THE UNIVERSITY *of* EDINBURGH

Compilation and Code Generation for Efficient Data Science

Seyed Hesamoddin Shahrokhi

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

The University of Edinburgh

2024

Abstract

In today's data-driven world, data science plays an important role in benefiting from big data, enabling smart decision-making, and helping innovative acts across industries. The fact that data science creates tangible value for businesses and organizations has resulted in an increasing demand for efficient data science tools.

Python has become the main tool and language of choice for data scientists. It is mostly because of its simplicity and rich ecosystem of libraries such as Pandas, NumPy, and TensorFlow. However, Python's user-friendliness comes with the costs of inefficiency and lack of scalability. This limitation relates to the interpreted nature of this language and also the way its libraries are developed. While previous efforts towards more scalable data science in Python have explored avenues such as fine-tuned low-level kernels, auto-parallelization, and compilation to other languages, their approaches missed some enhancement opportunities or showed only limited coverage on the diverse spectrum of data science workloads.

In this thesis, we adopt a compilation-centric approach to address the challenges of scalable Python data science and introduce a framework comprising different compilation pipelines for the same goal. This framework aims to enhance the efficiency of data science workloads by translating them into SQL/C++. After these transformations, the workloads can be executed by a conventional query engine (RDBMS), with proven optimization and computation power, or a tailored query engine that is crafted for the given workloads. In the case of specialized query engines, we additionally propose a design for optimizing these engines that exploit batch-processing techniques to accelerate the execution and improve the overall performance. We showcase the efficacy of the proposed framework through comprehensive micro and end-to-end benchmarks.

By making data science processing more efficient, our framework not only accelerates data analytics and decision-making but also contributes to sustainable computing practices by reducing the computing resource requirements.

Lay Summary

Data science is a set of methods and scientific techniques for extracting valuable information from large amounts of data. Today, businesses and organizations use data science to make smarter decisions and move faster toward their goals. As the data under analysis becomes larger and larger, a demand for tools that can handle this huge amount of data soars.

People who do data science (data scientists) use a programming language called Python since it is easy to use and has lots of helpful tools around it. However, performing data science tasks using Python can be slow, specifically when the size of the data is very large. This is because of the way Python has been designed and works behind the scenes. Previously, researchers have tried to improve the speed of data science on big data but their efforts have not covered some improvement opportunities or different types of data science tasks.

In this thesis, we came up with a new way of doing fast data science in Python. We created a special system that precisely converts Python language into different languages including SQL and C++. This makes it easier for heavy data science jobs to run faster on the same computer.

Acknowledgements

First of all, I would thank my supervisor, Dr. Amir Shaikhha, whose support, patience, and technical expertise paved the way for my Ph.D. journey.

I am also deeply grateful to my wife, whose constant support and kindness have made this endeavor possible.

Furthermore, I wish to express my sincere appreciation to my parents and sister for their encouragement and continuous support throughout the years of my academic progress.

Finally, I extend my thanks to Huawei for their generous support of the Distributed Data Management and Processing Laboratory at the University of Edinburgh, specifically for funding my Ph.D. studies at this prestigious institute.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Parts of the material in this dissertation have been published in the following publications:

- **Shahrokhi, H.**, Kaboli, A., Ghorbani, M., Shaikhha, A. (2024). PyTond: Efficient Python Data Science on the Shoulders of Databases. In Proceedings of the 40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, Netherlands, May 13-17, 2024.

- **Shahrokhi, H.** and Shaikhha, A. (2023). Building a compiled query engine in Python. In Verbrugge, C., Lhotak, O., and Shen, X., editors, Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Montreal, QC, Canada, February 25-26, 2023, pages 180–190. ACM.

- **Shahrokhi, H.**, Groeger, C., Yang, Y., and Shaikhha, A. (2023). Efficient query processing in Python using compilation. In Das, S., Pandis, I., Candan, K. S., and Amer-Yahia, S., editors, Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023, pages 199–202. ACM.

- **Shahrokhi, H.** and Shaikhha, A. (2023). An efficient vectorized hash table for batch computations. In Ali, K. and Salvaneschi, G., editors, 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States, volume 263 of LIPIcs, pages 27:1–27:27. Schloss Dagstuhl - Leibniz-Zentrum fur Informatik.

I also emphasize that the work explained in Section 4.6, is a concise explanation of an integration effort done by Callum Groger (MInf student) who is co-supervised by me. The complete version of the mentioned chapter will be a part of his degree's final project. Lastly, I would like to add that during the work on this thesis, I also co-authored the following paper:

- Shaikhha, A., Ghorbani, M., and **Shahrokhi, H.** (2023). Hinted dictionaries: Efficient functional ordered sets and maps. In Ali, K. and Salvaneschi, G., editors, 37th European Conference on Object-Oriented Programming, ECOOP 2023,

July 17-21, 2023, Seattle, Washington, United States, volume 263 of LIPIcs, pages 28:1–28:30. Schloss Dagstuhl - Leibniz-Zentrum fur Informatik.

(*Seyed Hesamoddin Shahrokhi)*

Table of Contents

Chapter 1

Introduction

1.1 Data Science and Python

With the exponential growth of data in the digital age, organizations across different sectors increasingly rely on data-driven strategies to gain a competitive edge, improve operational efficiency, and foster innovation. Identification of patterns, trends, and correlations hidden within complex datasets, empowers researchers and practitioners to drive impactful actions.

Data science is an interdisciplinary field that takes advantage of computer science and statistical techniques to extract valuable insights from large volumes of data and facilitate evidence-based decision-making. In the typical scenario, data scientists start solving a given problem by collecting relevant data from different sources. Then, they clean and refine the data, ensuring its cleanliness and suitability for subsequent analysis. After that, using a diverse set of statistical and algebraic methods, they analyze the data to find the answers to their questions. Finally, their efforts manifest themselves in the form of structured text or visualizations that facilitate the interpretation of the analysis outcome.

As a tool for data science, Python language has received a great deal of attention in recent years. The Kaggle survey of the most popular programming languages for data scientists [59] ranked this language in the first place followed by SQL. The simple syntax, easy debugging experience, fast-growing community, and rich set of libraries around this language have made it the first choice for the implementation of data science pipelines even for developers without a computer science background.

Although Python is a popular choice for data science, as datasets grow larger and computational demands increase, the Python data science workloads encounter performance, memory usage, and computational efficiency issues. These scalability limitations in Python and its famous libraries require data scientists either to provide more robust computational resources or rewrite their Python code in another language (such as SQL or C++) when they want to put their tested prototypes into production.

In this thesis, we aim to investigate the challenges in scaling data science workloads in Python and propose solutions for making Python a good choice both for prototyping and production. We plan to achieve this by combining techniques from database, compiler, and high-performance computing (HPC) research domains. As the outcome of this research, we provide design and implementation insights that contribute to the advancement of scalable data science methodologies, facilitating the exploration of large-scale datasets for research, industry, and societal applications.

1.2 Challenges of Data Science at Scale

As mentioned earlier, Python is considered the de facto standard for data science. In this section, we discuss the challenges of Python data science at scale.

Interpretation Overhead. Unlike compiled languages, Python code is executed line by line by an interpreter. This interpreted nature of Python introduces overheads of consecutive language backend calls. This overhead grows with the size of the program and leads to performance bottlenecks and slower execution times.

Lack of Global Optimizations. Since the Python interpreter evaluates each line of code separately, it lacks overall knowledge about the code. As a result, it cannot exploit the opportunities for global optimization (e.g. loop fusion and dead-code elimination) across the entire program. This limitation of Python also leads to sub-optimal performance, specifically when the code is not written carefully.

Memory In-Efficiency. Another critical challenge lies in the inefficient memory usage of certain Python libraries. While Python offers an extensive ecosystem of libraries for data analysis and machine learning, some of these popular libraries may not be optimized for memory efficiency. For example, when working with large datasets, Pandas library [16], which is the most popular Python library for relational algebra processing, may consume excessive memory, causing memory exhaustion errors or slowdowns.

1.3 Towards Scalable Data Science

There is a tail of endeavors towards making Python workloads more efficient and scalable. In this section, we review the major solutions to this problem.

Fine-Tuned Kernels. In this approach, the Python libraries implement their API backends in terms of fine-tuned kernels written in C language (using Python/C API). By taking this approach, the library executes pre-compiled kernels that can be executed significantly faster than interpreting Python native backends. Moreover, the fine-grained memory management features of C, make it easier for the library developers to offer memory-efficient kernels. Famous Python libraries such as NumPy [15], which is used for linear algebra processing, and TensorFlow [21], which is used for machine learning, are built using this approach.

While there are advantages to taking this approach, the fact that the code is still executed by the Python interpreter results in interpretation overhead and missing the

global optimization opportunities.

Parallelization Frameworks. Solutions in this category aimed to improve the scalability of Python data processing tasks using parallelization. Some approaches [1, 6] propose libraries with APIs for writing parallel code, while others [63] offer a new library that adds a parallelization layer on top of an existing library.

Although the described approach can exploit the available hardware more efficiently, it has serious limitations. The explicit parallelization frameworks require users to understand and implement parallel code which is not straightforward specifically for non-technical users. On the other hand, the implicit parallelization approaches require users to import a new library that may not act completely similar to the main library, leading to a poor user experience.

Source Compilation. Previous research in this category [27, 26, 42, 38, 34, 63, 50, 85, 62, 48] explored the ways of transforming Python source code into other languages with more efficient execution backends. The target code can be a lower-level program in C++ or LLVM [50, 85, 62, 48] or a higher-level program in SQL [27, 26, 42, 38, 34, 63]. The most important advantage of this approach over the previous ones is its ability to apply global optimizations over the entire input program. This is usually done by converting the source into an intermediate representation (IR) that facilitates optimizations. Then, the program will be optimized and converted to the target language. This approach also brings the program out of Python's interpreted environment and bypasses its related overheads. Last but not least, if the generated low-level code is memory-efficient, or a standard query engine is used for the generated SQL, by taking this approach, the memory efficiency issue can also be covered.

Even though the source compilation approach seems to offer more coverage compared to the other approaches, it also comes with limitations. Lowering to typed languages such as C++ requires Python developers to pass input types to the program. Moreover, compilation adds overhead to the execution. However, these compilation overheads are negligible when targeting long-running or repetitive workloads.

In this thesis, by adopting the third category of solutions, we aimed to enhance the performance of Python data science workloads. The details of our proposed framework are discussed in the next section.

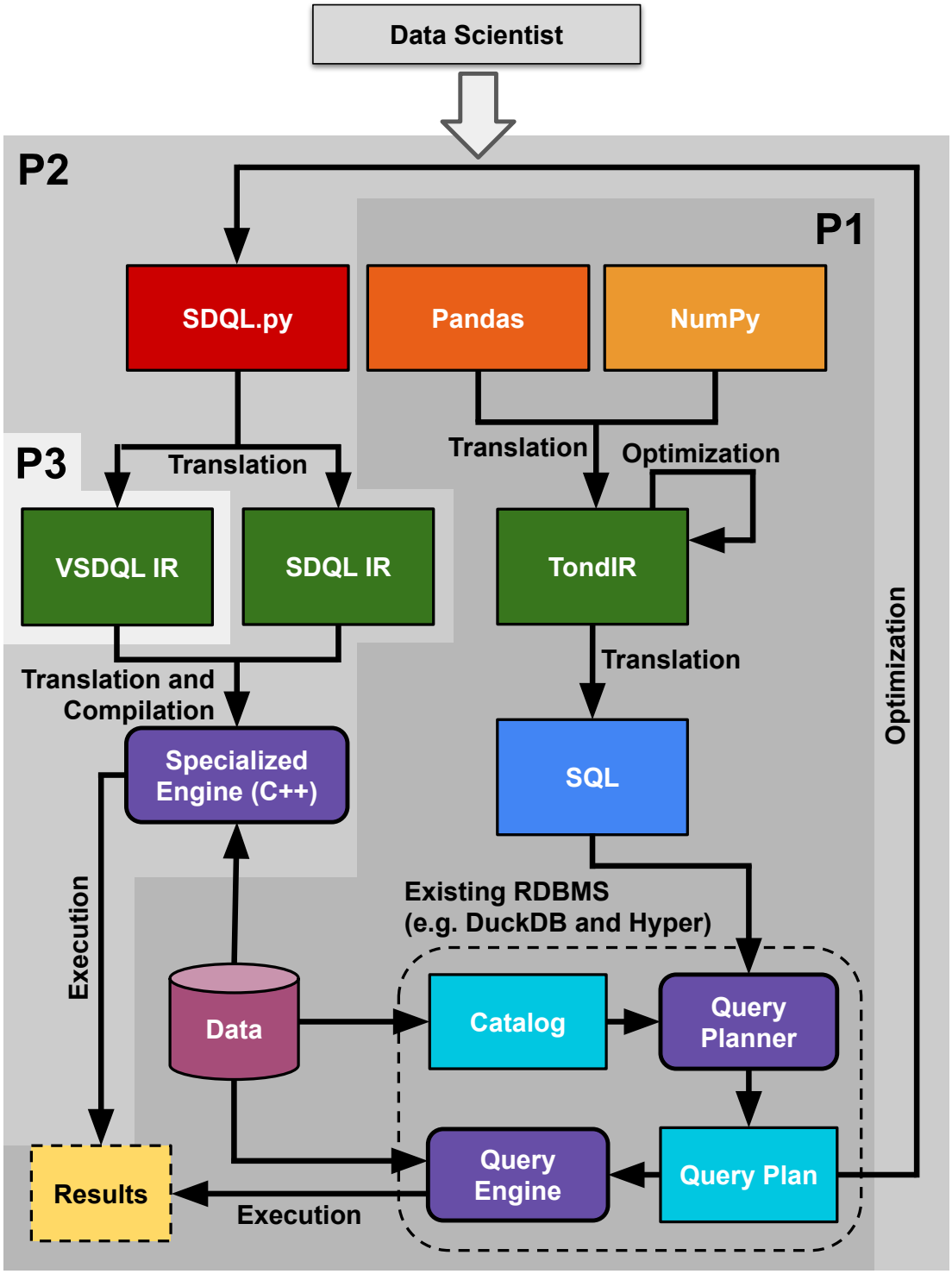


Figure 1.1: An overview of the proposed framework for scalable data science

1.4 Contributions and Structure

A concise overview of what is covered in this thesis is depicted in Figure 1.1. The figure shows three major pipelines (P1, P2, and P3) that shape our proposed framework for scalable data science in Python. In this section, we present the contributions of this thesis while providing the reader with the structure of the other sections.

- We present a comprehensive approach for compiling Python data science (Pandas/NumPy) into optimized SQL (cf. **Chapter 3**). This in-database approach exploits the proven processing power of relational query engines to overcome the scalability issues in Python data science. P1 pipeline in Figure 1.1 shows an overview of this approach.

- We present a compiled query engine embedded in Python (cf. **Chapter 4**). This approach provides a Python DSL for writing analytical queries and transforms the workloads into efficient C++. Then, the generated C++ will be converted to a runtime that acts as a specialized query engine for that specific query. P2 pipeline in Figure 1.1 shows an overview of this approach.

- We present the design for a modern batch hash table that can be used as an underlying data structure to improve the performance of joins (cf. **Chapter 5**). We show how this hash table can be used inside the specialized query engines that we mentioned in the second contribution. P3 pipeline in Figure 1.1 shows the position of this approach with respect to the other elements of our framework.

