CIAO: An Optimization Framework for Client-Assisted Data Loading

Cong Ding[§]
Peking University
congding@pku.edu.cn

Dixin Tang, Xi Liang, Aaron J. Elmore, Sanjay Krishnan
The University of Chicago
{totemtang@, xiliang@, aelmore@cs., skr@}uchicago.edu

***Abstract*—Data loading has been one of the most common performance bottlenecks for many big data applications, especially when they are running on inefficient human-readable formats, such as JSON or CSV. Parsing, validating, integrity checking and data structure maintenance are all computationally expensive steps in loading these formats. Regardless of these costs, many records may be filtered later during query evaluation due to highly selective predicates – resulting in wasted computation. Meanwhile, the computing power of client ends is typically not exploited. Here, we explore investing limited cycles of clients on prefiltering to accelerate data loading and enable data skipping for query execution. In this paper, we present CIAO, a tunable system to enable client cooperation with the server to enable efficient partial loading and data skipping for a given workload. We proposed an efficient algorithm that would select a near-optimal predicate set to push down within a given budget. Our experiments show that CIAO can significantly accelerate the data loading processing and query execution.**

I. Introduction

Databases often centralize data collected from multiple, distributed client systems. For example, a single log server in a data center may collect `syslog` events from all other servers [1]. Or, a time-series database may collect environment sensor readings from sensors placed around a building [2]. As such deployments scale up, data loading (i.e., parsing, validating, and storing the client data) becomes an under-appreciated bottleneck on the database server due to the computation intensive tasks of parsing input formats, converting types, and validating input [3], [4]. This bottleneck delays the results of downstream analytics queries, and can increase the latency of any decision-making system that consumes those results. Database systems eagerly ingest and load data, as there is no mechanism for the system to determine if the data will be relevant and needed for likely future queries.

Prior research literature describes three main approaches to relieve pressure on the database: client-side parsing [5], raw-format query processing [6], [7], and hardware acceleration [4], [8]–[11]. In client-side parsing, we leverage excess processing capacity on the client devices to parse and serialize the data before ingestion on the central server. Client-side parsing reduces the data loading burden on the server and the network transfer between the server and the client. The main disadvantage of the client-side parsing is that it requires relatively capable and powerful client devices to implement—if the clients are too under-powered it can actually hurt the overall per-record loading latency. An alternative is to simply avoid data loading on the server when possible, and directly process queries over the raw-format data. While this approach avoids assumptions about the client and relieves the data loading bottleneck, it often results in suboptimal downstream query processing. Structured formats, like columnar storage, may have an upfront loading cost but greatly improve downstream query latency especially in comparison with a row-oriented raw data format. Regardless, raw processing still requires expensive ingestion when a record or attribute is required for a query's predicate evaluation.

Clearly, there is a careful balancing act between client-side processing, data loading costs, and downstream query processing. However, to an administrator, the only metric that matters is the per-record processing time: the time from when an event happens to when it is reflected in the query result. The various factors combine in complex, deployment-specific ways to result in a final per-record processing time. Today's approaches pick one point in this design space and don't give the user enough flexibility to reason about the trade-offs in a variety of hardware settings. Client-side parsing may lead to overall worse performance when the clients are under-powered, and raw-format query processing may lead to worse performance if there are repeated queries over the same data.

We present CIAO, an optimization framework that can determine what processing to do on a client given a computation budget to maximally benefit downstream query processing. CIAO identifies a set of predicates that can be applied directly by the client using simple string pattern matching, and selects the set of predicates for a client to evaluate using a client's slackness via a specified time limit. Clients evaluate these predicates and include lightweight bitvectors to indicate what records satisfy what predicates. The server then selectively loads records that satisfy at least one predicate, and sets aside the other raw data to be loaded when needed (e.g. just-in-time loading). For records that are loaded into the internal format. CIAO retains the bitvectors to use for data-skipping [12]. CIAO is developed as part of a project on resource-efficient database systems, CrocodileDB [13], to explore how to improve resource utilization in the data loading process.

