Scirpy/LAFP: Optimizing Data Access from Pandas Programs

A Credit Seminar Report (CS694)

Master of Technology
in
Computer Science & Engineering

by
Pranab Kumar Paul
(23M0800)

Under the Supervision of
Prof. S. Sudarshan

Department of Computer Science & Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Mumbai - 400076, India

May, 2024

Acknowledgement

I extend my gratitude to **Prof. S. Sudarshan**, my seminar supervisor, for his unwavering guidance and support during the seminar. His weekly meetings and patient explanations have greatly contributed to the smooth progression of the seminar. I also wish to express my appreciation to my senior, **Chiranmoy Bhattacharya**, whose assistance has been invaluable. Our late-night discussions have been instrumental in deepening my understanding of practical concepts.

Pranab Kumar Paul
(23M0800)

Contents

Chapter 1

Introduction

In the current era of computing, the optimization of database and data science applications has become increasingly crucial to manage the growing complexities posed by expanding dataset sizes and demanding computational tasks. The motivation for optimizing these applications is rooted in the necessity to address specific performance challenges prevalent in modern computing environments. These challenges include mitigating the impact of random I/O operations, minimizing network delays, and streamlining inefficient nested or synchronous query submissions that can degrade system performance.

To effectively tackle these issues, various optimization strategies aimed at transforming existing applications are adopted. One key approach involves implementing query batching techniques within loops or outer queries, which helps consolidate database operations and reduce the overhead associated with repeated query executions. Additionally, employing query decorrelation methods enables the more efficient handling of nested queries, enhancing query performance and scalability. Another essential optimization tactic is automated query result prefetching, which minimizes unnecessary data transfers and enhances overall query execution efficiency.

Furthermore, Python-based data science applications, especially those utilizing Pandas, face similar challenges with memory limitations and performance constraints as dataset sizes grow. To tackle these issues, systems like SCIRPy leverage source-to-source transformations and static analysis to optimize data selection and streamline operations, ultimately improving data representation efficiency. Concurrently, LazyFatPandas (LaFP) introduces lazy evaluation techniques within Pandas, facilitating early data selection and reducing computational overhead in ad-hoc exploratory data analysis tasks.

These advancements underscore the importance of optimization in today's data-driven world, where responsive and scalable applications are essential for effective data analysis. By implementing optimization frameworks in both database and data science realms, organizations can navigate modern-scale datasets more efficiently, ensuring optimal performance and resource utilization in demanding computing environments.

Chapter 2

Rewriting Database Applications for Optimization

Rewriting database applications for optimization is a crucial task aimed at overcoming performance challenges due to various reasons. This necessity arises due to issues such as random I/O operations, network delays, and inefficiencies caused by nested queries or synchronous query submission processes. To enhance efficiency and mitigate performance bottlenecks, it becomes imperative to transform existing applications. This can involve strategies like batching multiple database queries occurs with in loops or outer query, query decorrelation to handle nested queries more effectively, automated query result prefetching to minimize unnecessary data transfers, and converting synchronous query submissions into asynchronous processes for improved performance and scalability. Such optimizations are pivotal for ensuring responsive and scalable database applications in today's demanding computing environments.

2.1 Rewriting Procedures for Batched Bindings

Frequent invocation of database queries or procedures/user-defined functions within loops or outer query clauses can lead to poor performance due to random I/O operations and network round-trip delays, especially when involving database access. Hence, Efficient query handling is essential to mitigate such performance bottlenecks. Query decorrelation addresses this issue but restricted only to nested queries.

Listing 2.1: Query Invoking a Simple UDF [1]

```
SELECT id
FROM orders
WHERE market='BSE' AND number_of_offers(item_id, price, currency_code) > 0;

/*Function Definition*/
INT number_of_offers(INT item_id, FLOAT price, VARCHAR currency_code)
DECLARE
    FLOAT price_rupee;
BEGIN
    IF (currency_code == 'Rupee')
        price_rupee := price;
    ELSE
        price_rupee := price * (SELECT ex_rate FROM currencyEx WHERE
        ccode := currency_code;)   /*sq1*/
    END IF
        RETURN SELECT count(*) FROM offers WHERE itemid = item_id AND
        price >= price_rupee;  /*sq2*/
END;
```

Here in the above code listing the user defined function named count_offers consisting of queries get invoked repeatedly for different parameters. The repeated execution of parameterized queries and updates results in suboptimal performance due to the inherent randomness in input/output operations and delays caused by network round-trips.

2.1.1 Batched form of Operations

Processing a set of parameters all at once in a batch is called a batch operation. For example, a relational join can be thought of as an aggregated version of relational selection, in which several records are simultaneously subjected to a parameterized predicate.

Batched Forms of Pure Functions

Let's consider $\text{sq}(x) = x^2$. Its batched form, $\text{sqb}(S_x)$, can be expressed as the set of pairs $\{(x, x^2) : x \in S_x\}$, where S_x represents a set of input values over which the square function is applied.

To put it simply, a function's batch form takes a set of parameters and outputs a set with all of the results that arise from processing those inputs.

Batched forms of Parameterized Relational Queries

An n-tuple set of values for the query parameters $p_1, p_2, ..., p_n$ is processed as input by the batched form qb of a query q with n parameters. The combined results of q for each of the parameter tuples in this set make up the result-set of qb. The n parameters $(p_1, p_2, ..., p_n)$ and the m attributes $(v_1, v_2, ..., v_m)$ returned by the query q are included in each tuple in the result-set of qb.

Listing 2.2: Batched form of Queries in Listing 2.1 [1]

```
/*sq1rb(p): Batch form of the qeury sq1*/
SELECT p.currency_code, q.ex_rate
FROM p JOIN currencyEx q ON p.currency_code = q.c

/*sq2rb(p): Batch form of the query sq2*/
SELECT p.itemid, p.price_rupee, count(q.itemid) AS number_of_offers
FROM p LEFT OUTER JOIN offers q
ON q.itemid = p.item_id AND q.price >= p.price_rupee
GROUP BY p.item_id, p.price_rupee
```

Batch-Safe Operations: Batched forms are mainly defined for a specific restricted class of operations which is referred as the *batch-safe operations*. When processing a set of arguments, a batch-safe operation satisfies the following two conditions:

1. The outcome of the operation for any parameter remains consistent regardless of the processing order of the parameters.
2. The resulting system state remains consistent regardless of the processing order of the arguments.