In **Chapter 2**, we provide the background knowledge required for a better understanding of the thesis. Then, after discussing the contributions in Chapters 3, 4, and 5, we summarize the contributions and discuss the future work in **Chapter 7**.

Chapter 2

Background

2.1 Relational Databases

This section provides background information about relational databases, their underlying data model, management software, querying, and modern design.

2.1.1 Data Model

Relational databases are considered a foundation for a majority of software applications. They provide a structured framework for organizing and accessing data. These systems use the relational model that organizes data into tables, also known as relations, comprised of rows and columns. Each column represents an attribute of an entity being modeled, while each row corresponds to a specific instance of that entity type. This tabular structure facilitates efficient storage, retrieval, and manipulation of data.

In relational databases, each row in a table must be uniquely identifiable. This uniqueness is enforced through primary keys (PK), attributes, or combinations of attributes that uniquely identify each record within a table. On the other hand, foreign keys (FK) establish relationships between tables within a relational database. A foreign key is a column or set of columns in one table that references the primary key in another table. This referencing creates a link between the two tables and ensures that relationships between entities are maintained and enforced.

2.1.2 Relational Database Management System (RDBMS)

Relational Database Management Systems (RDBMS) are complex software solutions designed to efficiently store, manage, and retrieve structured data according to the principles of the relational model. These systems comprise several key components and we briefly describe some of them with more relevance to this thesis work.

Catalog. At the core of an RDBMS, there is a catalog component. This component is a metadata repository that stores schema-related and statistical information about relations. This information acts as a reference guide for database administrators and query optimizers to design better strategies for query execution.

Query Planner. The query planner, also known as the optimizer, is a component responsible for analyzing SQL queries and generating optimal execution plans. This component consumes catalog information and uses heuristics to explore possible query execution strategies and picks the most efficient one to answer the query.

```
SELECT
    d.name AS department, AVG(s.score) AS score_avg
FROM
    department AS d, student AS s
WHERE
    d.id = s.department_id and s.entry_year > 2000
GROUP BY
    d.name
ORDER BY
    score_avg DESC
LIMIT 5
```

Figure 2.1: An example SQL query on sample *department* and *student* relations.

Query Engine. Once the optimal execution plan is determined, the query engine component executes the query on the available data. This component interprets the execution plan suggested by the query planner and manages the retrieval and calculation of data until reaching the expected results. Although all components of an RDBMS are required to be efficient, the query engine is one of the most critical ones since it executes repetitive computations over large amounts of data to prepare the results. Over decades, query engine designers could constantly enhance the performance of this component by using sophisticated techniques. We describe some of these techniques in Section 2.1.4.

2.1.3 SQL

SQL is a simple and powerful language designed for working with relational data. In this section, we review the core elements of this language by using a simple query. Consider the SQL query in Figure 2.1 which is executed on the *department* and *student* relations shown in Figure 2.2. In this query, we aim to calculate the average scores of students in each department, considering only those who entered after the year 2000. In the results, we are interested in the department name and the average score for the top five departments sorted on the average score. Here, we break down the query components and explain each of them separately:

FROM Clause. The FROM clause identifies the tables from which we are retrieving data. In this case, we are fetching records from the `department` table (aliased as `d`) and the `student` table (aliased as `s`). Referring to more than one relation in the FROM clause results in a Cartesian product of the referred tables. In other words, all combinations of records from different tables will be constructed in the output.

department	
id	**name**
1	D1
2	D2
3	D3
4	D4
⋮	⋮

student			
id	**dept_id**	**year**	**score**
1	3	2001	85
2	1	1992	47
3	2	2012	88
4	7	1853	92
⋮	⋮	⋮	⋮

Figure 2.2: A sample schema for *department* and *student* relations.

SELECT Clause. The SELECT clause specifies the columns to be included in the result set. Here, we are selecting the department name (aliased as `department`) and the average score of students (aliased as `score_avg`).

WHERE Clause. The WHERE clause filters rows based on specified conditions. Here, we select records of `student` table where their `dept_id` (FK) matches to the `id` column (PK) of the `department` table and their `year` column is greater than 2000. The first condition that determines the quality of joining two tables is called the join condition.

GROUP BY Clause. The GROUP BY clause categorizes rows based on values of the group-by column(s). In this example, we group the data based on `d.name` to calculate the average score for each department.

ORDER BY Clause. The ORDER BY clause sorts the result set in ascending or descending order. Here, we sort the departments based on their average scores in descending order.

LIMIT Clause. The LIMIT clause restricts the number of rows returned by the query. In this case, we pick the top 5 departments with the highest average scores.

Since it is not trivial to put all SQL commands into one running example and for the sake of brevity, a broader set of primary SQL commands is shown in Figure 2.1.

2.1.4 Query Engine Design

There has been a long tail of research and practice in designing efficient query execution engines. In this section, we discuss the three mainstream designs covering their strengths and weaknesses.

Tuple-at-a-Time. The tuple-at-a-time, also known as the Volcano model [40], processes data one row at a time. In this model, each operator consumes a tuple, performs its operation, and produces the result before moving on to the next tuple. This approach is intuitive and easy to implement, making it suitable for small-scale systems and edu-

Table 2.1: A summary of primary SQL commands.

Command	Description
SELECT	Specifies the columns projected in the results set.
FROM	Specifies the table from which data will be retrieved.
WHERE	Filters rows based on specified conditions.
GROUP BY	Categorizes rows with the same values for a specific column(s) into the same groups.
HAVING	Filters grouped rows based on some conditions, similar to the WHERE clause for rows.
ORDER BY	Sorts the result set in ascending or descending order based on specified columns.
LIMIT	Limits the number of rows returned in the result set.
WITH	Specifies a query that can be reused in the subsequent queries.
IN	Checks whether a value matches any value in a list.
EXISTS	Tests whether a subquery returns any rows.
OUTER JOIN	Retrieves records from one or both tables even if there is no corresponding record in the other table.

cational purposes. However, it often suffers from poor performance when dealing with large datasets due to the high overhead from processing each tuple individually.

Vector-at-a-Time. The vector-at-a-time approach [28], also known as vectorized execution, on the other hand, operates on batches of data, processing multiple tuples simultaneously. This approach leverages modern CPU architectures and memory access patterns to achieve higher throughput and better cache utilization compared to the tuple-at-a-time approach. By reducing the overhead of processing individual tuples, vectorized execution engines can significantly improve query performance over the traditional tuple-at-a-time model of execution.

Data-centric Approach. The data-centric approach [60], optimizes query execution by making a compiled version of the query comprising tight loops of interleaved operations. The compiled code operates on the data organized in columnar format and exploits high data locality. Although the data-centric and vectorized approaches are considered state-of-the-art query processing models, to the best of our knowledge, none of them are proved to be more efficient than the other in all scenarios [47].

2.2 Python Data Science

In this section, we provide background information about Python and two of its popular data science libraries, Pandas and NumPy.

2.2.1 Python

Python has quickly emerged as a leading programming language famous for its simplicity, versatility, and mature ecosystem. These properties of Python have led to widespread adoption by users with/without computer science backgrounds across various industries and domains.

Python is an interpreted language meaning that it executes a given source code line by line without prior compilation into machine code. This attribute expedites development cycles and creates a rapid feedback loop which is ideal for prototyping and scripting tasks. In contrast, compiled languages do the compilation to machine code before execution. It yields potentially faster performance but sacrifices the agility inherent in interpretation.

Python's dynamic typing system distinguishes it further. In a dynamically typed language, variables need not be explicitly declared with a data type. It makes the development easier and more flexible. This contrasts with statically typed languages, where types must be defined at compile time, potentially reducing errors but again at the expense of verbosity and agility.

Python also offers extensibility through its so-called Python/C API. Using this feature, one could develop APIs and Domain Specific Languages (DSLs) that offer simple interfaces in Python while having efficient C/C++ backends. This fusion of high-level abstractions with low-level performance optimizations empowers developers to build scalable, efficient software solutions. An example of an embedded DSL in Python is *SQLAlchemy* library, which abstracts SQL queries into Pythonic constructs, simplifying database manipulation and enhancing developer productivity.

Decorators are another interesting feature of Python. By putting decorators on Python functions, developers can change the way a function behaves by executing complementary logic before/after function execution. This feature is beneficial specifically in the implementation of concerns such as logging, caching, or authentication while enhancing code readability and maintainability.

Python is popular not only for its generic programming capabilities but also for the wide spectrum of features that its libraries (specifically data science libraries) offer. *NumPy* library provides support for large, multi-dimensional arrays and matrices, along with a useful set of mathematical functions. *Pandas* offers powerful data structures and data analysis tools, facilitating data manipulation and exploration. In the machine learning domain, *scikit-learn* offers a rich suite of algorithms and utilities for

Table 2.2: A summary of primary Pandas APIs.

API	Description
`df[col] | df.col`	Column selection
`df[condition]`	Rows filtering
`df.head(n)`	Top `n` rows selection
`df.col.unique()`	Distinct column values
`df.sort_values(by, ascending)`	Sorting on columns list
`df.apply(func)`	Mapping using `func`
`df.aggregate(func)`	Reduction using `func`
`df.groupby(by, axis)`	Grouping
`df1.merge(df2, how, on)`	Joining DataFrames
`df1.isin(df2)`	Containment filtering
`df.pivot_table(index,...,func)`	Making a pivot table

predictive modeling and data mining. Additionally, *TensorFlow* is a great choice for deep learning, providing a scalable framework for building and training neural networks.

Since the focus of this thesis work is mostly on relational and linear algebra workloads, in the next two sections, we discuss Pandas and NumPy libraries in more detail.

2.2.2 Pandas

The Pandas library is a popular tool for relational algebra (RA) operations in Python. Pandas introduces `DataFrame` as its primary data structure. The DataFrame data structure is a flexible representation of 2-dimensional data; it is easy to add, modify, and remove the columns and rows of a given DataFrame by calling their related APIs. Table 2.2 shows a set of primary Pandas APIs.

The `merge` API works similarly to a join in RA. The `how` argument of this API determines the type of requested operations including the Cartesian product, inner join (default value), left, right, and full outer join.

To demonstrate the `pivot_table` API, consider the following expression:

```
df.pivot_table(index='a', columns='b', values='c', func='sum')
```

In this example, the output DataFrame has a column `a`, and its distinct values are used as group representatives. Then, for all rows that fall into each group, a column

will be associated with each distinct value of the column `b`. In any of those columns, we have the sum of values in column `c`. A sample input DataFrame (`df`) and the result of applying the above API call on it is shown below:

a	b	c
x	v1	10
y	v3	30
y	v1	60
z	v2	20
y	v3	40
x	v2	60
z	v2	50

$\longrightarrow$

a	v1	v2	v3
x	10	60	0
y	60	0	30
z	0	70	0

2.2.3 Numpy

The NumPy library is a popular tool for linear algebra (LA) computations in Python. This library is based on fine-tuned kernels that offer an efficient execution. The basic data structure in NumPy is called *array* which is an efficient representation of a tensor of an arbitrary order. Here is an example of tensors of different orders defined in NumPy:

```
# Order-1 Tensor (Vector)
vector = np.array([1, 2, 3])
# Order-2 Tensor (Matrix)
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Order-3 Tensor (3D Tensor)
tensor_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

One of the interesting and powerful APIs provided by NumPy is called `einsum`. Adopted from the concept of Einstein Notation (Summation) [14], this API provides a concise, comprehensive, and efficient way to write linear algebra operations in Python. Each `einsum` API call takes a list of arguments. The first argument is a string that defines the requested LA operation, while the rest are the required operands. Table 2.3 lists some common LA operations and their equivalent NumPy expression. As it can be inferred from the table, many of the specific APIs for LA in NumPy (e.g., `sum`, `transpose`, `inner`, `matmul`) can be expressed in terms of an `einsum` API. However, APIs such as `all`, `nonzeros`, `round`, and `compress` cannot be expressed solely by using `einsum`.

Table 2.3: A list of common LA operations in NumPy and their equivalent `einsum` expressions.

API	Args of `np.einsum`	Description
`m.sum(axis=0)`	`('ij->j',m)`	Matrix ColSum
`m.sum(axis=1)`	`('ij->i',m)`	Matrix RowSum
`m.sum()`	`('ij->',m)`	Matrix Sum
`inner(v1, v2)`	`('i,i->',v1,v2)`	Vector Inner Prod.
`outer(v1, v2)`	`('i,j->ij',v1,v2)`	Vector Outer Prod.
`m.transpose()`	`('ij->ji',m)`	Matrix Transpose
`matmul(m1,m2)`	`('ij,jk->ik',m1,m2)`	Matrix Mult.
`v.all()`	–	All Set Check
`v.nonzero()`	–	Non-Zeros Index
`v.round()`	–	Rounding
`v.compress(c)`	–	Slicing

Chapter 3

In-Database Python Data Science

In this chapter, we propose PyTond, an efficient way of compiling Python data science workloads into SQL. Most of the contents in this chapter are previously published [73]. An overview of this work is presented in the first pipeline (P1) of Figure 1.1. The contents of this chapter are from a submission in the final stages of the peer-review.

3.1 Introduction

In recent years, researchers have proposed innovative approaches to overcome the scalability constraints of Python and its associated libraries by leveraging techniques from compilers and database communities [26, 27, 38, 42, 63, 34, 48, 50, 62, 74, 85]. One of the mainstream solutions [26, 27, 38, 42, 63, 34] focuses on leveraging the proven computational capabilities of database engines. These approaches translate the source code into SQL, allowing it to be executed on any standard relational database engine (portability). As data often resides inside databases, this approach can also bring the computation near the data and lessen unnecessary data movements. Moreover, by delegating query optimizations to query engines, the solution yields an efficient execution.

Although transforming Python to SQL has promising benefits, the existing proposals have limitations. First, their translation coverage is restricted to a limited set of Pandas or NumPy APIs; to the best of our knowledge, none of the existing proposals support the TPC-H benchmark [42]. Second, their generated code is not idiomatic SQL to fully exploit the potential performance benefits offered by the query engines. Lastly, the efforts on translating linear algebra expressions (e.g., NumPy APIs) to SQL are limited to the sparse data layout, which is not optimized for workloads with dense vectors/matrices [26, 27].

This chapter presents PyTond, an automated approach for the in-database execution of Python data science workloads. PyTond takes workloads written in Pandas and/or NumPy and generates their semantically equivalent SQL code. The translated code is rewritten to idiomatic SQL code to make it more prone to optimizations offered by the underlying query processing engine. Finally, the translation process supports both sparse and dense data layouts for tensors (e.g., vectors and matrices), which enables PyTond to generate optimized SQL code for linear algebra operations.

The contributions of this chapter are as follows:

- We present PyTond, a framework for the portable in-database execution of hybrid Python data science workloads. This is achieved by translating Pandas and

Table 3.1: A summary of state-of-the-art in-database Python execution approaches. Blatcher et al. [27] only supports the `einsum` API in NumPy on sparse data layout. Grizzly [42] has limited support for different Pandas APIs. PyTond provides support for data stored in dense and sparse layouts.

Approach	Generic Python	Pandas	NumPy	Multiple Data Layout	SQL Rewriting
ByePy [38]	●	○	○	◐	○
Blatcher et al. [27]	○	○	◐	◐	○
Grizzly [42]	◐	◐	○	◐	○
PyFroid [34]	○	●	○	◐	◐
PyTond *(This Approach)*	○	●	●	●	●

NumPy APIs to SQL code. We discuss its high-level design that offers expressiveness and efficiency and the background in Section 3.2.

- We show that PyTond is unique in capturing Pandas (relational algebra), NumPy (linear algebra), and their hybrid workloads while considering different data layouts. The key enabling technology in achieving expressiveness is TondIR, a novel intermediate language that enables an extensible translation of Pandas/NumPy APIs (Section 3.3).

- We apply optimizations on the TondIR program to generate SQL code that paves the way to further optimizations and more efficient processing by the underlying database engine. Our discussion on the efficiency of the approach is covered in Section 3.4.

- We conduct benchmarks (Section 3.5) to showcase the superiority of our approach over Python and state-of-the-art for various workloads, including all TPC-H queries and hybrid data science workloads.

3.2 Design Overview

The high-level overview of PyTond is depicted in Figure 3.1. Here we elaborate on each element of the presented design.

Data Science Code. PyTond supports the data science workloads that are written using Pandas, NumPy, or a combination of them. Since our approach is based on static analysis of the source code, data scientists do not need to change their program logic or even the imported libraries. The only required change is to add the `@pytond` decorator on top of their functions (cf. Section 3.3).

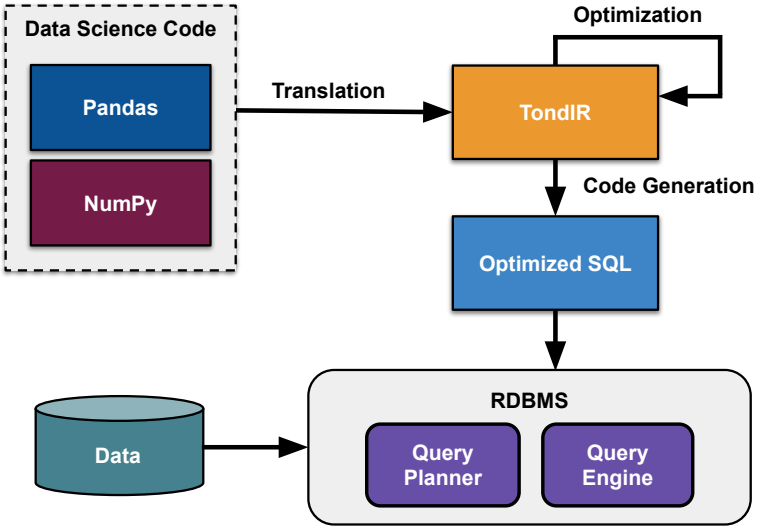


Figure 3.1: High-level design of PyTond.

Translation. The translator module starts by finding the decorated functions in the source code and constructs the abstract syntax tree (AST) of them. Then, it transforms the ASTs into our tailored intermediate representation (IR), based on pre-defined translation rules. The translation process and its various challenges will be discussed in Sections 3.3.2, 3.3.3, and 3.3.4.

TondIR. TondIR is our gateway to expressiveness and our lever to improve efficiency. This IR is designed to capture the intended workloads and is amenable to applying a variety of optimizations. An in-depth analysis of this IR will be done in Section 3.3.1.

Optimization. The optimizer module gets an TondIR code and applies a set of well-defined optimization rules to construct a more efficient version of the input. In Section 3.4, we will investigate the transformations done during the optimization phase. We will also assess its effects on the overall performance of PyTond (Section 3.5).

Code Generation. After the optimization is done, we translate the optimized TondIR code to standard SQL. Our SQL code generation challenges are discussed in Section 3.3.5.

Data. The data that is fed into the data science pipelines is usually stored either in files (e.g. CSV files) or databases. For huge data sizes, the poor performance of Pandas popular file reader API (read_csv) pushes the developers towards using the databases as their main data storage locations. The capability of databases for handling massive data besides their recent advances (e.g. in-memory databases [47]) also justifies this preference. By taking the opportunity of data being kept in databases, our approach can gain access to the data and bring the computation (execution of the optimized

SQL code) near the data. Before the execution, by querying the database catalog, our approach can access the meta-data information (e.g., schema and integrity constraints) and use it during the optimization process. We discuss the potential benefits of having such knowledge in Section 3.3.

Numerical Data Layouts. Focusing on the numerical data, specifically matrices, regardless of the storage location, the data is usually kept in a 2-dimensional format. In this common data layout, each data point can be accessed using its row number and its column name. We refer to it as a *dense layout*,[1] comparing it to *sparse (COO) layout*. To construct a sparse (COO) representation of a given dense input, we first create an ID column (starting from 0), a sequential unique integer column associated with each row. Then, we virtually assign a unique integer number (starting from 0) to each column. Finally, we transform the given matrix to a new 2-dimensional representation with this header: (row_ID, col_ID, val). This layout is beneficial whenever the input data is sparse (many values are None or 0) but needs extra transformations on input/output.

In PyTond, we provide the flexibility of using both dense and sparse layouts by passing this information to function decorators (Section 3.3). Having the support for both layouts, specifically for the LA workloads, makes it possible to process numerical data in its natural format (dense) while benefiting from the advantages of sparse layout when the data is sparse. The state-of-the-art [27] only supports the sparse format.

RDBMS. As the final step in our pipeline, the optimized SQL code is passed to the underlying relational database management system (RBDMS) for further optimization and final execution. The RDBMS receives the query, does a bunch of thoughtful optimizations over it, and passes it to its query engine to be executed on the already-loaded data. It is worth mentioning that the optimizations that are done in both TondIR and query planner level, could not be easily done by the developers or in the interpreted Python environment. Last but not least, the query engine executes the generated physical plan of the query in a fine-tuned hardware-conscious fashion. This execution can exploit the recent advances in query processing including push-based and vectorized engines [47].

3.3 TondIR

In this section, we present TondIR, our intermediate representation designed to capture Python data science workloads efficiently and pave the way for subsequent optimiza-

[1] Blatcher et al. [26] call this representation *database-friendly*.

Table 3.2: Grammar of TondIR. In atoms we have $\theta \in \{ <, <=, =, <>, >=, > \}$

Program	P	::=	$R \mid P\ R$	List of rules.
Rule	R	::=	H `:-` B`.`	Head (access) and body (products).
Head	H	::=	r[`group` $(\bar{x})$][`sort` $(\bar{x},\bar{b})$ [`limit` (n)]]	Aggregates over a relation. Arguments in [] are optional.
Relation	r	::=	$X\ (\bar{x})$	Access to a relation X with variables $\bar{x}$.
Body	B	::=	$a \mid B\ ,\ a$	Body is a chain of atoms.
Atom	a	::=	$r \mid x\ \theta\ t \mid$ `exists` (B)	Relation access, logical, or existential filters.
Term	t	::=	$x \mid agg(t) \mid ext(\bar{x}) \mid$ `if` $(t,t,t) \mid t \diamond t \mid c$	Variable, aggregations, external functions, conditional, arithmetic or constants.
Constants	c	::=	$s \mid [\ \bar{s}\ ]$	Scalar or relation constants.
Scalars	s	::=	$n \mid s \mid b \mid f$	Scalar constants: integer, string, boolean, or floating point.

tions. We start by specification of the TondIR design. Following that, we explain the integration of our translation approach into Python that ensures a seamless transition to TondIR. Next, in dedicated sections, we discuss the translation process from TondIR to Pandas and NumPy. Then, we elaborate on our strategy for translating TondIR to SQL. Ultimately, we wrap up all components and demonstrate an end-to-end translation from Python to SQL.

3.3.1 TondIR Design

To design an appropriate intermediate representation (IR) for our approach, we consider its ability to express hybrid data science workloads involving relational algebra and linear algebra, as well as its flexibility for applying different optimizations. To this end, we designed a logic-based IR (similar to Datalog [22]) with the mentioned properties. The grammar of this IR is shown in Table 3.2.

Grammar. Each program (P) is composed of several rules (R). Each rule is in the form of an assignment from a body (B) to a head (H). A head is an access to a relation (r) with an optional group-by ([`group` $(\bar{x})$]) and an optional ascending/descending ($\bar{b}$) sort with/without limiting ([`sort` $(\bar{x},\bar{b})$ [`limit` (n)]]), over a list of variables. The relation access (r) is described by providing the name of the relation (X) and the list of its variables ($\bar{x}$). The relation column names are bound to the position of each variable X inside $\bar{x}$. This will guarantee sound code generation for column names even after local changes in the variable names in the intermediate translation/optimization pipelines. The body is a chain of atoms (a). Each atom can be a relation access, a comparison/assignment (θ) between a variable (x) and a term (t), or an existential filter (`exists` (B)). If the left side of an assignment is an already defined variable, we con-

sider the operation an equality comparison rather than an assignment. Finally, a term is an aggregation over another term ($agg(x)$), an external function call over variables ($ext(\overline{x})$), a conditional statement (**if**(t,t,t)), an arithmetic operation over terms ($t \diamond t$) or a constant value (integer (n), string (s), boolean (b), floating point (f), or relation ($[\overline{s}]$)).

Contextual Information. Collection of contextual information about the workload can guide the optimizations on the generated TondIR. In PyTond, we use two distinct sources to gather such information: database catalog and decorator arguments.

Since we assume that the data is imported into the database, PyTond queries the DBMS catalog to implicitly collect contextual information about the dataset, and embed them into TondIR constructs. The extracted information includes but is not limited to the table constraints (e.g., primary key, foreign key, uniqueness), and schema information (e.g. cardinality, column names and types).

The other source of collecting contextual information is through the function decorators. As discussed in Section 3.2, to transform a Python function to SQL using PyTond, users need to add `@pytond` decorator on top of their functions. The contextual information can be explicitly passed to PyTond as the arguments of function decorators.

3.3.2 Python Embedding

PyTond scans the Python code to identify functions decorated with `@pytond`. The abstract syntax tree (AST) is extracted for each function using the `ast` module. Then, the extracted AST is passed to the translation preprocessing pipeline.

Normalization. PyTond converts the AST to A-Normal Form (ANF) [56]. In the ANF representation, we extract each nested logic, assign it to a variable, and put a variable access instead of the nested logic. This makes the translation easier as we only need to define rules for simple expressions. Here is an example of converting a Pandas code to ANF:

```
res = (df1[b>10]['a']).merge((df2[y=='r']['x']), left_on='a', right_on='x')
```

```
v1 = df1[b>10]
v2 = v1['a']
v3 = df2[y=='r']
v4 = v3['x']
res = v2.merge(v4, left_on='a', right_on='x')
```

As shown above, the nested projection and filtering logic are decoupled into consecutive assignments to globally unique variables. After this program transformation, PyTond only needs to use a single transformation rule to transform each line of source code. In the normalization phase, to preserve the program semantics, the input variable names (`df1` and `df2`) remain unchanged.

Type Inference. PyTond fetches the contextual information (cf. Section 3.3.1) from decorator arguments and database catalog. Then, having the normalized program and its context, PyTond will do type inference. The extracted type information will be used in translation/optimization pipelines.

Relation Access Renaming. To avoid variable name collisions during TondIR code generation, PyTond assigns a unique name to each variable in each body relation access (`r`).

In the rest of this section, we discuss the rules and challenges of translation from Pandas/NumPy to TondIR.

3.3.3 Pandas to TondIR

The TondIR equivalents of sample Pandas APIs are listed in Table 3.3. Due to the flexible and expressive nature of TondIR, the common Pandas expressions can be easily transformed to this IR.

To translate the merge API, PyTond takes different translation approaches based on the `how` attribute of the API. If it is a Cartesian product (`how='cross'`), PyTond assigns unique names for all variables in both relations. If it is an inner join (`how='inner'`), PyTond assigns unique names to all variables except the join variables, which will have similar names. For the case of full, left, and right outer joins, it takes an approach similar to the Cartesian. However, it introduces special external atoms $ext(\bar{x})$ that contain information about the join type and joined columns. These atoms are called $outer\_full(\bar{x})$, $outer\_left(\bar{x})$, and $outer\_right(\bar{x})$.

Although the presented APIs are only a subset of what is covered by PyTond and Pandas, the extensibility of TondIR makes the coverage of the other APIs feasible. Next, we focus on the challenges in translation from Pandas to TondIR.

Implicit Renaming. In Pandas, when we use the merge API to combine DataFrames with overlapping column names, the API automatically renames the common columns of the first DataFrame to `[col_name]_x` and their corresponding columns in the second DataFrame to `[col_name]_y`. The `_x` and `_y` constants can be changed by setting the

Table 3.3: Translation of sample Pandas/NumPy APIs to TondIR. `df`, `df1`, `df2`, `m`, `m1`, `m2` all have `n` columns. The result of each API call is stored in variable `R`. For example, the first Python code equals to: `R=df[col]`

Library	Python	TondIR
Pandas	`df[col]`	`R(col) :- df(c1, ... , cn).`
Pandas	`df[condition]`	`R(c1, ... , cn) :- df(c1, ... , cn), (condition).`
Pandas	`df.aggregate(func)`	`R(c1, ... , cn) :- df(c1, ... , cn), (c1=func(c1)), ... , (cn=func(cn)).`
Pandas	`df1.merge(df2, how='inner', on='cx')`	`R(x1, ... , x(2n-1)) :- df(a1, ... , x , ... , an), df(b1, ... , cx , ... , bn).`
Pandas	`df.sort_values(by, asc).head(n)`	`R(c1, ... , cn) sort(by, ascending) limit(n) :- df(c1, ... , cn).`
Pandas	`df.groupby(col).sum()`	`R(c1, ... , cn) group([col]) :- df(c1, ... , cn), (c1=sum(c1), .... , cn=sum(cn)).`
NumPy	`v.all(axis=1)`	`R(c1) :- v(ID, c1), (c1=min(c1)).`
NumPy	`v.nonzero(axis=1)`	`R(ID) :- v(ID, c1), (c1!=0).`
NumPy	`v.round()`	`R(ID, c1) :- v(ID, c1), (c1=round(c1)).`
NumPy	`v.compress(condition, axis=1)`	`R(ID, c1) :- v(ID, c1), (condition), (ID=UID()).`
NumPy	`einsum('ii->i', m)`	`R(ID, c1) :- m(ID, c1, ... , cn), (c1=if(ID=1, c1, if(ID=2, c2, ... , if(ID=n, cn) ... )))`
NumPy	`einsum('ij,ij->ij', m1, m2)`	`R(ID, c1, ... , cn) :- m1(ID, a1, ... , an), m2(ID, b1, ... , bn), (c1=a1*b1), ... , (cn=an*bn).`

`suffixes` argument of merge API. If the shared columns are the ones used for joining the dataframes, the API retains the original column name, including only one instance of it in the merged result. An example of a merge operation with the mentioned properties is shown below:

```
df1 = ... # df1 column names: [a,b,c]
df2 = ... # df2 column names: [a,c,d]
df3 = df1.merge(df2, left_on='a', right_on='a')
```

In this example, the header of the output DataFrame (`df3`) will be [a, b, c_x, c_y, d]. During its type inference phase, PyTond considers this implicit renaming to generate correct types for merge API calls.

Implicit Joins. Pandas allows users to append to the columns of a DataFrame. This can be expressed as an implicit join operation. Consider the following example in Pandas:

```
df1 = ... # df1 column names: [a,x,y]
df2 = ... # df2 column names: [b,z]
df3 = DataFrame()
df3['a'] = df1['a']
df3['b'] = df2['b']
```

In this example, we create a new column on the left-hand side DataFrame by using the values from a column on the right-hand side. Note that the program can be even more complex when we update (instead of create) a column in `df3` or have more complex expressions from different DataFrames on the right-hand side. The high abstraction level of DataFrames in Pandas can hide the challenges of handling such scenarios. However, to generate SQL, we must explicitly express the implicit operations done at Pandas background. Back to our example, since `df3` is initially empty, we can generate a projection from `df1` that only projects column `a`. But, in the next line, we need to add a column `b` from another DataFrame (`df2`) and it needs a join between `df3` and `df2`. Here, there is an assumption that the source and target columns have equal rows. PyTond generates the following TondIR in this scenario:

```
df3(a) :- df1(a, x, y).
df3(ID, a) :- df3(a), (ID=UID()).
df2(ID, b, z) :- df2(b, z), (ID=UID()).
df3(a, b) :- df3(ID, a), df2(ID, b, z).
```

The `UID` term is an external atom ($ext(\bar{x})$) and is used to create an identity (PK) column for a relation. In this code, PyTond automatically adds the necessary rules/terms to make the implicit join on `df2` and `df3` explicit. As it is discussed in 3.4, PyTond optimizer will finally remove unnecessarily introduced rules/terms (e.g. when there is a self-join).

Pivot Translation. We presented the interface of Pandas `pivot_table` API in Table 2.2. To translate this API, we need information about the distinct values specified by the `columns` argument. This information can be provided via the decorator or by querying the target columns before code generation. Considering the example from Section 2, its equivalent TondIR code is as follows:

```
R1(a, v1, v2, v3) group(a) :-
  R(a, b, c), (v1=sum(if(b=v1,c,0))),
  (v2=sum(if(b=v2,c,0))), (v3=sum(if(b=v3,c,0))).
```

We must generate distinct columns for unique values in column `b`. To achieve this, we examine the value of this column in every row and determine to which designated columns (v1, v2, and v3) that particular value should contribute. Subsequently, we aggregate the contributed values for each column using a **`sum`** expression and assign the result to the respective column name. Finally, the embedding of the `group` clause ensures the accurate grouping done within the program.

3.3.4 NumPy to TondIR

Table 3.3 shows translations of a subset of NumPy APIs to TondIR. In all NumPy APIs that use arrays in their input, we assume a unique `id` column stored in the input. We also generate/project such a column after each API call if the result is an array. The all API checks if all values in the vector are True. We implemented it by applying the min function to vector values. In the compress API, by passing a conditional expression, we can get the compressed slice of the values in the vector that are matched to the condition. Since this API removes some unique IDs from the `id` column, we need to regenerate unique IDs using **`UID`**. The following section will cover the detailed discussion on einsum API translations.

Next, we explain challenging parts of the translation from NumPy to TondIR.

Einsum Translation. The einsum API translation in our approach has a two-phase pipeline. First, we compute an execution plan for a given einsum expression, and then we generate code by considering the extracted plan.

In the planning phase, we first take the information about data layout (dense vs. COO) from the function decorator. If nothing is passed, we consider the default data layout, which is dense. If the data layout is COO, we take the approach proposed by Blacher et al. [27] but generate TondIR (not SQL). The fully denormalized nature of the COO layout allows us to generate TondIR code for different einsum expressions. On the other hand, if the dense format is requested, we will continue planning for the

given einsum expression as will be discussed.

After thoroughly exploring possible binary einsum expressions (with matrix/vector operands), we came up with a minimal set of kernels listed in Table 3.4. Our planner will use these kernels to construct the other possible einsum expressions. Here, there is a trade-off between automation and efficiency. We can implement more efficient kernels manually, but it needs much more effort. We decided to go with this minimal set of kernels and let the optimizer take the responsibility for optimizing the generated TondIR. The translations for the ES3 and ES7 kernels are shown in the last two rows of Table 3.3.

Now, we explain the translation of einsum expressions to our fundamental kernels based on an example binary einsum expression. Assume that we have the einsum expression `'ab,cc->ba'` as input. This expression can be represented as `'ij,kk->ji'` in TondIR since `a`, `b`, and `c` appeared in the first, second, and third non-repeated position of the expression respectively. Since variable `k` does not appear on the right-hand side, TondIR needs to sum over the whole matrix (`'kk->'`). Therefore, TondIR applies kernel ES3 over the right-hand side of the compression and transforms `'kk->k'`. Then, TondIR proceeds to apply kernel ES1 on the same matrix to produce the expected result `'k->'`. Now, the computation is simplified and turned into `'ij,->ji'`. However, this computation is not a fundamental kernel yet. As the next step, TondIR swaps the left-hand side and right-hand side of this computation and converts it to `',ij->ji'`, which is similar to an einsum computation of kernel ES6. As the final step, TondIR transposes the input using kernel ES4 and creates the einsum expression `',ij->ji'` which is kernel ES6. During the planning process, if either inputs to the einsum API are constant, PyTond will inline constant values inside its predefined kernels (constant folding).

TondIR also supports einsum expressions with more than two inputs (non-binary) using the Python *opt_einsum* library [84]. First, PyTond passes the einsum expression to the opt_einsum path optimizer. Then, opt_einsum breaks the computation to several einsum computations with less number of inputs. Finally, TondIR processes and translates the computations if all the einsum expressions produced by opt_einsum are in a binary format.

3.3.5 TondIR to SQL

To generate SQL from TondIR, we take each rule and create a SQL **`WITH`** clause that contains the SQL equivalent of the logic defined by the rule. Here is an example rule

Table 3.4: List of base `einsum` kernels in PyTond. All other `einsum` expressions can be reduced to these kernels. Given an `einsum` expression, the characters `i`, `j`, and `k` show the first, second, and third non-repeated variables respectively. `_` represents an scalar value. The kernels are presented as args of a NumPy `einsum`.

ID	Fundamental Kernels	Description
ES1	`('i->',v)`	Vector ColSum
ES2	`('ij->i',m)`	Matrix ColSum
ES3	`('ii->i',m)`	Diagonal to Column
ES4	`('ij->ji',m)`	Transpose
ES5	`(',->',s1,s2)`	Scalar Product
ES6	`(',ij->ij',s1,m1)`	Scalar Times Matrix
ES7	`('ij,ij->ij',m1,m2)`	Hadamard Product
ES8	`('ij,ik->jk',m1,m2)`	Batch Vector Outer Product
ES9	`('ij,ik->ij',m1,m2)`	Matrix-Vector Multiplication

and its equivalent SQL code:

```
R1(r, s) :- R(a, b, c),(r=sum(a)),(s=sum(b)).
```

```
WITH R1(r, s) AS { SELECT SUM(a) AS r, SUM(b) AS s FROM R }
SELECT * FROM R1
```

Using this translation approach, a program will be translated to a chain of **`WITH`** clauses followed by a final **`SELECT`** statement that returns the results of the last translated rule.

Final Renaming. To generate SQL, PyTond needs to consider possible collisions on relation/variable names. We present our collision resolution mechanism through an example. Suppose that we want to translate the following TondIR to SQL:

```
// Original schema of R is R(x,y)
R1(s) :- R(a, b), R(a, c), (c<10), (s=a*c).
```

In this example, we do a self-join on relation `R` while applying a filter on the right-hand side relation. Then, we have an aggregation that will be solely projected in the results. Now, we show the SQL code generated by PyTond:

```
WITH R1(s) AS {
    SELECT SUM(R1.y*R2.y)
    FROM R AS R1, and R AS R2
    WHERE R1.x = R2.x AND R2.y < 10
}
```

As it can be inferred from the SQL code, PyTond first set unique aliases for each involved relation. Then, using those aliases, it will rename all variables within the query

using this pattern: `[relation_alias].[original_colname]`. To keep the generated SQL simple, PyTond does not rename any relation/variable when there is only one relation inside the rule. The column names needed for the variable renaming are fetched from the database schema (cf. Section 3.3.1). This phase also reverts possible column name mismatches that joins might deliberately introduce (cf. Section 3.3.3).

Sort and Limit. A Common Table Expression (CTE) in SQL is very similar to the concept of views, but it is temporarily defined in the context of a larger query. CTEs are defined using `WITH` clause in SQL, and similar to the views, they cannot keep the results ordered after an `ORDER BY` command. Since in analytical queries, it is usual to sort and pick only a subset of the results (e.g., almost all of the TPC-H queries), our code generation must be considerate about `ORDER BY` and `LIMIT` clauses in the final rule of the program. To have a semantic-preserving translation, we automatically pick the target terms and put them at the final selection part of the entire program, where the database engine can correctly execute it.

Unique ID Generation. To generate unique identifiers (translation of `UID()` expression) in TondIR(), we use the following window function in SQL:

```
SELECT
    row_number() OVER(ORDER BY col) AS ID
FROM rel
```

In this SQL command, the order of `col` is used as a reference to generate sequential unique IDs. If this argument is not set, the database systems usually use the order of the first column in `rel`. However, because of the non-deterministic behavior of query engines in creating IDs when no ordering column is passed, we only use `UID` in the first appearance of a relation in the code and carry the generated column down to the end of the pipeline. It guarantees our immediate access to the ID column of a relation when it is required. We also do not allow translation rules to remove such a column. In the optimization phase (cf. Section 3.4), we prune the unused IDs and make the code as simple as possible.

Backend Adaptation. Although most relational database engines have adopted the standard SQL as their interface language, there are minor details, mostly in the interface of their external functions (e.g. string operations), that the code generator should be aware of to generate a target-compliant SQL code. TondIR makes it easy to support different dialects of SQL required by DBMSes, without the need to change any other components of PyTond; we generate SQL code compatible with two modern database systems (cf. Section 3.5.1).

```
a=x.merge(y, on='id').
  drop('id', axis=1).
  to_numpy()
b=np.einsum('ij,ik->jk',a, a)
```

(a) Original Python code.

```
v1=x.merge(y, on='id')
v2=v1.drop('id', axis=1)
v3=v2.to_numpy()
v4=np.einsum('ij,ik->jk',v3,v3)
```

(b) Python code after ANF.

```
v1(ID, c0, c1) :-
  x(ID, c0),
  y(ID, c1).
v4_1(ID, c0, c1, c2, c3)
  group(ID) :-
  v1(ID, c0, c1),
  v1(ID, c2, c3),
  (c0=sum(c0*c2)),
      (c1=sum(c0*c3)),
  (c2=sum(c1*c2)),
      (c3=sum(c1*c3)).
v4_2(c0) :- (c0=[0, 1]).
v4_3(ID, c0, c1) :-
 v4_1(ID, c0, c1, c2, c3),
     v4_2(c4),
 (c0=(if(c4=0, c0, c2))),
 (c1=(if(c4=1, c1, c3))).
```

(c) Translated TondIR code.

```
WITH v1(ID, c0, c1) AS {
    SELECT r1.ID, r1.c0, r2.c1
    FROM x AS r1, y AS r2 WHERE r1.ID
        = r2.ID },
 v4_1(ID, c0, c1, c2, c3) AS {
    SELECT r1.ID,
    SUM(r1.c0*r2.c0) AS c0, SUM(r1.c0
        *r2.c1) AS c1,
    SUM(r1.c1*r2.c0) AS c2, SUM(r1.c1
        *r2.c1) AS c3
    FROM v1 AS r1, v1 AS r2
    WHERE r1.ID = r2.ID GROUP BY r1.
        ID },
 v4_2(c0) AS { VALUES (0), (1) },
 v4_3(ID, c0, c1) AS { SELECT
  (CASE WHEN r2.c0=0 THEN r1.c0 ELSE
      r1.c2) AS c0,
  (CASE WHEN r2.c0=1 THEN r1.c1 ELSE
      r1.c3) AS c1,
  FROM v4_1 AS r1, v4_2 AS r2 }
SELECT * FROM v6
```

(d) Generated SQL code.

Figure 3.2: Example of an end-to-end translation from Python to SQL.

3.3.6 Putting it All Together

To conclude Section 3.3, we present an end-to-end example, demonstrating the compilation from a hybrid (Pandas/NumPy) program to SQL (Figure 3.2). In this example, we first join relations `x` and `y` on their shared `id` column. Then, we drop the `id` and convert the result DataFrame to a NumPy array (`a`). Subsequently, we pass the created array as the operands to an einsum API call that computes a covariance matrix.

As the first step, Python code will be transformed to its ANF version (Figure 3.2b). Then, the equivalent TondIR code of the ANF program is generated (Figure 3.2c). As discussed in 3.3.5, TondIR does not allow dropping the ID columns. Thus, the drop API call is ignored in the translation. Since we already have the ID column, the to_numpy API call is also ignored. Finally, PyTond plans the einsum API call and gener-

ates TondIR using the ES8 kernel (cf. Table 3.4). In the last phase of the transformations, based on the rules and techniques discussed in Section 3.3.5, PyTond generates SQL code using the TondIR representation of the program (Figure 3.2d).

3.4 Efficiency

In this section, we discuss the optimizations applicable to the TondIR representation of a data science workload. We will evaluate the effects of these optimizations in Section 3.5.3.

Local Dead Code Elimination. Local optimizations are applicable to a single rule in TondIR. Here is an example:

```
R1(y) :- R(a, b), (x=a), (y=a*b).
```

```
R1(y) :- R(a, b), (y=a*b).
```

As it is shown, there is an assignment in the rule that its variable is not listed in the head of the rule. In such cases, we remove the redundant assignment.

Global Dead Code Elimination. Global optimizations are applied to a TondIR program beyond the scope of a single rule. Consider the following code in TondIR:

```
R1(a, b, c, d) :- R(a, b, c, d), (a<10), (c=d).
R2(a, s) group(a) :- R1(a, b, c, d), (s=sum(a)).
```

```
R1(a, b) :- R(a, b, c, d), (a<10), (c=d).
R2(a, s) group(a) :- R1(a, b), (s=sum(b)).
```

We see that the attributes in the positions of variables `c` and `d` are not used in the second rule but are listed on the head of the first rule. In this scenario, we eliminate these unnecessary attributes from the first rule. This is a program-wide decision that can only be made by having a big picture of the program.

Group-Aggregate Elimination. There are cases where we can eliminate group-by aggregations. Consider this example:

```
R1(ID, s) group(a) :- R(ID, a, b, c), (s=sum(b)).
```

```
R1(ID, s) :- R(ID, a, b, c), (s=b).
```

Table 3.5: List of the flow breakers used in rule inlining.

Flow Breaker	Description
Group By	**group** clause in the head
Sort	**sort** clause in the head
Aggregate	**agg** term in the body
Distinct	**unique** term in the body
Computed Column	**x=t** and **x** is used by the dependent rule
UID	**UID** term in the body
Outer Join	outer-join terms (cf. Section 3.3.3) in the body
Reused Rules	a rule with more than one dependants
Sink Rule	final rule in the program

In this example, there is a group-by-summation on the primary key of the relation `R1`. In such cases, if we know that the grouping column(s) are unique, we can remove the grouping and all the summation columns. PyTond brings this knowledge from the database catalog during the Python to TondIR translation (Section 3.3.1).

Self-Join Elimination. In certain situations, there are redundant self-joins. Consider the following example:

```
R1(z) :- R(a,b1,c1,d1), R(a,b2,c2,d2), (z=b1*c2).
```

```
R1(z) :- R(a1,b1,c1,d1), (c2=c1), (z=b1*c2).
```

In this example, we can eliminate the self-join since (1) it is on a unique column and (2) no filter is applied to either of the relations. In this case, all the information needed from the second relation can be obtained from the first one without any loss. Here, again we use the table constraints knowledge from the database catalog.

Rule Inlining. By applying this optimization, we aim to fuse a chain of rules until we reach specific rules that are not fusible. We call these specific rules *flow breakers*.[2] Thus, in the first step, we find the dependency among rules and make a dependency graph. Then, based on the structure of each rule and the information in the dependency graph, we find the flow-breaker rules. In Table 3.5, we summarized different kinds of flow breakers. An example of rule inlining is shown below:

```
R2(b, c, d) :- R1(a, b, c, d), (a > 1000).
R3(b, d) :- R2(b, c, d), (c != "A").
R5(e, g) :- R4(e, f, g), (f > 100).
R6(b, g) :- R3(b, x), R5(x, g)
R7(b, m) group(b) :- R6(b, g), (m = max(g)).
```

[2]Not be confused with pipeline breakers in query engines.