The key architectural insight is a marriage between raw-format query processing and client-side parsing: the client

[§]Work done at ChiData group of the University of Chicago.

2375-026X/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00187

devices directly manipulate the raw data without fully parsing it. We leverage techniques similar to Sparser [6] and UDP [4] to directly apply popular filters to raw data records. However, instead of evaluating predicates on the raw JSON data at the server, which requires complex changes to the execution engine, we use simple filtering on client-side that respects a computation budget. These filters give us annotations that can be used for "partial data loading" on the server, where only the most relevant data is eagerly loaded. The filters also facilitate data skipping when the database is queried.

The core optimization problem in CIAO is to select a subset of predicates to be pushed down with respect to a computation budget on the client side. We prove that this is a *submodular* problem, that is, it has diminishing marginal returns. We leverage algorithms from the submodular optimization literature to appropriately select what computation to do on the client with optimality guarantees. Experimentally, we show a trade-off between the client's budget and the downstream server loading and query processing savings. We implemented this system using popular data systems components and formats (e.g., JSON, Parquet, and Spark) and evaluated its performance on three real-world datasets. Our experimental results show that the system substantially accelerates data loading by up to 23x and query execution by up to 21x and improves end-to-end performance by up to 19x.

This paper is an abridged version of a full technical report[1] that includes additional technical details and experiments.

II. Related Work

We now discuss the related research projects on fast data parsing and ingestion, and in-situ query processing with lazy data loading. We note that none of these projects considers pushing predicates to the clients to reduce the data loading cost and accelerate query processing.

Our prior study [13] and others [3] show that data loading is a time-consuming process, especially for text-based data formats (e.g. JSON or CSV). Many research projects consider accelerating the data loading process by exploiting the modern hardware. Instant loading [11] leverages SIMD instructions to accelerate parsing CSV files and interleaves the index creation with parsing data to make data quickly available. Several other projects [8], [9] exploit SIMD to quickly parse JSON files or general text-based data formats in a distributed setting. UDP [4] builds a programmable accelerator and offloads the data loading process to the hardware accelerator.

CIAO is different from these projects in that none of them considers leveraging the computing power of the clients to accelerate the data loading process, and they do not consider partial loading to make data quickly available for queries.

Many projects consider not loading the data upfront, but directly queries data in its raw format (e.g. CSV or JSON) and gradually loads data while processing queries. NoDB, and later RawDB, [7] executes queries over CSV files directly and builds light-weight indices to accelerate future queries [14].

[1]The full report can be found at http://arxiv.org/abs/2102.11793

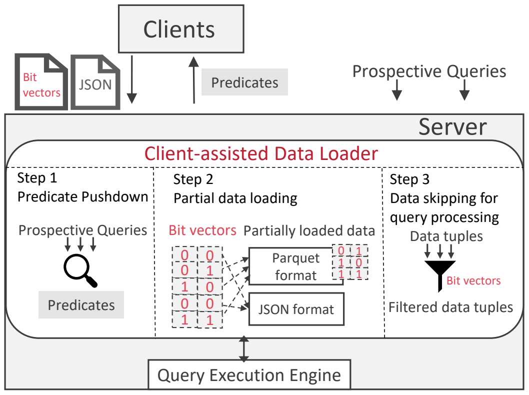

Fig. 1. An overview of client-assisted data loading

Later work explored parallel in-situ query execution over scientific formats [15]. Invisible loading [16] piggybacks the data loading process with MapReduce [17] jobs that analyze the raw data. Invisible loading leverages MapReduce jobs' parsing and tuple extraction operations to incrementally load tuples into a database system. Database cracking [18] builds indices incrementally when the underlying data is queried. Data skipping projects [12] consider building coarse-grained index to skip irrelevant data.