Rewriting Loops to Use Batched Forms

An operation q inside loop L of program P is considered batchable with respect to loop L if we can modify P to P' by moving q's invocation outside the loop and replacing it with a single invocation of the batched form qb, ensuring equivalent functionality. This approach optimizes program execution by consolidating repetitive operations into a single batched invocation outside the loop. While an operation may be deemed batch-safe, there can be instances where batching a particular invocation of the operation within a loop is not feasible due to potential side effects resulting from actions within the loop.

Generating Batched Forms of Procedures

For any function f that is side-effect free or a batch-safe operation, including complex procedures, Its simple batched version can be generated as shown in Figure 2.1. This is encapsulating the procedure in a loop that repeatedly invokes the procedure while iterating over a collection of parameters. But for a batch size of k, the overhead (cost($fb\text{-}trivial$)) is almost equal to k times cost(f), thus this type of rewriting does not produce much advantage. However, this rewriting technique can still be useful in client-server contexts to minimize round-trip delays.

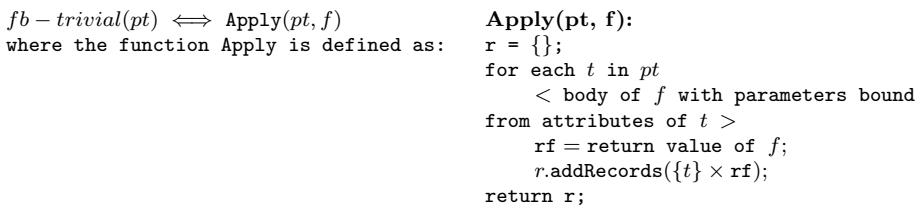

Figure 2.1: Trivial Batched Form of a Procedure [1]

2.1.2 Program Transformation

In Section 2.2.1, an operation invocation within loop L is considered batchable if it can be rewritten outside the loop using its batched form qb, assuming the operation is batch-safe. Program transformation rules presented enable batching of statements within loops under specific conditions, achieving program refinement and equivalence through iterative application of rules.

Rewriting Set Iteration Loops (Rule 1)

In a direct scenario, a loop consists of a single statement that executes the intended operation for batching. Specifically, we examine cursor update loops, which involve iterating over each record in a query or table. This loop format, depicted in Figure 2.2, iterates through a series of tuples, and the operation's outcomes are assigned back to the attributes of the tuple corresponding to the current iteration. The symbol Π^d represents the projection operation without duplicate elimination. This type of rewriting, as illustrated in Figure 2.2, exemplifies how batching can be applied within cursor loops to efficiently process and update records in a database context.

Figure 2.2: Batched form of cursor loop [1]

Splitting a Loop (Rule 2)

Batching a statement within a loop requires segregating the target statement from other loop contents to facilitate batched execution. The goal is to partition the loop into segments where the statement intended for batching is isolated, enabling the application of transformation rules to optimize batch processing. This kind of rewrite is shown in figure 2.3.

If a loop has a series of statements ss made up of two successive sub-sequences $ss1$ and $ss2$ (i.e., $ss = ss1 + ss2$), and if there are no output dependencies or loop-carried flow between $ss1$ and $ss2$, or between these statements and the loop condition, then the loop can be divided into two loops with $ss1$ and $ss2$.

Reordering Statements (Rule 3)

Because of the loop-carried flow dependency from s3 to s1, we are unable to split the loop and go directly to batch s1. Statements S1 and S3 must be rearranged in order to resolve this problem before the loop is divided. In the fugure 2.4, this is seen.

Batching Across Nested Loops (Rule 4)

Loops in a program can be nested within one another, forming a hierarchical structure. The query or update operation that we want to batch could be located at any level within this loop hierarchy. Rule 4 facilitates this transformation. Essentially, we attempt to extract the statement from the innermost enclosing loop first and then proceed to extract it from subsequent higher-level loops.

```
for each r in SELECT grantid, empid, gnum FROM grantload loop
    int internalid = foo(r.grantid, r.empid);
    INSERT INTO grants VALUES (internalid, r.empid, r.gnum);
    total += r.gnum;
end loop;
```

```
TABLE(key, empid, gnum, internalid) t;
int loopkey = 0;
for each r in SELECT grantid, empid, gnum FROM grantload loop
    RECORD(key, empid, gnum, internalid) s;
    int internalid = foo(r.grantid, r.empid);
    s.key = loopkey++;
    s.empid = r.empid;
    s.gnum = r.gnum;
    s.internalid = internalid;
    t.addRecord(s);
end loop;

for each s in t loop order by key
    INSERT INTO grants VALUES (s.internalid, s.empid, s.gnum);
end loop;

for each s in t loop order by key
    total += s.gnum;
end loop;
```

Figure 2.3: splitting a cursor loop [1]

```
s0:     while (category != null) loop
s1:          int count = q(category); // Query to batch
s2:          sum = sum + count
s3:          category = getParentCategory(category);
        end loop;
```

Figure 2.4: Motivating Example for Reordering Statements [1]

Experiment and Result
From various experiments it is observed that the techniques introduced in this paper results in approximately 40% reduction in processing time. These rewriting techniques increase the program size by approximately 15% but the program transformation tooks verly little time (less than a second).

2.2 Holistic Optimization by Prefetching Query Results

This paper focuses on improving the performance of database and web-service applications by introducing an automated query result prefetching technique. The proposed approach aims to enhance performance through optimized prefetching, with a focus on minimizing unnecessary prefetches. It addresses shortcomings commonly found in traditional prediction-based prefetching methods, aiming to achieve more efficient and effective data retrieval and caching strategies.

