSION

Reactive Cache (ReCache) is a cache-based performance accelerator for raw data analytics over heterogeneous file formats. ReCache maintains caches storing the intermediate results of database operators, achieving low caching overhead for individual queries and high caching performance over the entire query workload. ReCache automatically decides the fastest in-memory data layout for nested and relational cached data. Using heuristics informed by timing measurements and workload monitoring, ReCache maximizes cache performance by dynamically choosing between nested column-oriented layouts (like Parquet), relational column-oriented layouts and relational row-oriented layouts.

In addition, ReCache tracks the usage pattern of each cached item as well as the estimated cost of reconstructing a cached item from its origin. This enables it to make much more informed cache eviction decisions than existing eviction algorithms like LRU.

Finally, unlike most traditional forms of caching, ReCache maintains caches with varying overheads and layouts. To minimize overhead, it tracks the time consumed by caching operations and, if the measured overhead is too high, reacts quickly by switching to low-overhead caching techniques. Experiments on synthetic and real-world datasets show that ReCache reduces workload execution times by 19-75% compared to existing state-of-the-art approaches.

# 8. REFERENCES

[1] The LLVM compiler infrastructure. http://llvm.org. Accessed: 2017-01-30.

[2] The Yelp Dataset Challenge. http://yelp.com/dataset_challenge. Accessed: 2017-06-30.

[3] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[4] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 466–475. IEEE, 2007.

[5] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *EDBT*, 2013.

[6] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 241–252. ACM, 2012.

[7] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1103–1114. ACM, 2014.

[8] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 718–729. VLDB Endowment, 2003.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.

[10] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[11] S. Blanas et al. Parallel data analysis directly on scientific file formats. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014.

[12] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[13] Y. Cheng and F. Rusu. SCANRAW: A Database Meta-Operator for Parallel In-Situ Processing and Loading. *TODS*, 40(3):19:1–19:45, 2015.

[14] D. Das, J. Yan, M. Zait, S. R. Valluri, N. Vyas, R. Krishnamachari, P. Gaharwar, J. Kamp, and N. Mukherjee. Query optimization in oracle 12C database in-memory. *PVLDB*, 8(12):1770–1781, 2015.

[15] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split Query Processing in Polybase. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013.

[16] K. Dursun, C. Binnig, U. Cetintemel, and T. Kraska. Revisiting reuse in main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2017.

[17] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *TODS*, 25(4):457–516, 2000.

[18] S. Finkelstein. Common expression analysis in database applications. In *Proceedings of the 1982 ACM SIGMOD international Conference on Management of Data*, pages 235–245. ACM, 1982.

[19] A. Floratou, N. Megiddo, N. Potti, F. Özcan, U. Kale, and J. Schmitz-Hermes. Adaptive caching in Big SQL using the hdfs cache. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 321–333, New York, NY, USA, 2016. ACM.

[20] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM Sigmod Record*, 26(4):63–68, 1997.

[21] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. *ACM SIGMOD Record*, 16(3):395–398, Dec. 1987.

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

[23] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten, et al. Monetdb: Two decades of research in column-oriented database architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, 35(1):40–45, 2012.

[24] S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 701–710. ACM, 1997.

[25] M. Ivanova, M. Kersten, and S. Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *SSDBM*, 2012.

[26] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. *ACM Transactions on Database Systems (TODS)*, 35(4):24, 2010.

[27] Y. Kargın, M. Kersten, S. Manegold, and H. Pirk. The dbms-your big data sommelier. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1119–1130. IEEE, 2015.

[28] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *PVLDB*, 9(12):972–983, 2016.

[29] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 7(12):1119–1130, 2014.

[30] J. Kestelyn. Introducing Parquet: Efficient columnar storage for apache hadoop. *Cloudera Blog*, 3, 2013.

[31] Y. Kotidis and N. Roussopoulos. Dynamat: a dynamic view management system for data warehouses. In *ACM SIGMOD Record*, volume 28, pages 371–382. ACM, 1999.

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

[33] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[34] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[35] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of

web-scale datasets. *PVLDB*, 3(1-2):330–339, 2010.

[36] MySQL. Chapter 24. Writing a Custom Storage Engine. `http://dev.mysql.com/doc/internals/en/custom-engine.html`.

[37] F. Nagel, P. Boncz, and S. D. Viglas. Recycling in pipelined query evaluation. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 338–349. IEEE, 2013.

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

[39] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE Data Eng. Bull.*, 37(1):3–11, 2014.

[40] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *PVLDB*, 10(10):1106–1117, 2017.

[41] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008.

[42] H. Pirk, M. Grund, J. Krueger, U. Leser, and A. Zeier. Cache conscious data layouting for in-memory databases. *Hasso-Plattner-Institute*, 2010.

[43] P. Roy, K. Ramamritham, S. Seshadri, P. Shenoy, and S. Sudarshan. Don't trash your intermediate results, cache'em. *arXiv preprint cs/0003005*, 2000.

[44] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.

[45] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[46] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

[47] W. Zhao, Y. Cheng, and F. Rusu. Vertical partitioning for query processing over raw data. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 15. ACM, 2015.