The difference between CIAO and these projects is that CIAO considers leveraging the computing capacity of the data clients to build light-weight index to enable partial loading and accelerate query processing.

III. Overview and Assumptions

We now give an overview of CIAO and show how it leverages the data clients' computing power to accelerate data loading and query processing. Specifically, CIAO pushes predicates of prospective queries to the data clients (e.g. edge sensors). Based on a computation budget clients will evaluate simple predicates on the data before sending them to the server and generate bit-vectors that indicate whether a tuple is valid for a predicate. The bit-vectors are sent along with the raw data to the server. After, CIAO utilizes the bit-vectors to selectively load the raw data format (e.g. JSON) from the clients into a binary data format that is more amenable to query (e.g. Parquet). Finally, when processing a query, the bit-vectors are used to skip irrelevant tuples that do not belong to the query.

Fig. 1 shows an overview of CIAO. The first step is `predicate pushdown`. Choosing the predicates to push down systematically considers two factors: how many new tuples a predicate can filter out for the prospective queries (i.e. the new tuple is marked as not valid for a predicate) and the increased cost of evaluating this predicate on the client-side. Therefore, this step takes the following information as the input to decide the predicates to be pushed down: 1) the frequencies of queries that are expected to be executed; 2) the selectivity of each predicate in the prospective queries; 3) the cost of evaluating a predicate on the client-side; and 4) a *computation budget* that we allow on the client-side to evaluate the predicates we choose to push down. Here the computation

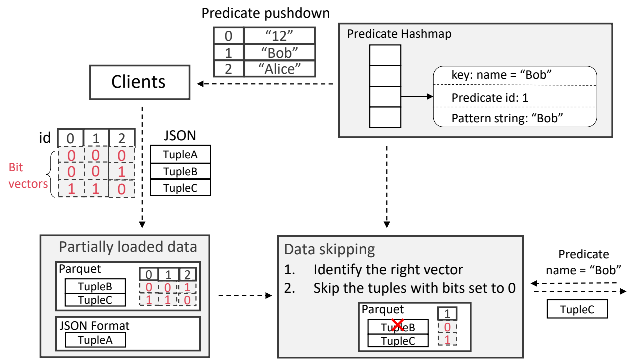

Fig. 2. An example of the workflow of CIAO

budget is specified by the database administrator and is defined as the average amount of computation cost of evaluating predicates for each new tuple. We estimate the frequencies of prospective queries and selectivities of predicates based on historical statistics and develop a cost model to estimate the cost of evaluating a predicate.

Given this information, we use a greedy algorithm to decide the predicates to be pushed down to the client-side with respect to the computation budget. This algorithm is optimized to select the predicates that filter out the most tuples for each unit of increased computation cost on the client-side, which is discussed in Sec. V. In this paper, we assume that the data clients generate JSON objects and use string operations (e.g. check whether a JSON object contains a substring) to evaluate the predicates. Sec. IV discusses the predicates we support. We further assume that data clients send JSON objects in chunks (e.g. 1k objects for each chunk). Each *JSON chunk* is associated with a set of bit-vectors, where each bit-vector corresponds to a predicate. As shown in Step 2 of Fig. 1, a bit 0/1 means the tuple is invalid/valid to a predicate.

When CIAO receives the JSON chunks from the clients, we selectively load parts of the JSON objects (i.e. data tuples) into Parquet files, which is shown as Step 2 in Fig. 1. Specifically, we choose to load a JSON object if it is at least valid to one predicate, that is, the JSON object's bit is marked as valid for at least one bit-vector. The rationale here is that we load the data that is likely to be accessed by prospective queries. Therefore, a JSON chunk is split into two parts: one is loaded into the Parquet format that is available for querying and the other is left in a raw JSON format, which requires later parsing and conversion to analyze the unprocessed records. Fig. 2 shows an example that `TupleB` and `TupleC` are converted into Parquet format, but `TupleA` is left as JSON format. The parquet file is also associated with a set of new bit-vectors that are derived from the bit-vectors of the original JSON chunk and represent whether a tuple is valid for each predicate.