Listing 2.3: Motivating Example for Prefetching Oppurtunities [2]

```
void generateReport(int cust_id, int currency, String Date){
  ResultSet res=executeQuery(SELECT * FROM Bank  WHERE custId=?, cust_id); /*q1*/
  while(res.next()){
    int account_id = res.getInt ('accId');
    processAccount(res);
    processTransactions (account_id, Date);
  }
  ResultSet q=executeQuery (SELECT * FROM consumer WHERE custId=?, cust_id); /*q2*/
  processCustomer(q);
}
```

The goal is to optimize query performance by strategically initiating prefetch requests for subsequent queries as soon as their required parameters become available, such as at the beginning of a function. This approach aims to maximize overlap between network latency and query execution with local computation by initiating asynchronous prefetching requests at the earliest possible program points. The objective is to minimize idle time and improve overall system efficiency by proactively fetching data needed for subsequent operations.

2.2.1 Query Anticipability Analysis

The objective is to strategically initiate query prefetch requests asynchronously at program points where network latency, query execution can overlap significantly with local computation, optimizing system efficiency.

Following conditions determine the earliest possible points e to issue query q:

(a) At these locations, all the parameters needed by query q are available.

(b) Executing query q at positions e and p yields the same results.

(c) These conditions are not met by the predecessors of e.

Insert prefetch requests for query q with parameters v only at points guaranteed for subsequent execution of q with parameters v to prevent wasteful prefetching.

Anticipable expressions analysis [3] is a data flow analysis technique employed to eliminate redundant computations of expressions. This technique supports expression motion by allowing the computation of an expression to be moved forward to earlier points along control flow paths.

If every path from u to *End* involves an execution of q that is not preceded by any statement that changes the parameters of q or influences the results of q, then the query execution statement q is **anticipable** at a program point u.

In bit vector analysis for query anticipability, Gen_n sets a bit to 1 if query q is executed at statement n. Kill_n sets a bit to 1 if statement n affects q due to parameter assignment or database updates, assuming any database change impacts q.

To compute query anticipability, propagate data flow information in the reverse direction of control flow, ensuring that data dependencies and influences are traced backwards from query execution points to determine if a query will definitely be executed with specific parameters in subsequent operations. In particular, the data flow equations [2] are:

$$In_n = (Out_n - Kill_n) \bigcup Gen_n \tag{2.1}$$

$$f(x) = \begin{cases} \phi & \text{if n is } End \text{ node} \\ \bigcap_{s \in succ(n)} In_s & otherwise \end{cases} \tag{2.2}$$

Equation 2.1 defines Out_n as the intersection ($\cap$) of In values from each successor of node n. Out_{End} starts as empty ($\varnothing$) since no queries are anticipated at the exit point. A query statement becomes anticipable at Out_n only if it is anticipable along every path from n to the end, reflecting this using set intersection ($\cap$).

The control flow graph (CFG) nodes are inspected in reverse topological order, and until the system achieves a fixpoint, the values of Out_n and In_n are iteratively computed for each node. This iterative process ensures that the data flow information is propagated and updated correctly across the CFG, optimizing the analysis for query anticipability. Anticipability may be impeded by critical

edges in the Control Flow Graph (CFG), which run from fork nodes (nodes with multiple successors) to join nodes (nodes with multiple predecessors). By adding new nodes along the edges and making sure the new node has just one predecessor (the fork node), crucial edges are eliminated in order to remedy this. Anticipability is improved along at least one path with this method, which is frequently employed in code motion optimization.

2.2.2 Prefetch Insertion Algorithm

A novel algorithm to strategically position prefetch requests at the earliest feasible points within a procedure.

Intraprocedural Prefetch Algorithm

The procedure's CFG (Control Flow Graph) is sent into the algorithm, which outputs a modified CFG with prefetch requests added. The **InsertPrefetchRequests** algorithm works as follows: at first, it removes crucial edges by adding new nodes along them. After that, every query execution statement is collected into a set Q, which is used as the foundation for a bit vector-based query anticipability analysis. The methods `appendPrefetchRequest()` and `prependPrefetchRequest()` are also used by the algorithm to handle data reliance barriers and control dependence barriers, respectively. Comprehensive explanations of these techniques are not included here because to space restrictions, but they are available at [2].

Interprocedural Prefetching Algorithm

The interprocedural prefetch insertion algorithm utilizes the program's call graph and Control Flow Graphs (CFGs) of procedures to optimize prefetching operations. It employs the InsertPrefetchRequests as a subroutine, with key modifications including:

1. Adding prefetch requests of the format `submit(sqlQuery, parameterBindings)`, where `sqlQuery` is a string and `parameterBindings` is an array of basic data types, to the collection Q of query execution statements.

2. Relocating prefetch statements from their original positions before invoking `appendPrefetchRequest` or `prependPrefetchRequest`.

The main concept of the interprocedural algorithm is to initiate prefetch operations at the beginning of a procedure and extend them to all locations where the procedure is called. The `InsertPrefetchRequests` procedure is applied to each vertex's Control Flow Graph (CFG) as the algorithm traverses the vertices of the call graph in reverse topological order. After determining if the first statement in the CFG is a prefetch submission, it modifies formal parameters to match actual parameters at each call site by inserting the prefetch statement exactly before the relevant method invocation across all call sites.

2.2.3 More Enhancements

The improvements outlined in this section enable prefetching to occur at earlier stages, leading to enhanced performance. These enhancements rely on program and query transformations that preserve equivalence.

Transitive Code Motion

The objective of prefetching is to overlap query execution latency with local computations or requests to improve performance. Increasing the distance between prefetch requests and query execution statements in the CFG enhances this overlap. Data and control dependence barriers hindering prefetch insertion. To overcome these barriers, control dependence barriers are transformed into data dependencies using 'if-conversion' [4] or guarded statements [1], while data dependence barriers are resolved through anticipability analysis and transitive movement of barrier statements for earlier prefetching.

