Adaptive Cache Mode Selection for Queries over Raw Data

Tahir Azim*, Azqa Nadeem‡ and Anastasia Ailamaki*†

*École Polytechnique Fédérale de Lausanne ‡TU Delft †RAW Labs SA
tahir.azim@epfl.ch, A.Nadeem@student.tudelft.nl, anastasia.ailamaki@epfl.ch

ABSTRACT

Caching the results of intermediate query results for future re-use is a common technique for improving the performance of analytics over raw data sources. An important design choice in this regard is whether to lazily cache only the offsets of satisfying tuples, or to eagerly cache the entire tuples. Lazily cached offsets have the benefit of smaller memory requirement and lower initial caching overhead, but they are much more expensive to reuse.

In this paper, we explore this tradeoff and show that neither lazy nor the eager caching mode is optimal for all situations. Instead, the ideal caching mode depends on the workload, the dataset and the cache size. We further show that choosing the sub-optimal caching mode can result in a performance penalty of over 200%. We solve this problem using an adaptive online approach that uses information about query history, cache behavior and cache size to choose the optimal caching mode automatically. Experiments on TPC-H based workloads show that our approach enables execution time to differ by, at most, 16% from the optimal caching mode, and by just 4% on the average.

1. INTRODUCTION

As the data deluge continues, it has become increasingly important to be able to run analytics queries directly over raw data sources. The key challenge in achieving this goal is the high CPU and IO cost of reading and parsing raw data [2]. In contrast, loading raw data into a database first has a very high initial cost, but enables queries to run much more efficiently afterwards.

Data analytics systems commonly use on-the-fly caching of previously parsed data and intermediate operator results to address this challenge [2, 12]. 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.

Reading and parsing raw data is a costly operation, so creating in-memory caches of binary data can add significant overhead to a query. The alternative approach is to cache in memory only the file offsets of satisfying tuples, and reconstruct the tuples from those offsets when needed. Compared to the *eager* approach of caching entire tuples, this *lazy caching* approach has the benefit of lower caching overhead, but is more expensive to reuse.

Existing systems generally provide support for statically choosing one of these caching modes, but leave open the problem of dynamically handling this tradeoff. Recent work on reactive caching presented a technique to quickly switch to lazy caching if the observed overhead of caching entire tuples was too high [4]. However, if a lazily cached item is reused, it immediately converts it to an eager one, in order to make future reuse more efficient. Thus, it also assumes that a popular item must be cached in eager mode.

This paper further explores the tradeoff between lazy and eager caching modes, and shows that neither mode is optimal for all cases. Instead, the ideal caching mode depends on the size of the cache, the characteristics of the dataset, and properties of the workload such as query types and selectivity. Building on this result, we propose an adaptive approach to discover and use the optimal caching mode automatically at runtime.

In particular, this paper makes the following contributions:

- We experimentally demonstrate that neither lazy nor eager caching mode is ideal for all scenarios. Our experiments show that choosing the incorrect caching mode can increase workload execution time by over 200% compared to the optimal.
- We propose Acme, an adaptive cache mode selection algorithm for automatically determining the optimal caching mode at runtime, without relying on any advance information about the workload or dataset. Acme reduces the maximum performance penalty for choosing the incorrect caching mode to less than 16%, and the average penalty to just 4%.

The next section overviews related work in this domain. Section 3 presents experimental results showing that neither eager nor lazy caching is the best approach for all scenarios. Section 4 then describes and evaluates our algorithm for automatically selecting the ideal cache mode. Finally, we discuss some alternative design choices that did not succeed in meeting our design goals.

2. RELATED WORK

Caching is a well-studied problem in computer systems. In the area of databases, a large body of work exists on the problem of caching and reusing the results of previous query executions to improve performance.

Caching Disk Pages. All database systems retain disk pages in memory buffers for a period of time after they are read in from disk. 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 [9] proposed caching only those disk pages that are re-used within 5 minutes. A revised version of the paper ten years later found the results to be mostly unchanged [8].

Cost-based Caching. Cost-based caching algorithms, such as online Greedy-Dual algorithms [20, 10, 5], improve cache performance when the cost of reading different data items can vary widely. These algorithms prioritize evicting less costly items in order to keep expensive items in the cache. In addition, these algorithms account for recency of access: recently accessed items are less likely to be evicted than those accessed in the more distant past.