When we load a JSON object into a Parquet file, we store the bit-vector information of this object into the metadata of each data block of the file. During query processing, we leverage these bit-vectors to accelerate execution (i.e. Step 3 in Fig. 1). When we scan data blocks of the Parquet file, we extract the bit-vectors that belong to the predicates included in the query, take the intersected bit-vector of them (i.e.due to the conjunctive predicates), and use the bit-vector to skip data tuples. For each tuple from a data block, if the corresponding bit of the intersected bit-vector is 0, we discard it. Otherwise, we output it to the query process engine. For example, Fig. 2 shows a query with a predicate `name = ``Bob''`. We find its predicate id in the predicate hashmap (i.e. id = 1) and filter out the tuples whose corresponding bits are set to 0 (i.e. discarding `TupleA` and `TupleB`). When the query does not include a predicate that we have pushed down, CIAO scans both Parquet and JSON files to return all data tuples.

IV. Client-side Predicate Evaluation

A core contribution of our framework is to *evaluate query predicates on client-devices without full parsing*. We argue that this design decision achieves the best of both worlds: it reduces data loading costs on the server, while not shifting the parsing burden to the clients.

Data acquired from client systems are often generated in a string-based raw-data format like delimited files and JSON (JavaScript Object Notation). These formats are highly general as they can represent many different data types and both flat and nested structures. However, this generality means that the downstream parser has to expend additional computation for parsing and validation to support features the user may not use (e.g., like parsing escape characters). Our twist on this problem is to consider a client-server extension to this basic idea of raw-format query processing to facilitate both data-skipping on the server and partial data loading to avoid loading irrelevant data. For simplicity, we assume that the client-side generates data tuples in the JSON format. We note our solution can also be applied to other text-based data formats, like CSV.

Since JSON objects are represented as strings, a limited number of SQL predicates can be evaluated as string search operations. Our client-side framework converts supported SQL predicates into string-based pattern expressions that can quickly identify satisfying JSON objects. We currently support the following types of predicates and the corresponding examples are shown in Table I

- **Exact or Substring Match** The first two examples in Table I show the exact match and substring match respectively. For the exact match, the pattern string is the operand string (e.g. `` ``Bob'' ``) that compares against the value of a key. For the substring match, the pattern string is the substring we need to find (e.g. `` ``delicious'' ``).
- **Key-presence match** For key-presence match, the pattern string is the key string (e.g. the `` ``email'' `` of the third row in Table I).
- **Key-value match** For key-value match, we have two pattern strings: the key string and the value string. The client first searches for the key string and if the key string exists, the client will continue from the current string position to search for the next key-value delimiter (i.e. a comma `` ``,'' ``). Between the key string and delimiter, the client will check whether the value string exists.

TABLE I
EXAMPLES FOR SUPPORTED PREDICATES AND PATTERN STRINGS

Supported Predicates	Example	Pattern String
Exact String Match	name = “Bob”	“Bob”
Substring Match	text LIKE “%delicious%”	“delicious”
Key-Presence Match	email != NULL	“email”
Key-Value Match	age = 10	“age” “10”

Client-side filtering is a sort of converse to data-prefetching—as it is a bet that the server can make to hide latency from the end-user. And, similar to pre-fetching, we engineer client-side filtering to allow for false-positive cases, that is, a JSON object that is actually not valid for a predicate can be marked as valid.

Consider the example of exact match in Table I. The pattern string `''Bob''` can exist in the key-value pair that does not include `''name''` as the key. Therefore, when a query scans the filtered tuples based on the bit-vectors, it needs to evaluate all predicates in this query to verify that a tuple is actually valid to this query. However, false-negative cases will never happen in our predicate evaluation, that is, if we cannot find the pattern strings in a JSON object, this JSON object is not valid to the corresponding predicate.