```
R7(b, m) group(b) :-
  R1(a, b, c, x), (a > 1000), (c != "A"),
  R4(x, f, g), (f > 100), (m = max(g)).
```

This optimization, as it is shown in Section 3.5, is very effective since it prevents the program from creating intermediate results. It also helps the backend engines to make better execution decisions by seeing separated but dependent parts of the program in a more concise format. Since our TondIR design adopts the design of conjunctive query languages, it is worth mentioning that this optimization is conceptually close to the minimization of conjunctive queries via homomorphism [31].

3.5 Evaluation

In this section, we evaluate our approach using different workloads and assess its performance compared to competitors. After presenting the experimental setup, we discuss the results of our end-to-end benchmarks. Then, through micro-benchmarks, we analyze the scalability of PyTond and discuss the impact of different optimizations.

3.5.1 Experiments Setup

To conduct the experiments, we used a single machine equipped with 16GB of DDR4 RAM and an Intel Core i5-1145G7 2.6GHz with four cores and 320KB, 5MB, and 8MB of L1, L2, and L3 cache, respectively. Hyper-threading is disabled. We run experiments on Ubuntu 22.04.3 and use Python 3.10.12, NumPy 1.26.1, and Pandas 2.1.1. To compare the effects of different database engines as our backends, we use two modern database engines with different query execution paradigms. The first one is DuckDB [66], a column-based vectorized in-memory database engine embedded in Python. The second one is Hyper [60], a column-based in-memory engine representative of compiled query engines in our experiments. As our third backend option, we use LingoDB [45], an approach for efficient SQL to LLVM compilation (cf. Section 6). Since LingoDB is a research prototype, we show its results whenever the SQL workloads can be executed correctly on this engine. Specifically, we exclude the Grizzly/LingoDB alternative as it lacks the support for certain SQL window functions required for UID generation (cf. Section 3.3.3). We use DuckDB Python API 0.9.1, Hyper Tableau API 0.0.17971, and LingoDB Python API 0.0.1. We did five warm-up rounds in all the experiments and reported the mean of the next five execution rounds.

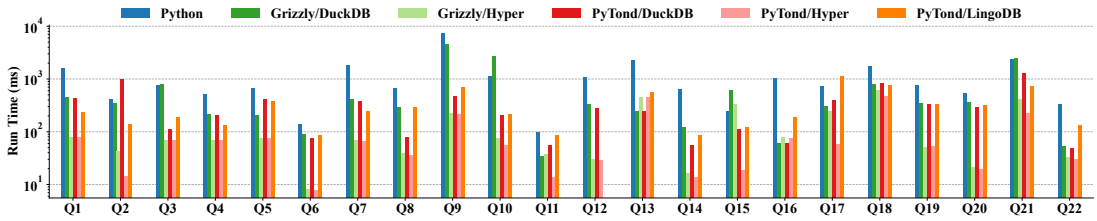


Figure 3.3: Performance results for TPC-H workloads on a single thread.

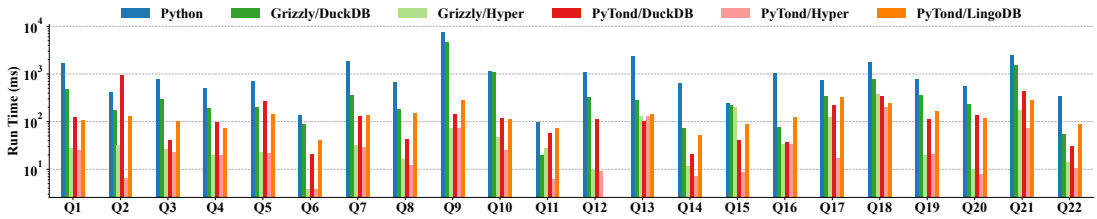


Figure 3.4: Performance results for TPC-H workloads on 4 threads.

Workloads. We included a range of experiments to show the applicability of our approach to various data science workloads. We use the dense layout (cf. Section 3.3) for linear algebra operations since it is (1) similar to the data layout to which the original Python code has access and (2) limitedly investigated by the current approaches [26, 27]. The characteristics of each approach are defined below.

- *TPC-H Queries:* TPC-H benchmark [19] is well-known in the analytical query processing. We use the Pandas version of TPC-H queries from [72] and run the queries on the TPC-H dataset with a scaling factor of 1. Although this benchmark does not represent the wide spectrum of data science workloads, it makes us confident about the system's ability to capture and process relational algebra, which is the backbone for most of the data analytics tasks.

- *Crime Index:* We include the Crime Index data science notebook [62] (with the scaling factor of 100) as a hybrid workload containing a sequence of Pandas, NumPy, and again Pandas operations. It filters an input DataFrame, converts it to an array, computes an einsum, converts the result of einsum back to a DataFrame, and returns the results after further filtering and projections.

- *Birth Analysis:* This is another hybrid data science notebook [62] that contains NumPy *Fancy Indexing* and Pandas pivot_table APIs. We used the available statistical dataset for this experiment.

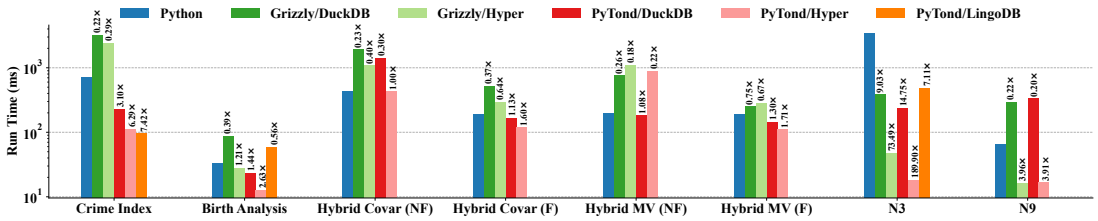


Figure 3.5: Performance results for data science workloads on a single thread.

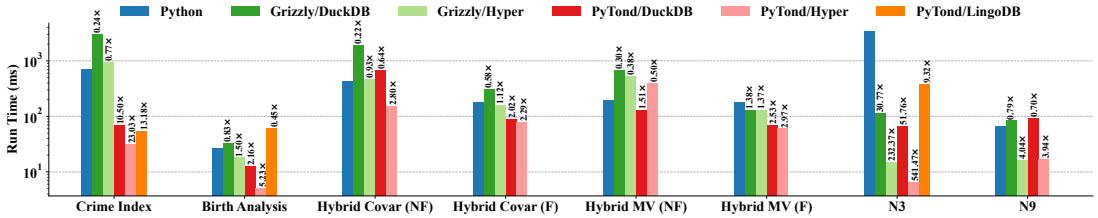


Figure 3.6: Performance results for data science workloads on 4 threads.

- *Kaggle Data Science Notebooks:* We also bring two top-voted Kaggle data science workloads (N3 and N9) into our evaluation. The details of these benchmarks are discussed in PyFroid [34].

- *Hybrid Matrix Calculation Experiments:* Besides the real-world data science workloads introduced before, this category adds two synthetic hybrid workloads. In both workloads, we first use Pandas to join two large tables. Then, we convert the result to a NumPy array. Finally, we execute an einsum over the results. In the first experiment, the einsum operation is a matrix-vector multiplication, while in the second one, it is a covariance matrix computation. We also include a slightly modified version of each experiment (*Filtered* version) where a join-dependent filter is applied to the results of the initial join, before doing the final einsum calculation.

Alternatives and Competitors. To evaluate the performance of our approach, we included two competitors in our experiments. The first one is Python. We compare our in-database approach with the normal execution of workloads in Python, which is backed by Pandas and NumPy implementations. The second competitor is Grizzly [42]. However, as it will be discussed in Section 6, its available version cannot provide the necessary support for our Pandas workloads. We assumed that Grizzly could cover our experimental workloads and consider the challenges in Python trans-

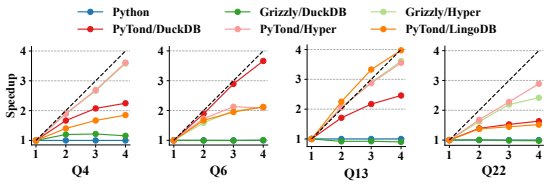


Figure 3.7: Scalability analysis for representative TPC-H workloads on 1, 2, 3, and 4 threads.

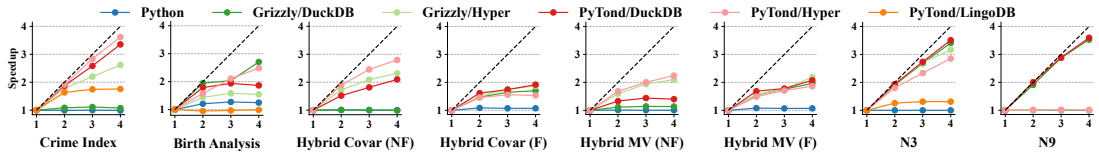


Figure 3.8: Scalability analysis for hybrid workloads on 1, 2, 3, and 4 threads.

lation and SQL code generation (Section 3.3). By putting this assumption, its generated SQL code would be similar to PyTond's generated code before applying the optimizations. We use this simulated approach as our second competitor. There are barriers to including other competitors in our experiments. Firstly, except Grizzly, which we compare against, other very relevant approaches such as PyFroid [34] and Ponder [17] lack accessible implementations for comparison. Secondly, certain competitors such as Daphne [33] do not offer automated translations from Pandas/NumPy, necessitating different workload versions written in their specific APIs. These approaches are aligned with our goal of efficient data science but not with a focus on automated SQL code generation and its mentioned challenges. Alternatives such as Modin [63] and Dask [1] are known for their high memory demand or distributed setup requirements [34]. However, PyTond aims to assist data scientists whether they are on light-weight commodity workstations or the cloud. Our preliminary analysis of Modin (on the same machine described in Section 3.5.1) shows poor performance (slower than Pandas) across TPC-H, Crime Index, and Birth Analysis benchmarks.

3.5.2 End-to-End Benchmarks

In our end-to-end benchmarks, we measure the performance of all workloads when executed by our predefined alternatives. Since we assume that the data is already settled in the database (cf. Section 3.2) and to do a fair comparison, we exclude the data loading time from all alternatives. The vertical axis in all Figures is in log-scale format.

We depicted the results of end-to-end experiments in the Figures 3.3, 3.4, 3.5, and 3.6. Figures 3.3 and 3.4 show the results for all TPC-H queries on 1 and 4 threads,

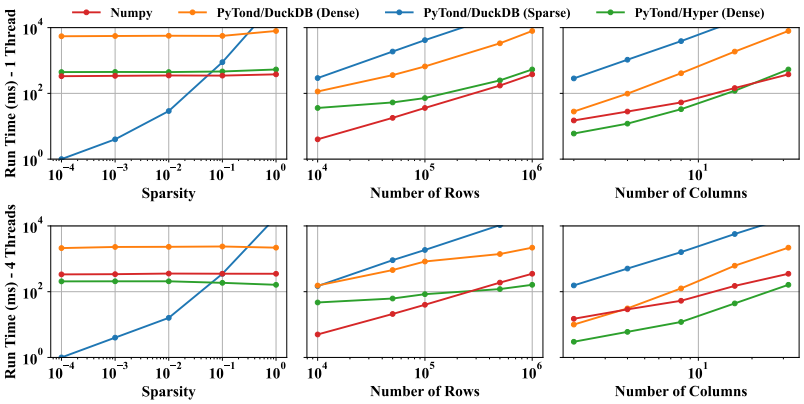


Figure 3.9: Performance comparison of Covariance Matrix computation in NumPy vs PyTond with different backends and layouts. The PyTond/Hyper (COO) alternative is eliminated because of its incompetency. We cut the very large values for PyTond/-DuckDB (COO) to improve the readability of the chart. We used 1,000,000 rows, 32 columns, and a sparsity of 1 for the fixed dimensions. Both axes are on a logarithmic scale. The

respectively. The initial observation drawn from these two charts is that PyTond is the first approach, offering complete coverage for the TPC-H benchmark. Figure 3.3 also shows that for almost all queries, even the Grizzly-simulated approach (on different backends) performs better than the normal Python execution. It also shows that our approach can make the Grizzly-simulated approach much more efficient using its novel SQL rewriting. The effects of optimizations done in PyTond are more significant on the DuckDB backend, as we see a geometric average of 2× and 1.5× speedup on all queries after enabling them in DuckDB and Hyper. This implies that Hyper backend does more advanced query planning than DuckDB. Figure 3.4 also verifies the same observations seen in Figure 3.3. Furthermore, it highlights that besides its inherent superiority over Python, our approach can benefit from parallelization, a feature not offered by Pandas. To summarize our TPC-H benchmarks, compared to Python and on a single thread, PyTond achieves a geometric average of 16× and 5× speedup on Hyper and DuckDB, respectively. It also achieves a geometric average speedup of 39× and 11× on 4 threads. About the LingoDB backend, although its join processing could not process our generated SQL for Q12, in all other cases, its performance is similar to DuckDB.

The next two figures are related to our real-world and synthetic data science experiments. Successful execution of these experiments implies that our approach can capture Pandas/NumPy workloads with a variety of (complex) APIs such as DataFrame

manipulations, pivot_table, and einsum. Figure 3.5 shows that PyTond is still competent or more efficient than Python and Grizzly-simulated approaches. These experiments reveal that even a systematic approach for Python to SQL translation (Grizzly-simulated) cannot offer high performance before applying the necessary optimizations. This can be clearly seen in the relative performance of PyTond and Grizzly-simulated in all experiments, especially the Crime Index. Further investigation on the effect of each individual optimization on the performance of PyTond is covered in Section 3.5.3. Similar to Figure 3.4, Figure 3.6 also verifies the same mentioned facts about the single-threaded setting of hybrid experiments while still showing the superiority of PyTond on 4 threads. An interesting result emerges from the N3 experiment, where PyTond achieves two orders of magnitude higher performance than Python. In this experiment, the Pandas data science code conducts consecutive relational algebra operations on 700MB of airline data. By comprehensively analyzing the entire code, PyTond generates a compact and efficient SQL that executes the entire pipeline with minimal data copying and movement, leading to significant efficiency gains.

To investigate the performance of our approach in the context of linear algebra, we did a benchmark on Covariance Matrix computation, which is a fundamental algebraic operation used in machine learning. The results of these experiments are shown in Figure 3.9. In this benchmark, we take three primary properties of the input matrix including the number of rows, number of columns, and sparsity into account. In each chart, we fix two of these properties and vary the other one. The sparsity (left-most) charts clearly demonstrate that our COO approach overtakes NumPy and other alternatives when the data is sparse. This is a significant achievement since the majority of machine learning models are sparse matrices. The other charts also imply that PyTond on Hyper is faster or competitive to NumPy with different input shapes.

3.5.3 Micro-Benchmarks

This section shows the results of two different sets of micro-benchmarks. First, we assess the scalability of our approach on the number of threads. Then, we show the individual effects of each optimization on the PyTond's overall performance.