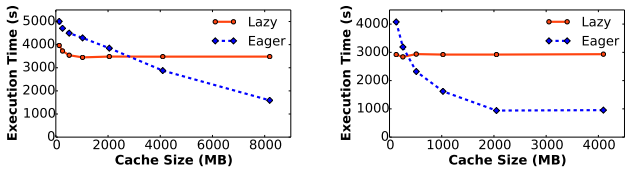
Caching Intermediate Query Results. Automatically caching intermediate results of query execution has been frequently studied [18, 13] but it has not been widely deployed due to its potential overhead and additional complexity. Recent work towards caching intermediate results of queries over relational data [15, 11] has further shown the benefit of this approach. This work uses a cost-based approach to decide which tuples to keep in the cache and which to remove. The cost of each cached data item is estimated based on fine-grained timing measurements and the size of the item.

Querying Raw Data. Many systems address the need to perform queries over data stored in their original raw formats. The traditional approach of completely loading raw data into a DBMS before running queries is a major bottleneck for use cases where immediately getting fast response time is important. Apart from full loading, DBMS offer an *external tables* functionality (e.g.,using a custom storage engine [14]), in which every query parses the entire raw file from scratch. Hadoop-oriented solutions [19, 17, 3] rely on adapters per file format. For example, when a dataset is stored in CSV files, a CSV-specific adaptor converts all values into a binary representation during query execution. Thus the query pays for the conversion of the entire file, even if it never accesses parts of the file. Other systems [7, 1] operate in a "hybrid" mode, loading data on-demand into a DBMS based on workload patterns, or when there are free CPU cycles [6]. Finally, several systems process raw data *in situ*, and use techniques such as index structures, caches and custom code-generated access paths to mask the costs of repeatedly accessing raw data [2, 12, 16].

Caching Raw Data. The NoDB [2] and Proteus [12] query engines for raw heterogeneous data both use caches to speed up query performance. While caches enable them to optimize query response time, their caching policies are relatively ad-hoc and simplistic. For example, they admit everything into the cache and use LRU for eviction, with the exception that they assume cached JSON items to always be costlier than CSV. ReCache [4] adds to this body of work by taking into account the cost of reusing a cached result and automatically choosing the fastest data layout for the cache. In addition, it caches only satisfying offsets of an intermediate query result if the overhead of caching entire tuples is too high. However, it assumes that popular data items must always be cached eagerly, even if the working set is large or the cache size is small. Proteus and ReCache also introduce the term *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. For brevity, we use the terms lazy and eager caching in this paper instead of low and high degree of eagerness respectively.

3. FINDING THE OPTIMAL CACHE MODE

We first test experimentally whether either one of lazy or eager caching can provide optimal performance in all cases. All experiments are conducted on an Intel Xeon E5-2660 processor with a clock speed of 2.20 Ghz and 8 cores per socket. The machine has



(a) Select-project-{aggregate, join} queries on CSV data.

(b) Select-project-aggregate queries on JSON data.

Figure 1: Workload execution time using lazy and eager caching modes on TPC-H datasets in CSV and JSON formats.

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. All experiments in this paper are built on top of ReCache [4], which uses LLVM 3.4 to generate custom code for each query. We run all experiments in single-threaded mode over warm operating system caches. We disable the use of any indexing structures (such as positional maps [2]), and cache the outputs of selection operators in each experiment. The caching system has support for operator subsumption, i.e. if the cache contains a superset of the tuples that would be returned by an operator, then the operator will scan these cached tuples.

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). 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 results on TPC-H CSV data are shown in Figure 1a. We find that for smaller cache sizes below 2 GB, using lazy caching enables 24-27% lower execution times. For cache sizes larger than 2 GB, eager caching performs better. These results seem quite intuitive in retrospect: since fully cached tuples are larger than tuple offsets, eagerly cached results fill up small caches more quickly and, therefore, get evicted more frequently. As a result, they are less likely to be available for reuse in future queries. Lazily cached results may be less efficient to reuse, but result in more frequent cache hits.

Query workloads over JSON data yield similar results. Our experiment uses a 2.5 GB JSON 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 100 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 this JSON data, eager caching involves unnesting [12] the hierarchical data in order to convert it into a relational layout. This obviously results in some data duplication, since multiple tuples with the same order ID may be associated with different lineitem records. In case of eager caching, each of these tuples is stored in memory. In case of lazy caching, if a single line of JSON generates multiple tuples all satisfying a predicate, then the file offset of the line is only stored once.