False-negative cases are not allowed in the predicate evaluation because they discard the JSON objects that should have been incorporated into query results. Therefore, range-based predicates or inequality predicates are not supported. In addition, if there are different data representations for the same number (`2.4` vs. `24e-1`), we do not support the number equality because it will result in false negative cases. These limitations also exist in other systems that evaluate predicates on Raw JSON objects (e.g. Sparser [6]).

V. Predicate Selection Optimization

Each predicate evaluated on the client incurs a cost. Since client-devices are often under-powered compared to the server, these costs can be significant. In this section, we describe an optimization algorithm for selecting which predicates to push down to clients. We can formulate this problem as a submodular maximization problem. Such problems are, not surprisingly, the class of problems with diminishing marginal returns, and are relevant because each additional predicate that is pushed down has diminishing returns due to overlaps. Such ideas have been leveraged in data management optimization problems in a number of prior works [19], [20].

We first set up the optimization problem as follows.

Consider a workload of m queries $Q = \{q_1, q_2, \cdots, q_m\}$, where each query has a predicate that is a *conjunction of disjunctive clauses*. Thus, for every query in the workload we have a set of candidate predicates to pushdown: for q_i the set of clauses (predicates) $P_i = \{p_1, p_2, \cdots, p_{n_i}\}$ that can be evaluated on the client. If a conjunctive predicate includes a disjunctive clause that we cannot support on the client, we do not consider it as a candidate to push down.

We further define some valuable notation that will help us formalize the optimization problem. Let the selectivity of a predicate p be denoted as $sel(p)$ and the cost of evaluating p for a JSON object as $cost(p)$. Let $freq(q)$ denote an estimate of the relative frequency that q is evaluated. We further assume that the computation budget on the client-side is fixed cost B, where B represents the average units of computation cost of evaluating predicates for each new tuple.

Let S be the set of predicates that we chose to evaluate on the client. For each query q_i, let $S_i = (P_i \cap S)$ be the set of the query's conjunctive clauses that have been pushed down. We can evaluate the probability of filtering a JSON object given the selected predicate set S is using a statistical independence assumption and the selectivity of each predicate:

$$f(q_i, S) = 1 - \prod_{p_j \in (S_i)} sel(p_j)$$

The optimization goal here is to maximize the *expected benefit* of predicate pushdown for all queries in Q:

$$f(S) = \sum_{q \in Q} f(q, S) \cdot freq(q).$$

We want to optimize this quantity while ensuring that $\sum_{p_i \in S} cost(p_i) \leq B$. In our experiments and the rest of the text, we present results with a uniform query frequency (though not necessarily a uniform predicate frequency). Formally, the optimization problem is defined as:

$$\begin{aligned} \text{maximize} \quad & f(S) \\ \text{subject to} \quad & \sum_{p_i \in S} cost(p_i) \leq B \end{aligned}$$

It can be proved that $f(S)$ is a submodular set function and based on the submodularity we use an approximation algorithm to solve the optimization problem. The prior work [21] shows that both naive greedy algorithm and greedy algorithm based on benefit-cost ratio can perform arbitrary badly with respect to the optimal solution. Fortunately, the better solution of the two algorithms has a bounded error with respect to the optimal solution. Specifically, we run the two algorithms separately and choose the one with the higher $f(S)$. One prior study [22] proves that this solution is no smaller than $\frac{1}{2}(1 - \frac{1}{e})OPT \approx 0.316 \times OPT$ where e is the mathematical constant and OPT is the $f(S)$ of the optimal solution.