Chaining Prefetch requests

In this model, query chaining involves registering event handlers for each query. When query results become available, events are fired to invoke subscribed handlers, initiating prefetch submissions for

subsequent queries in the chain. This process continues through a sequence of events and handlers until the chain is fully executed.

Rewriting Prefetch requests

Chaining queries can significantly enhance performance, particularly for iterative query execution with parameters derived from previous queries. Prefetch requests are consolidated, enabling correlation-aware query combination using decorrelation techniques [5]. The merged and rewritten queries are split into individual results stored in cache, leveraging SQL Server's OUTER APPLY [5, 1, 6] syntax for optimization.

Experiment and Result
Through extensive experimentation with the proposed techniques, a notable reduction in execution time ranging from 50% to 70% was observed. Additionally, these techniques successfully issued prefetches for 95% of the queries.

2.3 Program Transformations for Asynchronous and Batched Query Submission

This paper focuses on transforming synchronous query submission programs into asynchronous ones using data flow analysis and transformation rules. It introduces a runtime method to batch asynchronous requests for improved performance benefits.

```
p = executeQuery(query);
q = func(); // some computation not dependent upon p
fun(p, q); //computation dependent on p and q

Code with Asynchronous Query Submission
var_handle = submitQuery(query); // Asynchronous Query Submission
q = func();
p = fetchResult(var_handle); // Blocking call to fetch query result
fun(p, q);
```

Figure 2.5: A simple opportunity for Asynchronous query Submission [7]

In the illustrated figure, the `submitQuery()` function (in its asynchronous form) is overlapped with the computation of the `func()` function. Consequently, the rewritten program facilitates the overlapping of network round trips with local computations.

2.3.1 Models of Asynchronous Calls

There are two models for asynchronous call coordination: the callback model and the observer model.

Observer Model

With this method, the application making the call actively monitors the state of the asynchronous call it started. The caller program will wait until the results are ready if they are necessary for additional computations, essentially stopping operation. Processing call results in the order they were made is a particularly good use for the observer model.

Callback Model

In this method, a non-blocking call is combined with a callback function by the caller program. The callback function is used to handle the call's outcomes once the request has been fulfilled. When managing call outcomes requires simple processing logic and the order of processing is not critical, this event-driven architecture comes in handy.

2.3.2 Basic Transformation

In this section, the authors demonstrate the extension of transformation rules mentioned by Guravannavar et al. [1] to support asynchronous query submission. Every program transformation rule includes a syntactic pattern to match, along with specific preconditions that must be met. These preconditions leverage the inter-statement data dependencies identified through static program analysis.

Basic Loop Fission Transformation

Consider the format given below [7]:

```
while p loop
      ss1; s:  v = executeQuery(q); ss2;
end loop
```

Ensure absence of loop-carried flow dependencies, loop-carried anti-dependencies, and loop-carried output dependencies crossing the points before and after statement s to safely implement the described transformation (2.6).

```
1: procedure LoopFission(p, q, T)
2:     Table(T) t;
3:     int loopkey = 0;
4:     // loop for asynchronous query submission
5:     while p loop do
6:         Record(T) r;     ss'1;
7:         r.handle = submitQuery(q);  r.key =
   loopkey++
8:         t.addRecord(r)
9:     end while
10:        // blocking fetch and execution
11:    for each r in t order by t.key do
12:        ssr; v = fetchResult(r.handle); ss2
13:    end for
14:    delete t
15: end procedure
```

Figure 2.6: Basic Equivalence Rule for Loop Fission [7]

In certain scenarios, the application of above rule may not be feasible due to existing dependencies between statements. However, some of these dependencies can be resolved by reordering statements to eliminate them as mentioned in 2.4, enabling the implementation of the algorithm.

Control Dependencies to Flow Dependencies

If there exists any control dependencies between the statements then we need to convert the control dependencies into flow dependencies.

Nested Loops

When a query execution statement is located within a nested inner loop, splitting both the inner and outer loops can increase the number of asynchronous query submissions before requiring a blocking fetch. This method entails breaking the outer loop after the inner loop has been split. Consequently, the temporary table that was generated during the inner loop division nestles inside the temporary table that was generated during the outer loop division. This sequential process enhances the efficiency of query execution and data retrieval within nested loop structures.

2.3.3 Asynchronous Batching: Best of Both Worlds

This method, termed asynchronous batching, preserves the benefits of both batching and asynchronous submission while circumventing their drawbacks.

- Asynchronous batching reduces network round trips by batching requests.
- It overlaps client and server computations, minimizing database IO and improving response times.
- Memory requirements remain lower compared to pure batching due to smaller batch sizes.

The primary challenge in designing such a system is pinpointing the optimal balance between batching and asynchronous submission. This task involves determining the ideal batch size and number of threads to maximize performance. Three strategies are commonly used for this purpose: the One-or-all strategy [7], the Lower-Threshold Strategy [7], and the Growing Upper Threshold Strategy [7]. These are described briefly below:

- **One-or-all Strategy**: Threads wait for new requests; if there is only one pending request, it is executed individually; otherwise, all pending requests are batched together.
- **Lower Threshold Strategy**: This approach incorporates a batching threshold; if the number of pending requests exceeds the threshold, all requests are batched together; otherwise, only one request is executed individually.
- **Growing Upper-threshold based Strategy**: This strategy introduces an upper limit to the batch size, which starts small but increases dynamically if the database can handle the rate of request arrivals. This helps prevent one thread from executing large batches while leaving other threads idle.

Experiment and Result
Employing asynchronous submission with 12 threads results in approximately a 50% improvement, whereas batching yields about a 75% improvement over 40,000 iterations. Asynchronous batching, utilizing 48 threads with a reduced batching threshold of 300, achieves around a 70% improvement in performance.