Figure 1b shows the results. With a cache size of 128 MB, eager caching results in a performance penalty of almost 40% compared to lazy caching. Eager caching only starts out-performing the lazy approach when the cache size exceeds 512 MB. At 2 GB, the working set for the workload completely fits in memory, so further increasing cache size does not improve performance. This threshold basically depends on the size of the working set, which in turn depends on the nature of the workload and the dataset. For a DBA with no *a priori* workload knowledge, estimating this threshold is difficult, and incorrectly choosing the caching mode can increase execution time by over 200% (e.g. with a cache size of 2 GB).

We conclude that neither lazy nor eager caching is ideal in all cases. The next section addresses this problem using an automatic, online method for adapting to the correct caching mode based on historical knowledge of the workload and dataset.

4. ADAPTIVE CACHE MODE SELECTION

The experimental results in Section 3 show the importance of choosing the correct caching mode for achieving optimal performance. This is a difficult task because the workload may not be known beforehand and the properties of the dataset may be computationally expensive to ascertain. For this reason, we propose an adaptive online algorithm that can observe the input workload and the behavior of the cache to automatically choose the optimal caching mode.

4.1 Design and Implementation

The mode selection algorithm is based on the idea that given a known cache size, the system can continue executing in one cache mode, while maintaining a reasonable estimate of how the alternative would perform. The algorithm works by tracking a window of past queries, simulating a cache operating in the alternative caching mode, and statistically determining if there would be a significant improvement in performance using the alternative mode.

By default, the **A**daptive **c**ache **m**od**e** selection (Acme) algorithm starts off caching in eager mode. For each intermediate result cached using the current caching mode, Acme also creates a "shadow" entry for a hypothetically cached item in the alternative mode. This shadow entry does not store any data from the intermediate result. Instead, it only stores a pointer to the actual cached item, whose metadata it uses to estimate its own size, creation cost, and scanning cost.

After each query, Acme looks at the state of both the actual and shadow caches to decide if either one has exceeded its memory budget and requires evictions. The cache eviction policy is based on ReCache [4], which uses a cost-based algorithm to decide which cached item to evict. The algorithm relies on the following measurements to make this decision:

1. n, how many times an operator's cache has been reused.
2. t, the time spent executing the operator.
3. c, the time spent caching the results of the operator in memory.
4. s, the time spent scanning the in-memory cache when it is reused.
5. l, the time spent looking for a matching operator cache
6. S, the size of an operator cache in bytes

Based on these measurements, Acme computes a benefit metric, $b = n*(t+c-s-l)/log(S)$, for each item and feeds it to a Greedy-Dual algorithm [10, 5] to make eviction decisions. In addition to running this algorithm on the actual cache, it also runs the algorithm on the shadow cache in order to track the evolution of the shadow cache over time. Thus, after any query, Acme knows what the contents of the cache would be, if it were to use the alternative caching mode.

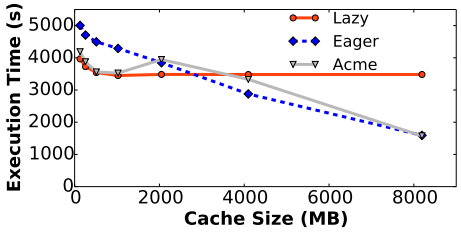
An important detail here is how to estimate the values of c, s and S for items in the shadow cache. Since data in the shadow cache was never actually materialized, we do not have actual measurements for these parameters. Therefore, Acme uses estimates for these parameters.

If the shadow cache is simulating lazily cached entries, it assumes $c \approx 0$, $S = sizeof(int)*i$ and $s = t_{eager}*i/N$, where t_{eager} is the value for t for the corresponding eagerly cached entry, i is the number of tuples in the cached item, and N is the total number of tuples in the cached item's original relation. These estimates reflect the fact that the cost of creating a lazy cache is very small, while its size and scan cost are both directly proportional to its size relative to its original relation.

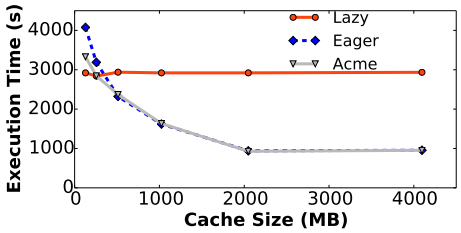
If the shadow cache is simulating eagerly cached entries, Acme assumes $c = k*c_{lazy}$, $s \approx 0$ and $S = i*S_{avg}$, where k is a constant greater than one, c_{lazy} is the time taken to create the corresponding lazy cache and S_{avg} is the average tuple size measured in the original relation. The idea behind these estimates is that the cost of creating an eager cache is substantially higher than a lazy cache. Based on the results in [4], which showed the average cost of eager caching to be at least 3 times as high as that of lazy caching, we use $k = 3$. In addition, the cost of scanning it is very low, while its size depends on both the number and sizes of its individual tuples.