VI. Experiment Results

A. Prototype implementation

We implement the predicate selection algorithm in Python 3, and the components of evaluating predicates on the client-side and partial data loading in C++. We use the `string::find` method of C++ STL for substring matching and choose rapidJSON[2] as our JSON parser. We build parquet files with the low-level interfaces provided by the Apache Arrow C++

[2]RapidJSON, http://rapidjson.org/

TABLE II
WORKLOADS FOR END-TO-END EXPERIMENTS

Workload	#Predicates	Min/Max #Predicates	Predicate Distribution
A	732	1/8	Zipfian(1.5)
B	617	1/7	Zipfian(2)
C	607	1/10	Uniform

project[3]. We integrate our data skipping mechanism with the query execution engine of Apache Spark 2.4.

All experiments are conducted in a Linux machine with an Intel i7-5557U CPU and 16 GB RAM. To simulate a server-client deployment, we implement a single-client and server on the same machine. All communication is simulated through file I/O, and all of the experiment processes are single-threaded.

B. Datasets

Yelp Open Dataset: The Yelp open dataset[4] is released by Yelp Inc. We use the 5 GB *review.json* file which includes 6M JSON objects. Each object contains the full review text data and the userId, the businessId, the date of review, and values of 4 other metrics a reviewer can use to evaluate a business.

Windows System Log: The Windows System Log dataset [1] is collected on a Windows 7 machine. The file contains 114M of rows. Each row contains the date and time of the log, the level of the log, the service that generates the log and the log message. The uncompressed file is 27G and spans 226 days.

C. Synthetic query workloads

We generate the query workloads for different datasets using a single query template: `SELECT COUNT(*) FROM <dataset> WHERE <conjunctive predicates>`, which can be used to evaluates the cost of scanning tuples from the base tables and shows the performance impact of partial data loading and data skipping. To generate queries for each dataset, we build a predicate pool and randomly draw the predicates from the pool to build each query's `conjunctive predicates`. We generate the predicate pools using predicate templates and change the candidate values for each predicate template. We estimate the selectivity for each predicate by evaluating them on sampled datasets.

To generate the conjunctive predicates of each query, we assign each predicate a probability that indicates whether this predicate is selected from the pool or not. We make sure that each query includes the same expected number of predicates and vary the distribution of how likely a predicate is selected. For example, if the predicate pool size is 100, the expected number of predicates is 3, and we use a uniform distribution, each predicate will be selected with the probability $\frac{3}{100} = 0.03$. It is possible to use a skewed distribution, like Zipfian, to simulate different levels of predicate skewness (i.e. if the distribution is more skew, a small number of predicates have higher probabilities of appearing in a query).

[3]Apache Arrow C++ implementation, https://arrow.apache.org/docs/cpp/

[4]Yelp Open Dataset, https://www.yelp.com/dataset

D. Impact of varied budgets

In this experiment, we generate three query workloads for each dataset to test the data loading time and query processing time of CIAO under varied computation budgets. Each workload includes 200 queries and has a different distribution of the predicate skewness. Table II summarizes the information about three workloads. The `#Predicates` column shows the summation of the number of predicates that are included in all queries. The column `Min/Max #Predicates` shows the minimum and the maximum number of predicates of a query. The last column shows the distribution of choosing the predicates. Here, workload A is highly skewed with a high predicate overlap, which represents the 'easy' case where CIAO can leverage predicate overlap and skewness to achieve the most benefit. On the other end, workload C has a low predicate overlap and the predicates are uniformly distributed, i.e. not skewed. Therefore, workload C represents the 'challenging' case where CIAO can be less beneficial. Finally, workload B is a middle ground of workload A and C. Note that we use Numpy to generate the Zipfian distribution and the smaller skewness parameter means higher skewness in its implementation (i.e. Table II shows that skewness parameters are 1.5 and 2 for workload A and B respectively.)

We first calibrate the cost model for the current hardware configuration. The cost model estimates the number of μs of evaluating each predicate on a JSON object. For each dataset, we vary the computation budget and show how CIAO performs given the same budget on different workloads. Our baseline in these experiments is the one with zero budget (i.e. no optimization is applied). The experiment results are shown in Fig. 3 and Fig. 4 for the Windows System Log dataset and the Yelp Review dataset respectively.