Scalability Analysis. Figures 3.7 and 3.8 illustrate the scalability of Python, Grizzly-simulated approach, and PyTond on DuckDB, Hyper, and LingoDB backends. The scalability of each approach is shown in terms of speedup over its single-threaded performance. For brevity, we included a representative set of 4 TPC-H queries [74] in our

scalability experiments. These Figures show a smooth scale-up in PyTond's performance specifically on DuckDB and Hyper backends. We can even find a near-linear scale-up in different cases including Q4, Q6, Q13, Crime Index, N3, and N9. In the Birth Analysis and N9 cases, the overall optimized run time is small, so the scalability results show saturated performance for some backends. In all of these experiments, there are two root causes behind Python's poor scalability. First, Pandas library does not support parallelization. Second, the more time-consuming part of hybrid workloads is attributed to Pandas code.

Effect of Optimizations. We analyzed the effects of optimizations for a representative subset of the workloads by starting from our baseline code (Grizzly-simulated) and applying each optimization on top of the others. The results are portrayed in Figure 3.10. This figure implies that all of the optimizations introduced in Section 3.4 are beneficial in their relevant scenarios. The *O1* alternative shows the effects of *Dead Code Eliminations* that improve the performance of Q9 in DuckDB. The *O2* alternative adds the *Group/Aggregate Elimination* and affects the Crime Index on both backends. This inefficiency is because of using a generalized einsum translation in the Crime Index that cannot be avoided (cf. Section 3.3.4). The *O3* alternative further improves the previous ones by applying *Self-join Elimination*, which affects the Crime Index and Hybrid Covar on both backends. Finally, in *O4*, we introduce *Rule Inlining*, that its individual and combined effects result in huge performance improvements in most of the experiments.

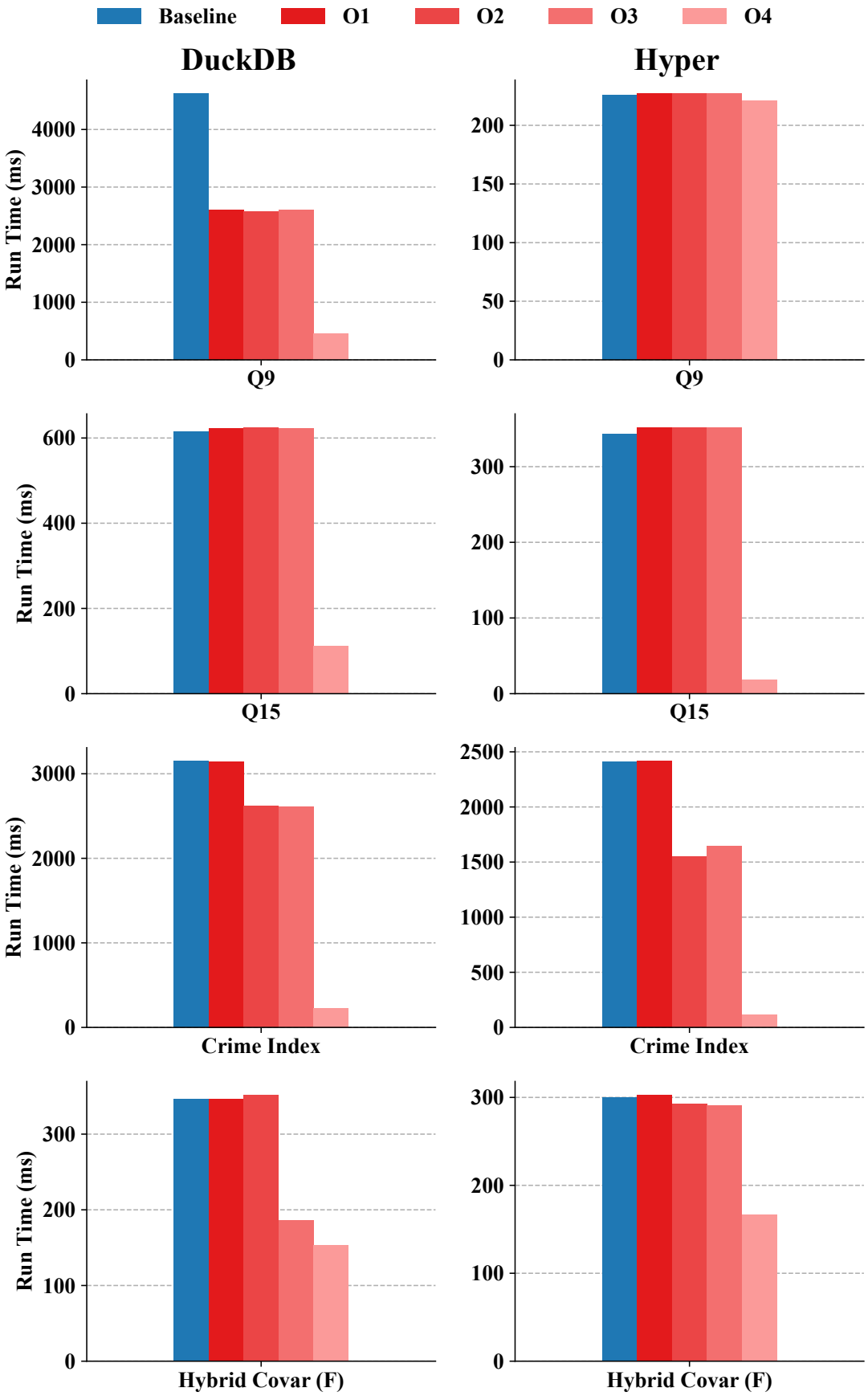


Figure 3.10: Break-down of optimizations applied to experimental workloads. **O1**: Global/Local Dead Code Eliminations, **O2**: O1 + Group/Aggregate Elimination, **O3**: O2 + Self-join Elimination, **O4**: O3 + Rule Inlining

Chapter 4

Compiling Database Queries to Low-Level Code

In this chapter, we propose SDQL.py, an approach for compiling analytical queries written in Python to efficient C++ code (P2 in Figure 1.1). Most of the contents in this chapter are previously published [74, 72]. The work explained in Section 4.6, is a concise explanation of an integration effort done by Callum Groger (MInf student) who is co-supervised by the thesis author. The complete version of the mentioned chapter will be a part of his degree's final project.

4.1 Introduction

Python provides native interfaces for interoperability with low-level code. The framework developers implement performance-critical kernels in C/C++ with a Python API. Frameworks such as NumPy, TensorFlow, and SciPy fall into this category. As mentioned in Chapter 1, while this approach significantly improves performance, it does not benefit from global optimization opportunities. The other primary approach aims to resolve the scalability problem by using compilation. Python frameworks such as Numba [50], JAX [39], and Glow [69] holistically optimize Python functions, mainly focusing on tensor and linear algebra processing.

The database research has observed a long tail of work on just-in-time compilation of queries [23, 47, 41, 60, 86, 45]. However, state-of-the-art research in query compilers is not well-connected to the Python language. For instance, Pandas, as the most relevant framework for query processing in Python does not benefit the optimization such as compilation, parallelization, global optimization, and vectorization. This gap deprives Python developers of access to the advances in query compilation.

Inspired by the line of previous work on query compilers, there have been recent efforts on employing query compilation for Python libraries such as Pandas [62, 33] and UDFs [85]. However, there is no framework that supports all the query plan operators that are necessary for the efficient processing of analytical queries, such as semi-, anti-, outer-, and group-join.

On the other hand, most of the existing works on query compilation or embedded query processing (e.g. DuckDB [66] and SQLite [18]) are using SQL on their front end. However, SQL does not have the debugging and rapid prototyping facilities (e.g., Jupyter notebooks) provided by Python. This is one of the important reasons behind the popularity of Pandas among data scientists.

In this chapter, we propose the first embedded query engine in Python that employs query compilation with support for the essential physical query operators required for

analytical workloads. By having a data-centric approach, this engine is able to apply global optimizations (e.g. vertical fusion) and generate high-performance code for the queries. Our framework provides a JIT-able Python library based on semi-ring dictionaries [78]. Our framework runs the Python code of all TPC-H queries competitive to or faster than the state-of-the-art engines in single- and multi-threaded settings.

The contributions of this chapter are as follows:

- We implement SDQL.py, an efficient and embedded query engine in Python. Our engine exposes a JIT-able library that is based on semi-ring dictionaries (Section 4.2), which makes it expressive enough to support the query operators that are essential for the efficient processing of analytical queries. SDQL.py enables the developers to do the prototyping and debugging in Python, yet easily build a compiled production-ready version of their code for deployment (Section 4.3). We also make our implementation open-source and available.[1]
- We present our effort on the integration of PyTond (cf. Chapter 3) into SDQL.py that offers the source-to-source compilation of Python data science to C++ (Section 4.6).
- We show the applicable optimizations on our IR and also on the generated code that are required to fully exploit the benefits of query compilation (Sections 4.4 and 4.5). The data-structure specialization is required for the efficient processing of semi-ring dictionaries in hash joins. Also, we present the different parallelization strategies for semi-ring dictionaries. Moreover, we show how specialized code generation for specific types of iterations can result in better performance.
- We experimentally evaluate the performance of our framework on the entire set of TPC-H queries. Our results show that our framework significantly outperforms Pandas and is faster than or competitive with the state-of-the-art in-memory engines in both single- and multi-threaded settings. Furthermore, we investigate the impact of individual optimizations in detail (Section 4.7).

4.2 Background on SDQL IR

Query Compilation. Traditional database systems use pre-compiled kernels (for each query operator) to process the data on a tuple-at-a-time basis. This has been sufficient for a long time for out-of-core scenarios. However, motivated by the recent advances in hardware technology, this design was proved to be suboptimal for in-memory

[1] https://github.com/edin-dal/sdqlpy

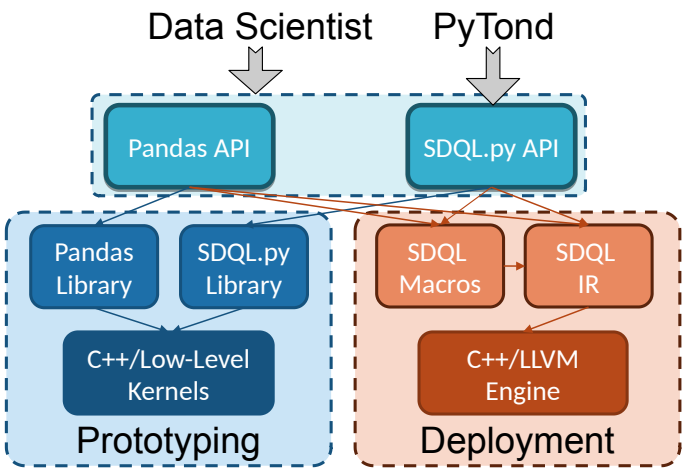


Figure 4.1: The workflow of SDQL.py

databases. Query compilers [49, 60, 79, 80, 46, 62] resolve this issue by compiling a query plan entirely and exploiting global optimization opportunities and hardware-specific features to offer higher performance. Although there are some specific-purpose compiled kernels or JIT-able scientific libraries for Python (cf. 6), to the best of our knowledge, there is no approach offering embedded query compilation in this language.

Semi-ring Structures. A semi-ring structure is defined over a data type with two binary operators + and *, with the identity elements 0 and 1 for each. Four examples of semi-ring structures are listed in Table 4.1.

Table 4.1: Examples of Semi-Ring Structures

Name	Type	Domain	Addition	Multiplication	Zero	One	Ring
Real	Sum-Product	$\mathbb{R}$	$+$	$\times$	0	1	✓
Integer	Sum-Product	$\mathbb{Z}$	$+$	$\times$	0	1	✓
Min-Product	mnpr	$(0,\infty)$	min	$\times$	∞	1	$\times$
Boolean	bool	$\{T,F\}$	$\vee$	$\wedge$	false	true	$\times$

Semi-ring dictionaries. If the value-type of a dictionary forms a semi-ring structure, then the dictionary also forms a semi-ring structure. In this case, the addition is point-wise meaning that values with the same key are added. Moreover, zero-valued pairs are implicit and can be removed from the dictionary. When multiplying a scalar and a semi-ring dictionary, the scalar value will be multiplied by the values of each pair in the dictionary. The following example from [78] clarifies additions and multiplications

over semi-ring dictionaries. Consider the dictionaries:

$$\text{dict1} = \{\text{a} \to 2, \text{b} \to 3\}, \quad \text{dict2} = \{\text{a} \to 4, \text{c} \to 5\}$$

The result of dict1 + dict2 is:

$$\{\text{a} \to 6, \text{b} \to 3, \text{c} \to 5\}$$

This is because:

$$\text{dict1} = \{\text{a} \to 2, \text{b} \to 3, \text{c} \to 0\}, \quad \text{dict2} = \{\text{a} \to 4, \text{b} \to 0, \text{c} \to 5\}$$

Element-wise addition results in:

$$\{a \to 2+4, b \to 3+0, c \to 0+5\} = \{a \to 6, b \to 3, c \to 5\}$$

Moreover, the result of dict1 * dict2 is:

$$\{a \to 2 \cdot \text{dict2}, b \to 3 \cdot \text{dict2}\}$$

The expression $2 \cdot \text{dict2}$ evaluates to:

$$\{a \to 2 \cdot 4, c \to 2 \cdot 5\}$$

By performing similar computations, dict1 * dict2 evaluates to:

$$\{a \to \{a \to 8, c \to 10\}, b \to \{a \to 12, c \to 15\}\}$$

On the other hand, dict2 * dict1 is:

$$\{a \to 4 \cdot \text{dict1}, c \to 5 \cdot \text{dict1}\}$$

After performing similar computations, the expression evaluates to:

$$\{a \to \{a \to 8, b \to 12\}, c \to \{a \to 10, b \to 15\}\}$$

Real-, Boolean-, and Integer-valued semi-ring dictionaries represent real vectors, relations with set semantics, and relations with bag semantics, respectively. Semi-ring dictionaries themselves form a semi-ring structure. Thus, we can nest them; for example, a matrix can be represented as a semi-ring dictionary with a vector (Real-valued semi-ring dictionary) as its value [78]. A set of algebraic rules including commutativity, associativity, and distributivity are valid for semi-ring structures.

SDQL. The intermediate representation of SDQL.py is based on SDQL [78], a simple, powerful, and expressive language that is centered around semi-ring dictionaries. SDQL is capable of capturing a wide spectrum of data analytics workloads including relational, nested relational, and linear algebra. In this chapter, we are focused on relational workloads to propose a full-fledged approach for all of the query operators and left the support of other/hybrid workloads for future work.

Queries in SDQL. Each query is represented as a chain of iterations over semi-ring dictionaries, referred to as *summations* in SDQL. Thanks to the algebraic nature of semi-ring dictionaries and by applying the associativity rule, iterations over unordered datasets are made possible. During each run of a summation, based on the logic of the code, a value is calculated and then aggregated into an output data structure. The result of aggregation operators also has a semi-ring structure. Different associative operations can be used for aggregation. However, to cover the relational queries, which is the focus of this work, we only need dictionary addition, scalar addition, and min/max operations.

Example. Suppose that we have two relations `R(k, a, b, c, d)` and `S(e, f, g, k)`. Assume a query that performs a selection and projection on each of them and then joins them based on `k`, which is a primary key in `R`. The corresponding SDQL code for the pipeline of these operators is as follows:

```
let R_filt = sum(r in R) if r.a==9 then {r -> true}
let R_proj = sum(r in R_filt) {<k=r.k, c=r.c> -> true}
let R_indx = sum(r in R_proj) {r.k -> r.c}
let S_filt = sum(s in S) if s.e=="pass" then {s -> true}
let S_proj = sum(s in S_filt) {<k=s.k, g=s.g> -> true}
sum(s in S_proj) if R_indx(s.k) then
    {<c=R_indx(s.k).c, g=s.g> -> true}
```

For each relation, we do a selection, and then a projection. After each of these operations, we materialize the results in an intermediate dictionary. Using the materialized result of `R`, we build the join index (`R_indx`). Then, by iterating over the materialized result of `S`, we compute the final result of the join operation.

Pipelined Example. Having separated summations and intermediate results will result in poor performance of the code. Thus, by optimizing the code using the data-centric computation (a push-based pipelined engine [60]), we have the following code:

```
let R_indx = sum(r in R) if r.a==9 then {r.k -> r.c}
sum(s in S) if s.e=="pass" and R_indx[s.k] then
    {<c: R_indx[s.k].c, g: s.g> -> true}
```

In this optimized version of the code, the projections and selections are vertically

Table 4.2: The API provided by SDQL.py.

`{ k: v, ... }`	Dictionary creation
`record({ "a": e, ... })`	Record creation
`D[k]`	Dictionary lookup
`rec.a`	Record field access
`R.sum(lambda x: e)`	Summation over `R`
`R.sum(lambda x: e2 if e1 else e3)`	Filtered summation over `R`
`R.partition()`	Partition `R`
`R.psum(lambda i, x: e)`	Parallel summation over `R`
`r1.concat(r2)`	Record concatenation
`unique(rec.a)`	Field with unique domain
`dense(rec.a, low_bnd, up_bnd)`	Field with dense domain

fused [47] with index building in the first summation, and index probing in the second one.

Extensions over SDQL. Even though this work, heavily relies on SDQL IR [78], it has the following extensions:

- It offers a Python library that enables developers to (1) use their Python programming skills and rely on IDE and debugging facilities provided by Python, and (2) exploit global optimization opportunities. In other words, we elevated the SDQL IR to a high-level DSL in Python.

- SDQL.py generates parallel C++ code, as opposed to the SDQL's single-threaded code generation. This is achieved by introducing new constructs at both library and IR levels, which makes it possible to support different parallelization strategies. Furthermore, SDQL.py performs data-structure specialization and leverages new code generation rules to improve the performance of semi-ring dictionaries.

- SDQL.py implements the entire set of query operators needed for all TPC-H queries, which goes beyond the subset of queries tested for SDQL [78], Weld, and open-source implementations of Hyper and Vectorwise [47].

4.3 Design Overview

In this section, we discuss the architecture of our system, the overview of which is shown in Table 4.1. Data scientists can write their queries in Python using the API of

SDQL.py or Pandas. The input to the system can also be fed automatically by PyTond, as will be discussed in Section 4.6. Then, depending on the execution mode, the code is redirected to an interpreter- or compiler-based pipeline.

SDQL.py API. The relational queries are written in Python language while using the constructs and APIs introduced in our Python library, named SDQL.py. A major part of the program logic can be defined using the Python standard syntax. From a data-type point of view, Integers, Doubles, Strings, Boolean, and even dictionaries are exactly the ones provided by Python. We only provide the `record` data type to encapsulate our definition of tuples. There is only a limited set of APIs (in a purely functional style) that the developer must use to be compatible with our code generation pipeline. The most important ones are `read_csv` and `sum`. The first one is used to import the data sets and the second one is to define SDQL summations over semi-ring dictionaries.

The queries written using SDQL.py API must be defined in terms of a Python function and annotated using the Python decorator `@sdql_compile`. This annotation accepts three arguments: (1) the type of inputs and (2) the execution mode and (3) the flag for auto parallelization. The previous pipelined example in Section 4.2 is expressed as the following Python function:

```
@sdql_compile(
{"R": R_type, "S": S_type}, exec_mode, parallelize)
def query(R, S):
  R_indx = R.sum(lambda r:
    {r[0].k: r[0].c} if r[0].a==9 else {})
  return S.sum(lambda s:
    {record({"c": R_indx[s[0].k].c, "g": s[0].g}): True})
    if (s[0].e=="pass" and R_indx[s[0].k]) else {})
```

Execution Modes. The execution mode is set by passing an argument to the function's annotation. There are two execution modes in our approach: (1) prototyping and (2) deployment. For prototyping, the developer's code will be interpreted in Python. This mode is slow and should only be used for debugging purposes. On the other hand, in the deployment mode, a compiled version of the Python code will be generated to be used in production.

Auto Parallelization. The third argument to the SDQL.py function decorators is a flag that enables auto parallelization for the body of the function. In Sections 4.4 and 4.5, we will explain the design decisions for auto parallelization of summations and also its related code generation patterns.

SDQL IR Generator. This component parses the input Python code using the `ast` package of the Python ecosystem. Then, it uses the parsed Abstract Syntax Tree (AST)

to generate the relevant SDQL IR. In addition, there are extra functions provided by the SDQL.py API (e.g., `joinBuild`, `joinProbe`), referred to as *SDQL Macros*. These will be expanded to the core IR of SDQL.

Pandas to SDQL IR Translation. To improve the usability of our framework, we have provided a subset of Pandas API which seamlessly calls the underlying Pandas library in prototyping mode or lazily constructs SDQL Macros & IR in the deployment mode. The Pandas API however is not as expressive as the SDQL.py and optimizations like Data-Structure Specialization and Assignment Summations (that will be discussed in Sections 4.4 and 4.5 respectively) cannot be expressed using this API.

Type Inference. Similarly to Numba [50] and JAX [39], and as opposed to Tuplex [85], we consider a statically typed program as input. We use the type information of the inputs (provided as the first argument to `@sdql_compile`) to perform type inference over the entire SDQL IR.

C++ Code Generator. This component uses the inferred type for all program expressions to generate their optimized equivalent in C++. There are two different versions of the code generator. One is optimized for single-threaded, while the other one fits multi-threaded scenarios. The generated code will be compiled and installed on the production machine as a specialized Python package.

4.4 Efficiency

In this section, we explain the rationale behind our design decisions toward optimizing the proposed approach.

4.4.1 Data Structure Specialization

Motivating Example. By profiling the generated code for TPC-H queries, we found inefficiencies for hash table construction in the build phase of the hash join. For example, the generated C++ code for the build phase of the last hash join in TPC-H Q9 is as follows:

```
long v1 = db["orders"].size();
hash_table<long, long> v2;
for(size_t v3=0; v3<v1; ++v3)
    v2.emplace(o_orderkey[v3], o_orderdate[v3]);
```

This code creates a hash table with the primary key of the data set as its key without applying any selection on the tuples (i.e., it is highly selective). The high selectivity

of this operation makes the key domain very dense. In other words, the output hash table contains all of the valid values for `o_orderkey`. In such cases, it is more efficient to use a native array, referred to as *dense array*, instead of a hash table. This choice, as also discussed in [25], will improve both the insertions and lookups on the key. The improved version of the previous code is as below:

```
long v1 = db["orders"].size();
auto& v2 = std::vector<long>(6000000);
for(size_t v3=0; v3<v1; ++v3)
    v2[o_orderkey[v3]-lower] = o_orderdate[v3];
```

The reserved size for a dense array equals to the domain size of the key attribute (`upper-lower+1`). This information can be obtained from the database catalog during query compilation. To insert/lookup a key in this simplified hash table, we need to first calculate `key-lower` to find the real index of the key in the array. Then, we have fast random access to its related value.

Transformation. SDQL provides facilities to represent dense arrays. Semi-ring dictionaries with a key type of dense integer (`dense_int`) correspond to dense arrays [70]. In SDQL.py we extend the type system of SDQL to encode the bounds of a dense integer (as `dense_int[LOW_BND, UP_BND]`). The transformation in terms of SDQL.py is as follows:

```
d.sum(lambda p: {k: v})
```

```
d.sum(lambda p: {dense(k, lower, upper): v})
```

Impact on Performance. In Section 4.7, we will cover the micro-benchmark results of using dense arrays in different scenarios and show its impact on the performance.

4.4.2 Parallelization

In this Section, we cover the challenges of generating parallel code out of SDQL IR. Since we target relational data analytics, the main technique we employ is data parallelization.

Motivation. SDQL IR is built around the notion of semi-ring dictionaries. Thus, we need a hash table as the output of a majority of the iterations (summations). The performance of the selected hash-table implementation directly affects the overall performance of the query engine. This is even more important for parallel execution.

Table 4.3: Examined Hash-Table Implementations.

Sequential	Parallel
std::unordered_map	tbb::concurrent_unordered_map [8]
ska::flat_hash_map [7]	tbb::concurrent_hash_map [8]
phmap::flat_hash_map [10]	phmap::parallel_flat_hash_map [10]
robin_hood::flat_hash_map [11]	libcuckoo::cuckoohash_map [52]
tsl::robin_map [13]	growt (Default config) [55]
tsl::hopscotch_map [12]	folly::AtomicUnorderedMap [9]

Sequential/Parallel Hash Tables. We conducted a broad exploration of the existing research literature and open-source projects on sequential and parallel hash tables to select the best one. Table 4.3 shows a list of hash-table implementations that we examined. Among the sequential implementations, based on the performance of insertion and lookups, we selected `phmap::flat_hash_map`. For the parallel ones, we selected `phmap::parallel_flat_hash_map` and `libcuckoo::cuckoohash_map`. This is because of (1) their better performance on parallel insertion and lookups and (2) their support for struct types as keys and values.

Summation Patterns. As explained in Section 4.2, in SDQL, all query operators are expressed as iterations (summations) over semi-ring dictionaries (hash tables or dense arrays). Depending on the body of summation, the output is computed by adding (for a scalar output) or appending (for a dictionary output) the intermediate results together. The parallelization of the former case is trivial. We focus on the parallelization of the latter case.

Dictionary Building Patterns. The case where the output is a dictionary mainly corresponds to two patterns. First, if the keys generated in each round of iteration are unique, then the iteration is building a dictionary indexed over the input relation, corresponding to the build phase of hash join. This dictionary is then used in the following pipelines, e.g., the probe phase of hash join. Second, if the keys are duplicated, then the result is a group-by aggregate.

Parallel Dictionary Building. As the query plan operators that return a relation as output are finally converted to one of the two dictionary building patterns (cf. Section 4.2), it is important to find the best parallelization strategy for these patterns. To achieve the best design, we examined the following 3 alternatives.

(1) Single Phase (Parallel). This is the most natural approach of parallelism for iterations. In this approach, the resulting value of each round of iteration is directly inserted into the output hash table. This requires support for concurrent insertions from the output hash table. The advantage of using this design is its simplicity, which

results in less complexity for code generation. In other words, this design delegates the parallel contention management concerns to the internal design of the selected parallel hash table. The SDQL.py code related to this method of parallel hash table generation is as follows:

```
d_part = d.partition()
result = d_part.psum(lambda i, p: e)
```

In the first line, the `partition` API breaks the input data apart making disjoint sets that are assigned to different threads (`i` is the thread number). Then, the `psum` API executes a summation over the partitioned input of each thread and stores the results in a global parallel hash table.

(2) Two Phase (Parallel-Parallel). In this design, we create a local dictionary for each thread. In the first phase, each thread iterates over the assigned partition of input data and aggregates the result in its local dictionary. The second phase is responsible for merging the local dictionaries; each thread finally inserts its local data into the output dictionary. In this design, for the case of group-by aggregates, we use our selected sequential hash table (`phmap::flat_hash_map`) for the first phase. But, for the second phase, similar to the single-phase approach, we still need to have an output hash table with concurrent insertion support. The first phase of this design has an important role in minimizing the contention rate for the final insertions into the output dictionary. The SDQL.py code for this approach is shown below:

```
d_part = d.partition()
result_part = d_part.psum(lambda i, p: {i: e})
result = result_part.psum(lambda i, p: p)
```

Similar to the previous approach, the input is partitioned and fed into the first parallel summation. However, since the first `psum` generates partitioned output, another `psum` is used for the final combination of the results and spills them in the output parallel hash table.

(3) Two Phase (Parallel-Sequential). The first phase of this design is the same as the previous one. The second phase performs the merging sequentially; a single thread iterates over all thread-local dictionaries and insert their elements into the output dictionary. As the last parallelization approach, the related SDQL.py code is written below:

```
d_part = d.partition()
result_part = d_part.psum(lambda i, p: {i: e})
result = result_part.sum(lambda p: p[1])
```

The process is similar to the parallel-parallel approach, however, in the last step, aggregation of the results is done sequentially by using a `sum` API.

Impact on Performance. In Section 3.5, we will investigate the behaviour of the different parallel strategies on the performance of group-by aggregation and the build phase of hash join. We will show that the third design performs the best, which makes it our chosen strategy for parallel code generation. In practice, if the user sets the third argument of a SDQL.py function decorator (`@sdql_compile`) to `True` (see the example of this decorator in Section 4.3), SDQL.py automatically runs each summation in parallel mode using the Parallel-Sequential approach. In other words, the following transformation happens:

```
result = d.sum(lambda p: e)
```

```
d_part = d.partition()
result_part = d_part.psum(lambda i, p: {i: e})
result = result_part.sum(lambda p: p[1])
```

Summary on Parallelization. To summarize our parallelization approach we consider the formal framework proposed by Farzan et al. [36]. The `partition` construct is similar to the divide operator for the input sequence which is an array of elements. But for the join operator of this formal framework, we have different decisions based on the parallelization strategy. In the single-phase (parallel), there is no join required, as the results are directly inserted into a concurrent container. In the two-phase approaches (parallel-sequential and parallel-parallel), the join operation is the same associative operator that is used in the summation; it combines the results of threads together either sequentially (parallel-sequential) or in parallel (parallel-parallel). Since the summation operator is associative, our loops can be parallelized without any extra information from each thread.

4.5 Code Generation

In this section, we present the specific C++ code generation decisions that we made in SDQL.py.

Aggregate vs Assignment Summations. As discussed in Section 4.2, in SDQL.py `summations`, the results of each iteration are always aggregated into an output data structure. In some cases, when the output value of each iteration is a dictionary and the keys

in the generated output are unique, the aggregation can in fact be replaced with an assignment. To enable this feature in SDQL.py, the user can use the `unique` API to mark a key domain as unique. The following example shows the C++ code generated for an assignment scenario:

```
result = d.sum(lambda p: unique(p[0].a): p[0].b)
```

```
hash_table<long, long> v1;
for (size_t v2=0; v2<d.size(); ++v2)
    v1.emplace(d_a[v2], d_b[v2]);
```

This is in different from the standard translation of summation, which is as follows:

```
result = d.sum(lambda p: p[0].a: p[0].b)
```

```
hash_table<long, long> v1;
for (size_t v2=0; v2<d.size(); ++v2)
    v1[d_a[v2]] += d_b[v2];
```

Parallel Summations. Different approaches towards parallelization of summation are discussed in Section 4.4. Here we show the code generation template of each approach. We use TBB [8] as our parallelization framework.

(1) Single-Phase (Parallel) Approach. In this template, since we have a single phase for summation, the output hash table must be a parallel (concurrent) data structure. After initialization of this hash table, we partition the range of items in the input data set (`d`) using `tbb::blocked_range` API. Then, we pass the partitioned input as the first parameter of a `tbb::parallel_for` to make the parallel iteration possible. As the second argument of `tbb::parallel_for`, we pass the lambda function that each thread executes on its own partitions. The lambda iterates over the data elements and executes the inner logic of our summation (`e`). In this type of summation, the generated key/value pairs will directly be appended to the output hash table (`pht`).