2.4 Extracting Equivalent SQL from Imperative Code in Database Applications

This paper proposes an approach to optimizing applications by extracting a concise algebraic representation encompassing imperative code and SQL queries. This representation is translated into SQL to enhance performance by minimizing data transfer volume and reducing latency through fewer network round trips. The below figure will be used for further explanations.

```
findMaxScore() {
  boards = executeQuery("from Board as b where
b.rnd_id = 1");
  scoreMax = 0;
  for(t :  boards) {
    p1 = t.getP1();
    p2 = t.getP2();
    p3 = t.getP3();
    p4 = t.getP4();
  }
  score = Math.max(p1, p2);
  score = Math.max(score, p3);
  score = Math.max(score, p4);
  if(score > scoreMax)
    scoreMax = score;
  }
  return scoreMax;
}
```

Figure 2.7: Motivating Example for extracting equivalent SQL from Imperative Code [8]

2.4.1 System Overview

The paper presents a new holistic optimization technique called D-IR (Dataflow-Intermediate Representation) for program variables in database applications. D-IR combines, into a single algebraic expression, the effects of several program statements on a variable. Using relational algebra and fold operations, this form is further translated into a functional representation (F-IR). In order to optimize F-IR, the study presents transformation rules, which are subsequently converted into SQL to improve performance. The transformation rules are illustrated in Figure 2.8.

2.4.2 DAG Based IR

The methods described use program regions to derive D-IR representations for program variables. Regions are organized sections of programs that include elements like as functions, if-else blocks, basic blocks, and loops. D-IR is an intermediate format that includes imperative code and database queries. It uses a variable-expression map (ve-Map) for program areas and an equivalent expression DAG (ee-DAG).

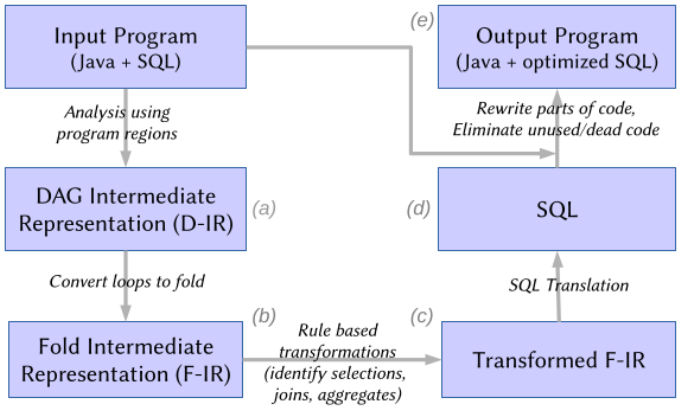

Figure 2.8: System Overview [8]

ee-DAG

An equivalent expression DAG (ee-DAG) is a directed acyclic graph used to represent expressions. Each node in the ee-DAG represents an expression, which can be a constant, variable, or query attribute (base case). Expressions can also consist of operators and operands, where each operand can itself be an expression. The relationship between operators and operands is depicted using directed edges in the graph structure. This graph provides a structured way to represent complex expressions in a hierarchical and interconnected format.

ve-MAP

The ve-Map is a key-value data structure in which a pointer to a node e in the equivalent expression DAG (ee-DAG) is associated with each key, which corresponds to a program variable v. The value of variable v is obtained by evaluating the equation e using the values that are accessible at the start of the program section. This mapping facilitates efficient lookup and evaluation of variable values using the structured representation provided by the ee-DAG.

2.4.3 F-IR Representation

The F-IR (fold intermediate representation) integrates D-IR with the fold function to provide algebraic or functional representations of cursor loops. The fold function, defined recursively, relies on a folding function f, an identity element id, and a recursive data structure to perform efficient fold operations.

`fold [f, id, [ ]] = id`
`fold [f, id, [`a_1`]] = f(id,` $a-1$ `)`
`fold [f, id, [`a_1`, ..., `a_{n+1}`]] = f( fold [f, id, [`a_1 `, ..., `a_n`]], `a_{n+1}`)`

2.4.4 F-IR Transformation

The changes are represented as equivalency rules, each of which accepts an input F-IR and, upon application, yields an equivalent output F-IR.

Rule T1 (Simplification) [8]:

`fold[append, [ ], Q] = Q`
`fold[insert, {}, Q] =` δ`(Q)`

Due to space constraints, only one simple example is presented here, although several rules are discussed in [8].

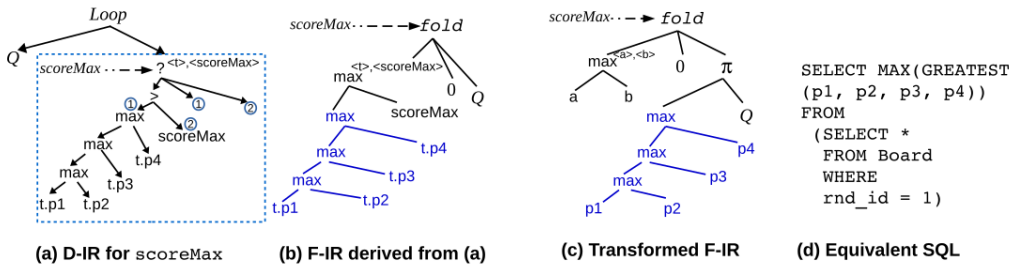

Figure 2.9: Various stages transformations on the code listing in 2.7 [8]

Chapter 3

Optimizing Data Science Applications

In today's data-driven landscape, Python-based data science applications utilizing libraries like Pandas are instrumental for analyzing increasingly large and complex datasets. However, as dataset sizes grow, these applications often encounter performance bottlenecks and memory limitations. To address these challenges, optimization techniques are crucial. SCIRPy [9] is a notable system that optimizes data science applications through source-to-source transformations, leveraging static analysis and transformation rules to enhance data selection, eliminate unnecessary operations, and improve data representation efficiency. Additionally, with the rise of ad-hoc exploratory data analysis (EDA) tasks using Pandas, LazyFatPandas (LaFP) introduces lazy evaluation techniques to optimize performance, enabling early data selection and reducing unnecessary computations based on column or row selections. These advancements underscore the importance of optimizing data applications to handle modern-scale datasets efficiently and effectively.