We observe the following trends in the experiment results. We see that the performance of the partial loading (denoted as `Data loading`) varies for different workloads. For the 'easy' workload A, partial loading is used even if the administrator uses a very small budget on clients. As expected, highly overlapped predicates and skewed distribution of predicates is beneficial for the optimization. For workload B, a small budget limits the number of predicates that can be pushed down. In such a case, there may not be an opportunity for partial loading (i.e. there is no tuple that is invalid for all queries). As we increase the budget, more predicates are pushed down and the server can now avoid loading irrelevant JSON objects by utilizing the bit vectors. For workload C, the server does not employ partial data loading due to low predicates overlapping and low skewness. Among all the workloads for all datasets, CIAO can accelerate the data loading process by 21x, with the budget of 1 μs.

In addition to partial data loading, our experiments show the results of the query processing time (denoted as `Query`). This is the total time to run the full workload of 200 queries. We see that as we have a larger computation budget, we can

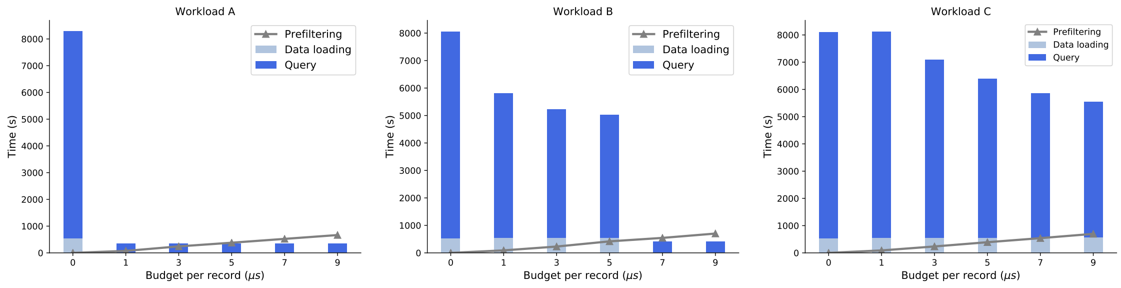

Fig. 3. End-to-End Experiments of the 3 workloads on the Windows System Log Dataset

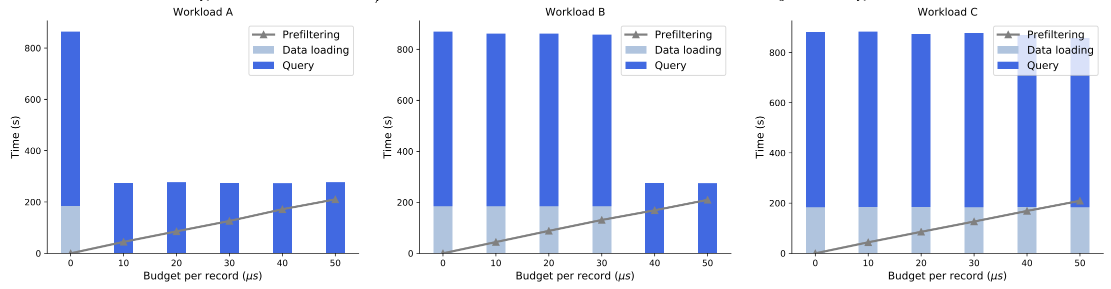

Fig. 4. End-to-End Experiments of the 3 workloads on the Yelp Review Dataset

push down more predicates to the client-side and we can skip more tuples to reduce the query processing time. Specifically, CIAO can accelerate query processing by 23x for the budget 1 μs. Finally, we show the total time of evaluating the predicates on the client-side (denoted as `prefiltering`). We see that the prefiltering time is increased as we have a larger budget.