After executing its eviction decisions, Acme computes for both the actual and shadow caches how much their content is (or could have been) contributing towards the efficiency of the entire system. This is computed by summing up the contributions of every item stored in the cache. For each item, the contribution is computed as $(t+c-s-l)$, which represents the cost that would be incurred if the item had to be reconstructed from its original data sources, minus the cost of reusing the cache. While in lazy mode, it is possible that both the lazy actual cache and the shadow eager cache remain under the size budget. In this scenario, it may be preferable to switch to the more efficient eager caching mode. Acme incorporates this into the cost model by multiplying each contribution by $\frac{MaximumCacheSize}{CurrentCacheSize}$ while in lazy caching mode.

Acme appends the sums of contributions for the actual and shadow cache into two separate lists. It then runs a statistical test of significance on the two lists to check if the shadow cache has significantly higher contributions. If the test returns true, Acme switches to caching data in the alternative mode for future queries. Meanwhile, Acme does not evict or modify existing items in the cache, resulting in zero switching overhead and allowing these items to be reused for future queries.



(a) Select-project-{aggregate, join} queries on CSV.



(b) Select-project-aggregate queries on JSON.

Figure 2: Workload execution time using lazy, eager and adaptive caching modes on TPC-H datasets in CSV and JSON formats.

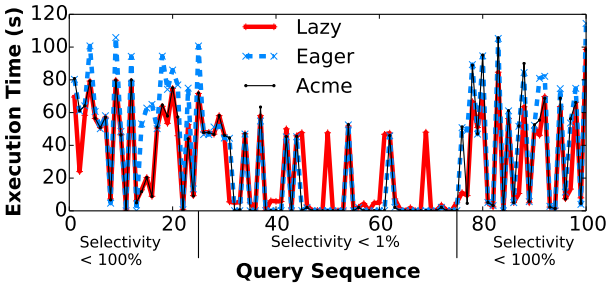


Figure 3: Execution times for a sequence of queries on CSV data using adaptive, lazy and eager caching modes. Between queries 25 and 75, selectivity $< 1\%$. Otherwise, it ranges from 0 to 100%.

When it switches the caching mode, Acme empties the contribution lists and the shadow cache because their contents now correspond to an old configuration. Furthermore, it only sums up contributions from those items in the actual cache whose mode matches the current configuration. It will now revert back to the previous configuration only if the shadow cache's benefit once again starts exceeding that of the actual cache by a significant margin.

4.2 Evaluation

We evaluate our approach on the same dataset and workload as Section 3 to test if it can adaptively select the correct caching mode. For testing statistically significant differences between the two modes, we use the one-tailed Student's t-test with a significance level of 0.05. We choose this test because it is designed to work with a small sample size, so it allows us to make a mode switching decision more confidently using very few samples.

Figure 2 shows the results. In both the CSV and JSON workloads, Acme is able to closely match the performance of the optimal caching mode. The average difference from the optimal over all data points is less than 4%, while the worst case difference is less than 16% (on the CSV workload with a cache size of 4096 MB). In the cases where the workload and the cache size necessitate a switch to lazy caching, the t-test is able to make this decision within the first 13 queries in the worst case, and within the first 3 queries in the best case.

Acme primarily under-performs when the difference in performance between eager and lazy modes is relatively small. In that case, its statistical test may not deem the difference between the two modes to be significant enough to switch. As a result, it may continue using the marginally under-performing cache mode.

The results also show that the Acme's automatic cache mode selection algorithm has almost negligible overhead. In every case where the cache mode does not switch to lazy mode, Acme's performance closely matches that of eager mode. We further confirmed this observation by directly measuring the additional estimation and significance testing overhead for each workload. Even in the worst case, the overhead accounted for less than 1 second of the entire workload execution time.