3.1 Optimizing Data Science Applications using Static Analysis

Data science applications are frequently developed in Python, utilizing libraries like Pandas. However, Pandas operations require data to be loaded into memory, posing challenges with larger datasets due to potential memory issues and performance degradation.

Listing 3.1: Motivating Example [9]

```
import pandas as pd
df = pd.read_csv('data.csv')     /* fetch data */
df = df[df.fare_amount > 0]      /* filter rows */
/* add features */
df['day'] = df.pickup_datetime.dt.dayofweek      /* aggregation */
df = df.groupby(['day])['passenger_count'].sum()
print(df)
```

3.1.1 SCIRPY Framework

This paper discusses techniques to optimize such applications through source-to-source transformations, employing static analysis and transformation rules.

First, the Python source code is transformed into an abstract syntax tree (AST), which is a structured representation of the code, in order to begin the optimization process within the SCIRPy framework. After that, this AST is converted into a JSON object, which makes the representation within the SCIRPy framework more adjustable and manipulable. This intermediate representation (IR) is then subjected to transformation rules that are based on static analysis, with an emphasis on maximizing computing efficiency and data access patterns. The IR is optimized as a result of these modifications. Ultimately, the refined IR is transformed back into Python source code, integrating the implemented enhancements and optimizations. SCIRPy's approach to improving data science applications and enabling more effective data analysis workflows is based on this iterative process of

AST transformation, IR optimization, and code development. Figure 3.1 illustrates this optimization pipeline within the SCIRPy framework.

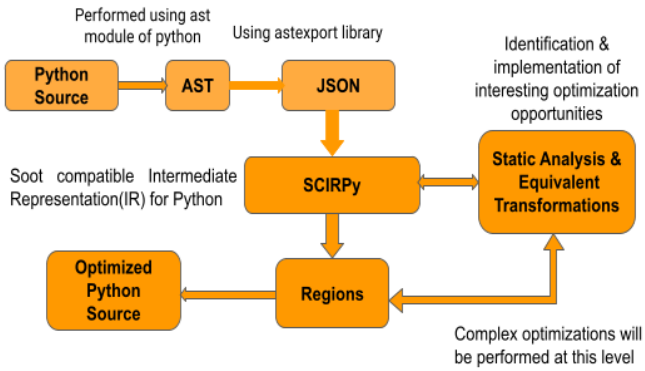

Figure 3.1: SCIRPY Framework [9]

3.1.2 Optimization Techniques in SCIRPy

Data science applications often encounter challenges with memory usage and performance when dealing with large datasets in Python using libraries like Pandas. SCIRPy introduces several optimization techniques to address these issues.

- **Data Selection**: Live attributes analysis (LAA) in Pandas optimizes data retrieval by identifying and retrieving only the columns that are actively used within the program. This selective fetching minimizes unnecessary data movement, resulting in more efficient memory usage and improved performance during data processing. LAA ensures that data operations focus solely on relevant attributes, reducing overhead and enhancing overall computational efficiency.
- **Dropped Columns**: In Pandas, live attributes analysis (LAA) excludes dead columns during dataframe creation, minimizing unnecessary data movement caused by fetching and discarding unused attributes. This optimization enhances memory efficiency and streamlines data processing by focusing only on relevant columns.
- **Metadata Based Optimization**: Using appropriate datatypes in Pandas reduces memory overhead, enhancing performance; SCIRPy's metadata facilitates automatic inference and efficient dataframe generation based on data types, further optimizing memory utilization. This approach ensures efficient data storage and computation in data science workflows.
- **Multistage Data Fetching**: MSDF retrieves necessary columns to avoid out-of-memory issues with large datasets, with a trade-off of longer execution times due to multiple scans. This approach prioritizes memory efficiency while acknowledging potential performance impacts from additional data retrieval steps.

These optimization techniques implemented within SCIRPy play a crucial role in enhancing the efficiency and scalability of data science applications, particularly when dealing with large and complex datasets in Python.

Listing 3.2: Optimizing version of Program listed in Listing 3.1 [9]

```
import pandas as pd
SO_columns = ["pickup_datetime" , "passenger_count" , "fare_amount"]
SO_d_d_t = ["pickup_datetime"]
SO_c_d_t = {"passenger_count" : "int64", "fare_amount" : "float32"}
df = pd.read_csv("data.csv", usecols=SO_columns, parse_dates=SO_d_d_t, dtype=SO_c_d_t)
df = df[df.fare_amount > 0]
df["day"] = df.pickup_datetime.dt.dayofweek
df = df.groupby(["day"])["passenger_count"].sum()
print(df)
```

Here, in the above listing 3.2, the Data Selection optimization retrieves only the columns that are utilized in the program mentioned in listing 3.1. These utilized or "live" columns are stored in the

`SO_columns` list. Additionally, with the Metadata Based Optimization, the data types of the columns are retrieved earlier in the program and stored in the `SO_c_d_t` list.

Experiment and Evaluation
Through various experiments, it's evident that nearly every experiment resulted in a significant reduction in both maximum memory usage and execution time compared to the original program. The transformation process itself is efficient, typically taking around two seconds for a moderately sized program.

3.2 Dynamic Rewriting of Pandas Code

The study discusses the problems with dataframe libraries becoming more and more popular for ad hoc exploratory data analysis (EDA) workloads, such as pandas [11]. These workloads are varied, ranging from unique functions written in Python to routines across libraries, and are difficult for existing systems optimized for bulk-parallel workloads to handle effectively. Although bulk-parallel dataframe libraries improve horizontal scalability, many different types of ad hoc EDA workloads on a single machine are not accelerated by them.

3.2.1 DIAS Overview

Dias is made up of two main parts. Its **syntactic pattern matcher** first verifies syntactic preconditions and aligns the input code with the left-hand side (LHS) of rewriting rules. Then, a **rewriter component** checks if the rules' runtime preconditions are met, and if it is, it changes the code to run on the right-hand side (RHS).