VII. Conclusion

We present CIAO, an optimization framework for loading data from distributed edge devices. CIAO selectively pushes query predicates to the client devices and leverages string pattern-matching primitives to generate bit-vectors that indicate whether a tuple is valid for a predicate. When the client data is loaded into the databases, CIAO leverages the bit vectors to partially load data and quickly skip data that is not relevant for incoming queries. Our experimental results show that the system substantially accelerates data loading by up to 21x and query execution by up to 23x.

References

[1] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: a large collection of system log datasets towards automated log analytics," *arXiv preprint arXiv:2008.06448*, 2020.

[2] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi, "Time series databases and influxdb," *Studienarbeit, Université Libre de Bruxelles*, 2017.

[3] A. Dziedzic, M. Karpathiotakis, I. Alagiannis, R. Appuswamy, and A. Ailamaki, "Dbms data loading: An analysis on modern hardware," in *Data Management on New Hardware*. Springer, 2016, pp. 95–117.

[4] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "Udp: a programmable accelerator for extract-transform-load workloads and more," in *MICRO*.

[5] S. Noll, J. Teubner, N. May, and A. Boehm, "Shared load(ing): Efficient bulk loading into optimized storage," in *CIDR*. www.cidrdb.org, 2020.

[6] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia, "Filter before you parse: Faster analytics on raw data with sparser," *Proc. VLDB*, vol. 11, no. 11, pp. 1576–1589, 2018.

[7] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: efficient query execution on raw data files," in *SIGMOD*, 2012, pp. 241–252.

[8] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, "Mison: A fast JSON parser for data analytics," *Proc. VLDB*, vol. 10, no. 10, pp. 1118–1129, 2017.

[9] G. Langdale and D. Lemire, "Parsing gigabytes of JSON per second," *VLDB J.*, vol. 28, no. 6, pp. 941–960, 2019.

[10] C. Ge, Y. Li, E. Eilebrecht, B. Chandramouli, and D. Kossmann, "Speculative distributed CSV data parsing for big data analytics," in *SIGMOD*. ACM, 2019, pp. 883–899.

[11] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann, "Instant loading for main memory databases," *Proc. VLDB*, vol. 6, no. 14, pp. 1702–1713, 2013.

[12] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin, "Fine-grained partitioning for aggressive data skipping," in *SIGMOD*, 2014, pp. 1115–1126.

[13] Z. Shang, X. Liang, D. Tang, C. Ding, A. J. Elmore, S. Krishnan, and M. J. Franklin, "CrocodileDB: Efficient Database Execution through Intelligent Deferment," in *CIDR*. www.cidrdb.org, 2020.

[14] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki, "Slalom: Coasting through raw data via adaptive partitioning and indexing," *Proc. VLDB*, pp. 1106–1117, 2017.

[15] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani, "Parallel data analysis directly on scientific file formats," in *SIGMOD*, 2014, pp. 385–396.

[16] A. Abouzied, D. J. Abadi, and A. Silberschatz, "Invisible loading: access-driven data transfer from raw files into database systems," in *EDBT*, 2013, pp. 1–10.

[17] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[18] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking," in *CIDR*. www.cidrdb.org, 2007, pp. 68–78.

[19] S. Choenni, H. M. Blanken, and T. Chang, "On the selection of secondary indices in relational databases," *Data & knowledge engineering*, vol. 11, no. 3, pp. 207–233, 1993.

[20] R. Li, M. Riedewald, and X. Deng, "Submodularity of distributed join computation," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1237–1252.

[21] A. Krause and D. Golovin, "Submodular function maximization," in *Tractability: Practical Approaches to Hard Problems*, L. Bordeaux, Y. Hamadi, and P. Kohli, Eds. Cambridge University Press, 2014, pp. 71–104.

[22] S. Khuller, A. Moss, and J. Naor, "The budgeted maximum coverage problem," *Inf. Process. Lett.*, vol. 70, no. 1, pp. 39–45, 1999.