Finally, we demonstrate Acme's ability to adapt to a changing workload in Figure 3. We fix the cache size at 256 MB and run the CSV workload. However, between queries 25 and 75, the workload switches its random selectivity from less than 100% to less than 1%. This allows eager caching to completely fit intermediate results in the cache. Acme adapts to the changing workload by switching to lazy caching on query 3, eager caching on query 29, and finally back to lazy caching on query 90. This adaptivity is reflected in the figure where Acme tracks closely with eager caching between queries 30 and 90, and with lazy caching otherwise.

5. DISCUSSION

We now briefly discuss the applicability of this work to traditional database systems that operate on relational data. We then describe some approaches we tried initially that did not meet our design and performance goals. We also outline possible directions for future research.

5.1 Applicability to Traditional DBMS

While this paper has focused on caching results of queries over raw data, Acme is also applicable to traditional databases storing relational data in row-oriented and columnar layouts. Row-oriented databases are essentially similar to CSV files, differing mainly in that data does not need to be parsed from plain text. Moreover, a file offset alone may not be sufficient to identify a tuple. These details can be easily incorporated into Acme's cost model.

With columnar databases, an additional difference is that an eager cache will likely contain only a small subset of attributes of a relation. Furthermore, to reconstruct a tuple from its columns using a lazy cache, an offset has to be stored for each column. In this case, the lazy cache would only be beneficial if (i) the total size of the columns was much larger than the total size of the integer offsets, or (ii) if reading the columns was very expensive. Nevertheless, integrating these differences into the cost model should be quite straightforward.

5.2 Alternative Designs

Before settling on our final design, we considered a number of initial ideas which we found to be incompatible with our design goals. As a first step, we considered a purely analytical, offline method to find the optimal cache mode. But this required prior knowledge of the workload and dataset.

Second, we implemented and evaluated a design where the cache stored satisfying tuples in both lazy and eager modes. When an eviction was required, the cache only evicted a fraction of the eagerly cached tuples to reclaim space. When reused, a cached item initially scans the full tuples in memory and then scans the satisfying lines in the original data source. While we attempted a number of cache eviction heuristics to make this work well, the primary

problem remained the same: the initial eager caching overhead was too high, both in terms of execution time and memory consumption. As a result, cache evictions and misses remained frequent, and its performance degenerated to that of eager caching.

A third, similar approach did not keep the lazy offsets in cache. Instead, after partially evicting some tuples out of an eager cache to meet the cache size limit, it only kept the file offset of the last tuple that remained. This allowed the cache to scan the cached item, and then partially scan the original data source. This approach also degenerated in performance to eager caching in every case, hence it was also unsuitable for small cache sizes.

Finally, we considered the case where the shadow cache also stored the data associated with each cached item. In this way, the system could switch caching modes faster, since the data for each shadow cache item would already be in memory. We did not explore this in more detail because it was also quite similar to our first attempt, so the space/time overhead of caching in two modes was likely to be excessive. However, a more detailed evaluation of this and other techniques might help uncover a better approach.

5.3 Future Work

There are many possible avenues for future work. In particular, we want to explore how Acme can be adapted for use in other real-world systems, including open source relational database engines and in-memory databases. For in-memory databases, it may be interesting to consider the memory hierarchy at a finer granularity by distinguishing between L1, L2 and L3 caches in the model. Based on advance knowledge of the workload and performance requirements, this approach could also be used for cache sizing. Finally, it will be valuable to evaluate the benefits and limitations of Acme on additional real-world workloads and benchmarks.

6. CONCLUSION

Caching the results and intermediate results of queries is a commonly used technique for accelerating database queries. An important design question while building the cache is whether it should store only the offsets of satisfying tuples, or the entire tuples. Manually choosing this caching mode is a challenge because the optimal mode depends on the cache size as well as the properties of the workload and the dataset. An incorrect choice can cause performance to be over 200% worse than the optimal.

This paper presents Acme, an adaptive cache mode selection algorithm, which observes the evolution of the cache under the existing workload to dynamically determine the optimal caching mode. This reduces the maximum performance penalty for choosing the incorrect caching mode to just 16% and the average penalty to 4%.

Acknowledgments. This work was funded in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No 650003 (Human Brain project) and by the European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement no 617508 (ViDa).

7. REFERENCES

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

[2] 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*.

[3] 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.

[4] T. Azim, M. Karpathiotakis, and A. Ailamaki. ReCache: Reactive caching for fast analytics over heterogeneous data. *Proceedings of the VLDB Endowment*, 11(3), 2017.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

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

[16] 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.

[17] 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.

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

[19] 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.

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