Pandas Rewrite Rules

One of the rewrite rule mentioned in the paper is as follows [10]:

```
df['A'].sort_values().head(n=5)
```

The above pandas statement is taking the first 5 values from the column name 'A' after sorting. So the above statement can be rewritten as follows:

```
df['A'].nsmallest(n=5)
```

This statement is directly fetching the 5 smallest values from the 'A' column. So the two rules are equivalent but the later one is the faster one because of less computations.

But one problem occurs while doing the above transformation is Preserving the correctness of the rewrites. The runtime precondition comes to the rescue and it ensures that the rewritten statement (mentioned in the RHS of the rule) is syntactically equivalent.

```
@{expr: called_on}
.sort_values()
.head(n=@{Constant(int): first_n})
⟶
@{called_on}.nsmallest(n=@{first_n})
```

(a) LHS ⟶ RHS

```
type(@called_on) == pandas.DataFrame
```

(b) Precondition

Figure 3.2: A Rewrite Rule Example [10]

3.2.2 DIAS Rewrite System

As mentioned earlier that DIAS has two main components (a) Syntactic Pattern Matcher and (b) Rewriter. It is time to deep dive into this:

Syntactic Pattern Matcher

The left-hand side (LHS) of every rewriting rule must be matched by a series of statements to be completed by the pattern matcher. On the processed code expressed as a Python abstract syntax tree (AST), it directly matches patterns. [12].

DIAS Rewriter

After successfully matching a piece of code with the lLHS of a rewrite rule, and in the absence of preconditions, the rewriter can substitute the RHS code for the LHS and execute it. This process is straightforward and involves a sequence of AST transformations.

To verify preconditions, the rewriter must infer execution details about the Python program, such as object types at specific points. Given Python's dynamic typing, such details are determined dynamically during execution rather than statically. The rewriter inserts the right-hand side (RHS) and directs program execution to it upon precondition satisfaction; otherwise, it directs execution back to the original code.

Experiment and Evaluation

Dias delivers significant speed improvements at the notebook level, achieving up to a 3.6× speedup. For per-cell operations, Dias is able to achieve speedup upto 57×. Importantly, Dias maintains low overhead, with a maximum of 23 ms and a geometric mean of 0.99 ms. Overall, Dias outperforms Modin [15] by up to 26.4× for whole notebooks, with a geometric mean speedup of 4.1×.

3.3 LazyFatPandas

The LazyFatPandas (LaFP) [13] framework extends Pandas by accepting Python programs designed for Pandas as input. LaFP operates using lazy evaluation, constructing a task graph and executing computations only when results are needed. This approach enables additional optimizations such as early data selection, and skipping unnecessary operations based on column or row selections.

3.3.1 Architecture

LazyFatPandas(LaFP) comprises two main modules: a rewriter for optimizing input programs statically and ensuring compatibility with the Runtime API, and a Runtime API for optimizing the task graph dynamically before execution on lazy frameworks like Dask [14] as mentioned in the figure 3.3.

The initial module of LaFP leverages the SCIRPy framework, which takes a Pandas program as input and applies the optimizations outlined in section 3.1.2. Subsequently, the optimized and rewritten program, along with its metadata, is passed to the Runtime API, the subsequent module within the framework. Here, the program is converted into a task graph, and various optimizations are applied dynamically at runtime, offering significant advantages in terms of optimization. The optimized task graph is then forwarded to the backend executor, where there are three available options: Dask [14], Modin [15], and Pandas [11].

3.3.2 Rewriting for Lazy Evaluation

The rewriter module in LaFP optimizes Pandas programs by replacing the Pandas import with LaFP's custom API and aggregating compute() calls for multiple output statements, enhancing runtime efficiency and backend selection.

- **Delayed Compute**: In the LaFP architecture, the API replaces Pandas calls by deferring computation until the 'compute()' method is invoked later. The placement of 'compute()' raises the question of where to integrate it effectively, with one approach being to add 'compute()' at statements that produce outputs to external devices like print or file writing. The 'compute()'

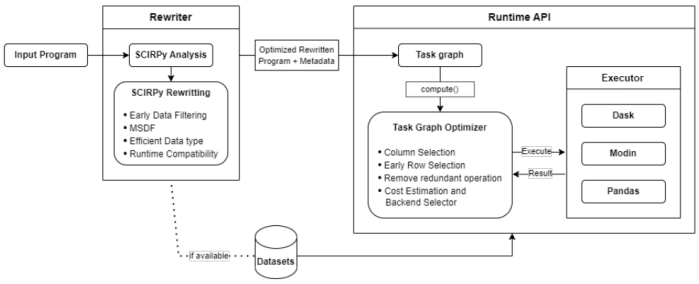

Figure 3.3: LazyFatPanda (LaFP) Architecture [13]

method triggers the computation of the task graph up to that point, involving task graph optimizations, cost estimation, and the actual computation, where the latter consumes the most time. Given the incremental nature of dataframe operations, calling 'compute()' for each operation is costly as it redundantly performs all three steps. To optimize, we delay output statements (and 'compute()') to consolidate operations and reuse intermediate results efficiently.

3.3.3 Runtime Optimizations

Performing optimizations, such as redundant operations removal, at runtime offers greater benefits than static analysis due to access to additional information not available during compile-time.

- **Column Selection**: In Pandas [11], the read* family of APIs typically reads all available columns by default, regardless of whether they are used in the program. Column Selection, similar to a project operation in databases, involves rewriting the input program to read only the necessary columns required for execution. This process aims to precisely identify the columns used within the program, conducted during runtime analysis because challenges arise when conditional statements or incomplete information during the static analysis phase of SCIRPY hinder the precise identification of utilized columns.
- **Row Selection**: Early data-filtering (Row Selection) is a key optimization aimed at reducing data volume for improved efficiency. This involves identifying row selection statements and moving them closer to the data source without altering program semantics. While row selection can be performed during static analysis, it faces limitations due to unavailable runtime information. Therefore, LaFP executes row selection dynamically at runtime to overcome these challenges.
- **Redundant Operation Elimination**: Liveness analysis identifies essential operations in data processing by marking nodes as "live" if their output contributes to the final computation. This technique helps eliminate redundant operations from computational graphs, optimizing workflow efficiency. By focusing on live nodes, unnecessary computations are minimized, enhancing performance and resource utilization.
- **Cost Estimation and Backend Selector**: Backend engine selection for executing the optimized task graph depends on memory requirements, with Pandas for in-memory datasets and Modin/Dask [14, 15] for larger datasets exceeding memory limits, determined by a cost function estimating total memory needs.

3.4 Improvements Done in LazyFatPandas using Dias optimizations

The Dias paper presents optimization rules originally implemented at compile time within Jupyter Notebook cells. In our LazyFatPandas framework, we've tailored these rules to perform optimizations dynamically at runtime, offering notable advantages over traditional compile-time optimizations. This runtime approach enables adaptive optimization based on specific runtime conditions, thereby enhancing performance and flexibility in data processing workflows.

- First we check the preconditions mentioned in each of the rules in the task graph (mentioned in the lazyfatpandas framework).
- If the conditions for a specific rule are met, we attempt to locate the left-hand side (LHS) pattern of that rule within the task graph.
- If the left-hand side (LHS) pattern is identified within the task graph, we proceed by transforming that specific sub-part of the task graph with the right-hand side (RHS) of the rule. During this transformation, a new node corresponding to the RHS rule is created and integrated into the existing task graph structure. This process effectively updates the task graph according to the applied transformation rule.

Chapter 4

Conclusion and Future Work

In conclusion of this survey, optimizing database and data science applications has become indispensable in addressing the performance challenges posed by growing datasets and complex computational needs. The rise of Python-based tools like Pandas underscores the urgency to manage large datasets efficiently and overcome memory constraints. Optimization strategies such as query batching, prefetching, and asynchronous processing are crucial for improving responsiveness and scalability in data-driven computing.

SCIRPy and LazyFatPandas (LaFP) exemplify innovative approaches to optimizing data science applications. SCIRPy's source-to-source transformations and static analysis enhance data selection and streamline operations, addressing critical performance concerns in data analysis workflows. LaFP's lazy evaluation techniques within Pandas optimize performance by enabling early data selection and reducing computational overhead in exploratory data analysis.

Furthermore, integrating optimization strategies like query batching, prefetching, and asynchronous processing is key to enhancing responsiveness and scalability in database applications. Rewriting database queries as discussed earlier allows for efficient query batching and optimization within loops or synchronous processes, mitigating performance bottlenecks associated with frequent database access.

By embracing these optimization frameworks and rewriting strategies, data scientists can navigate modern-scale datasets more effectively, ensuring responsive and scalable applications in demanding computing environments. These strategies not only boost performance but also optimize resource utilization, supporting ongoing growth and innovation in data science and analytics. Overall, optimization is essential for harnessing the full potential of data-driven computing and fostering continuous advancements in the field.

In our future work, we plan to implement the Dias rules within SCIRPy, which involves compile-time rewriting that cannot be executed dynamically at runtime. This integration will expand the capabilities of our framework by enabling more comprehensive optimizations during the compilation phase. By leveraging Dias rules at compile time, SCIRPy will enhance data selection, streamline operations, and eliminate unnecessary computations more effectively than runtime-based approaches.

Bibliography

[1] Guravannavar, Ravindra, and S. Sudarshan. "Rewriting procedures for batched bindings." Proceedings of the VLDB Endowment 1, no. 1 (2008): 1107-1123.

[2] Ramachandra, Karthik, and S. Sudarshan. "Holistic optimization by prefetching query results." In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 133-144. 2012.

[3] Khedker, Uday, Amitabha Sanyal, and Bageshri Sathe. Data flow analysis: theory and practice. CRC Press, 2017.

[4] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In Procs. ACM/IEEE conference on Supercomputing, pages 407–416, 1990.

[5] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex Query Decorrelation. In ICDE, 1996.

[6] I. T. Bowman and K. Salem. Semantic prefetching of correlated query sequences. In ICDE, 2007.

[7] Ramachandra, Karthik, Mahendra Chavan, Ravindra Guravannavar, and S. Sudarshan. "Program transformations for asynchronous and batched query submission." IEEE Transactions on Knowledge and Data Engineering 27, no. 2 (2014): 531-544.

[8] Emani, K. Venkatesh, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. "Extracting equivalent sql from imperative code in database applications." In Proceedings of the 2016 International Conference on Management of Data, pp. 1781-1796. 2016.

[9] Singh, Bhushan Pal, Mudra Sahu, and S. Sudarshan. "Optimizing Data Science Applications using Static Analysis." In The 18th International Symposium on Database Programming Languages, pp. 23-27. 2021.

[10] Baziotis, Stefanos, Daniel Kang, and Charith Mendis. "Dias: Dynamic Rewriting of Pandas Code." Proceedings of the ACM on Management of Data 2, no. 1 (2024): 1-27.

[11] Pandas 2020. API Reference for Pandas. Retrieved 2020-09-30 from https://pandas.pydata.org/pandas-docs/stable/reference/index.html api-reference

[12] Python ast module. 2022. https://docs.python.org/3/library/ast.html.

[13] Priyesh Kumar and S. Sudarshan, "Masters thesis project 2020-2022." 6 2022.

[14] Rocklin, Matthew. "Dask: Parallel computation with blocked algorithms and task scheduling." In SciPy, pp. 126-132. 2015.

[15] Petersohn, Devin, Dixin Tang, Rehan Durrani, Areg Melik-Adamyan, Joseph E. Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. "Flexible rule-based decomposition and metadata independence in modin: a parallel dataframe system." Proceedings of the VLDB Endowment 15, no. 3 (2021).