```
d_part = d.partition()
result = d_part.psum(lambda i, p: e})
```

```
using k_t = e_key_type;
using v_t = e_val_type;
parallel_hash_table<k_t, v_t> pht;
tbb::parallel_for(
tbb::blocked_range<size_t>(0, d.size()),
[&](const tbb::blocked_range<size_t>& r)
{
    for (size_t i=r.begin(), end=r.end(); i!=end; ++i)
    {// C++ code of e that appends the results to pht}
});
```

(2) Two-Phase (Parallel-Parallel) Approach. The first loop of this template is almost the same as we had in the single-phase approach. But here, we need to assign a local sequential hash table to each thread. To do so, we make use of the `tbb::enumerable_thread_specific` API which creates a set of thread-specific data structures of the same type. Then, in the lambda function, we first access the data structure (our sequential hash table) using the `locals` API of `tbb::enumerable_thread_specific`. Finally, the results of each iteration will be appended to this thread-local hash table. In the next loop, which is generated for the second `psum`, using the `tbb::parallel_for_each`, again we assign the local hash tables of the previous phase to separated threads and each thread will add its own key/value pairs to the output parallel hash table.

```
d_part = d.partition()
result_part = d_part.psum(lambda i, p: {i: e})
result = result_part.psum(lambda i, p: p)
```

```
using k_t = e_key_type;
using v_t = e_val_type;
parallel_hash_table<k_t, v_t> pht;
tbb::enumerable_thread_specific<hash_table<k_t, v_t>> ts;
tbb::parallel_for(
tbb::blocked_range<size_t>(0, d.size()),
[&](const tbb::blocked_range<size_t>& r)
{
    for (size_t i=r.begin(), end=r.end(); i!=end; ++i)
    {
       auto& local = ts.local();
       // C++ code of e that appends the results to local
    }
});
tbb::parallel_for_each(ts.begin(), ts.end(),
[&](auto& local) {
   for(auto& item : local) pht[item.first] += item.second
});
```

(3) Two-Phase (Parallel-Sequential) Approach. As the last method of code generation for parallel summation over semi-ring dictionaries, we generate a code similar to the previous approach for the first `psum`. However, since the second phase is a sequential phase, the output hash table (`ht`) is a sequential hash table. Also, the second summation is a non-parallel summation and its C++ code will be a sequential for-loop over each thread-local container.

```
d_part = d.partition()
result_part = d_part.psum(lambda i, p: {i: e})
result = result_part.sum(lambda p: p[1])
```

```
using k_t = e_key_type;
using v_t = e_val_type;
hash_table<k_t, v_t> ht;
tbb::enumerable_thread_specific<hash_table<k_t, v_t>> ts;
tbb::parallel_for(
tbb::blocked_range<size_t>(0, d.size()),
[&](const tbb::blocked_range<size_t>& r)
{
    for (size_t i=r.begin(), end=r.end(); i!=end; ++i)
    {
        auto& local = ts.local();
        //C++ code of e that appends the results to local
    }
});
for(auto& local : ts)for(auto& item : local) ht += local;
```

4.6 Integration to PyTond

As we briefly mentioned in Section 4.1, the input to our system can be an automatically generated SDQL.py code from PyTond (cf. Chapter 3). In this section, we discuss the integration of PyTond, our data science to SQL compiler, and SDQL.py. By integrating these two systems, a Python data science workload can be transformed into an efficient C++ code. In other words, we can craft a specialized engine for that specific data science workload.

To achieve this goal, as we presented in the thesis introduction (Figure 1.1), we first take the generated SQL by PyTond. Then, instead of passing it to the query engine for execution, we only use the query planner of the RDBMS by taking its generated query plan for the given workload. Afterward, we apply a set of optimizations to the query

plan to make it efficient. Finally, we unparse the plan and generate SDQL.py code. The optimizations we apply to the query plan are briefly discussed below:

Loop Fusion. By adopting the approach of push-based engines [60], we first identify the pipeline breakers in the given query plan. Then, we fuse subsequent summations until reaching a breaker. This global optimization eliminates the creation of unnecessary intermediate results and allows materialization only when it is needed.

Aggregate to Assignment Summation Conversion. The nature of this local optimization is discussed in 4.5. Here, since we have access to the database catalog information, we check if the keys for a summation output dictionary are unique and enable the relevant flag in the generated SDQL.py code.

Data Structure Specialization. This local optimization is discussed in 4.4. Here, since we have access to the cardinality information in the given query plan, we can decide to use a dense array instead of a hash table, if the key part of the output dictionary is dense enough (e.g. selectivity more than 90%).

Dead Code Elimination. This global optimization prevents code generation and materialization for the columns that exist in the input relations but are not used in the given query plan.

4.7 Evaluation

In this section, we experimentally evaluate the effectiveness of SDQL.py on both micro benchmarks and TPC-H queries.

4.7.1 Experimental Setup

All experiments are conducted on a single machine equipped with 16GB of DDR4 RAM, and an Intel Core i5-10210U 1.6GHz with 8 threads and 256KB, 1MB, and 6MB of L1, L2, and L3 cache respectively. Hyper-threading is enabled. We have used Ubuntu 20.04.3 as OS. Our C++ code is compiled with G++ 9.4.0 using the -O3 flag. We use Python 3.8.10.

Workloads. To show the performance of our approach on analytical workloads, we use the well-known TPC-H benchmark with a scaling factor (SF) of 1. In the implementation of the queries, we adopt the query plans from Hyper [60]. Similar to Typer [47], we remove the `LIMIT` and `ORDER BY` operators from TPC-H queries before running them in each of the evaluated database systems. Moreover, in our implementation of

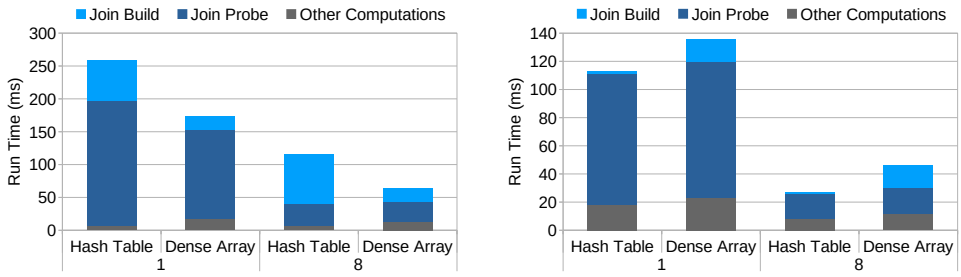


Figure 4.2: Comparing the performance of using hash tables and dense arrays for a high- (left) and a low-selectivity (right) join build in TPC-H Q9 on 1 and 8 threads.

Hyper query plans, we replace the *indexed nested-loop join* with *hash join* by building the required indices in runtime. In all of the experiments, we excluded compilation time. In code generation, there is always a risk of having a larger compilation time than the execution time of the workload. However, similar to [47], our focus in this chapter is on OLAP queries and the compilation time for these workloads can be ignored as the code can be generated ahead of time and executed many times afterward.

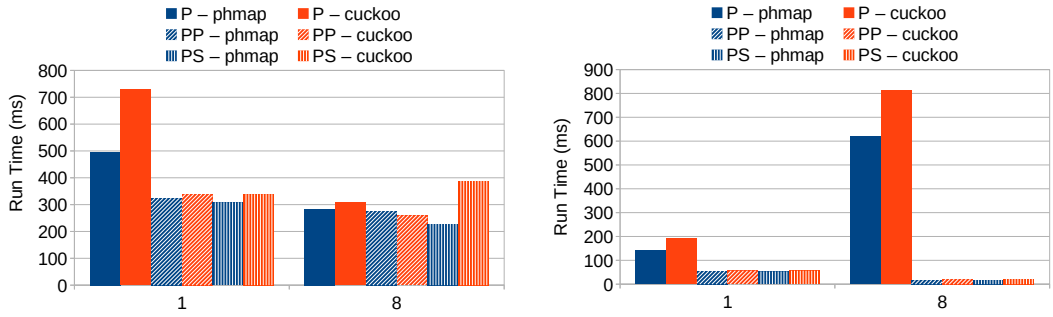
Competitors. In the experiments, we compare the performance of our approach with two state-of-the-art query engines: Hyper [60] and DuckDB [66]. Hyper is a column-store in-memory JIT-compiled query engine and we use its commercial closed-sourced version HyPer (v0.4-452-g9dabe83) since its up-to-date binary is no longer available. DuckDB is a column-store vectorized embedded DBMS and we use version 0.3.4 of it. Both of these systems support multi-threaded processing and it makes them a great baseline for the experiments. We also performed experiments on Pandas 1.4.1 to compare our work with this popular Python library.

4.7.2 Micro-Benchmarks

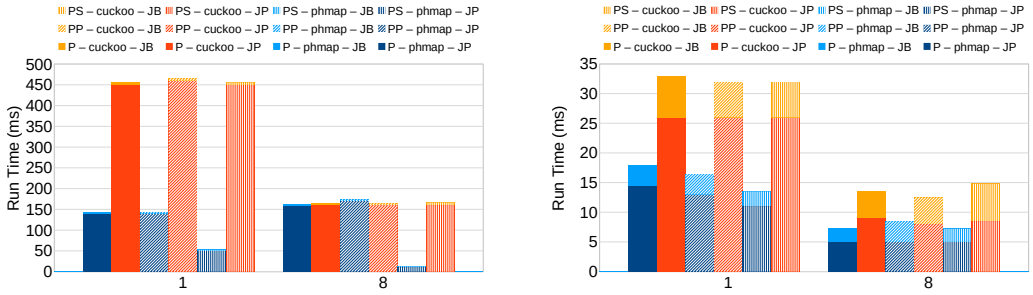
In this section, we present the results of micro-benchmarks for the optimizations discussed in Section 4.4.

Data Structure Specialization. To show the impact of this optimization, we use the last join of TPC-H Q9, which is the dominating operator of this query. Figure 4.2 depicts the performance results of both the join build and probe phases with and without data-structure specialization.

The left chart shows the run times of the standard query, which has a high selectivity (100%) for the build phase on the `orders` data set. In this case, using dense array results in a 1.5× and 1.8× speedup for single- and eight-threaded settings. However,



(a) Group-by aggregate in Q1. # of groups: (left) large (right) small



(b) Left-semi-hash join in Q14. # of probes: (left) large (right) small

Figure 4.3: Performance results of different parallelization strategies on 1 and 8 threads. (P) Single Phase, (PP) Two Phase – Parallel-Parallel, (PS) Two Phase – Parallel-Sequential.

we observe a negligible memory (de-)allocation overhead for the dense array.

The right chart corresponds to a slightly modified version of the query with a low selectivity (0.1%) on the same join build. As opposed to the previous case, the memory management overhead of dense arrays is no longer justified for the sparse key domain. Thus, in this case, the hash table implementation shows a 1.2× speedup on a single thread and 1.7× speedup for eight threads.

Parallelization of Group-By Aggregation. In Figure 4.3a, we study the performance of TPC-H Q1 by altering the parallelization approach for its group-by aggregate operator. The SDQL.py code for Q1 is shown here:

```
li_aggr = li.sum(lambda p:
{
    record({"l_returnflag": p[0].l_returnflag,
            "l_linestatus": p[0].l_linestatus}):
    record({"sum_qty": p[0].l_quantity,
            "sum_base_price": p[0].l_extendedprice,
            "sum_disc_price":
    (p[0].l_extendedprice * (1.0 - p[0].l_discount)),
            "sum_charge":
```

```
    ((p[0].l_extendedprice * (1.0 - p[0].l_discount)) *
    (1.0 + p[0].l_tax)),
            "count_order": 1
    })
}
if p[0].l_shipdate <= 19980902 else {})
result = li_aggr.sum(lambda p: {p[0].concat(p[1]): True})
```

As the above code shows, Q1 includes a single dictionary construction. We use two parallel hash table implementations for this dictionary (previously selected in Section 4.4.2) and run the experiments on one and eight threads. The left chart is related to a slightly modified version of the query which results in a larger number of groups (200K) while the right chart shows the run time for the standard query with only 4 groups in output.

We observe that the single-phase approach has the worst performance on both one and eight threads. On a single thread, the internal overhead of concurrent hash tables degrades the performance of this approach, and on eight threads, the increasing contention of parallel insert operations in the hash table results in poor performance, especially in the right chart with a higher contention rate. Regardless of the number of groups, the run times of the two-phase approaches are very close and the parallel-sequential variant performs slightly better than the parallel-parallel one with an average speed-up of 1.1× on eight threads and 5% improvement on a single thread for both scenarios.

Parallelization of Join Build. In Figure 4.3b, we study the impact of using different parallelization approaches on the build phase of the left-semi-hash join operator in TPC-H Q14. The benchmark is almost similar to the previous one shown in Figure 4.3a except that it also shows the impact of the changes on the join probe phase. The SDQL.py code for this query is as follows:

```
pa_indx = pa.sum(lambda p: {unique(p[0].p_partkey): True}
            if startsWith(p[0].p_type, "PROMO") else {})
li_prob = li.sum(lambda p: record({
    "A": p[0].l_extendedprice * (1.0 - p[0].l_discount)
        if pa_indexed[p[0].l_partkey] != None else 0.0,
    "B": p[0].l_extendedprice * (1.0 - p[0].l_discount)})
if p[0].l_shipdate >= 19950901 and
   p[0].l_shipdate < 19951001 else {})
result = (100.0 * li_probed.A) / li_probed.B
```

As the above code shows, TPC-H Q14 has two pipelines: (1) a join build, and (2) a join probe followed by a scalar aggregation. The probe phase lookups the values in the join hash table and can be easily parallelized in a single-phased fashion. It outputs

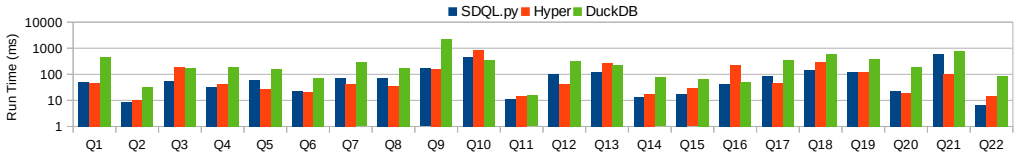


Figure 4.4: Performance results for TPC-H queries on a single-threaded setting.

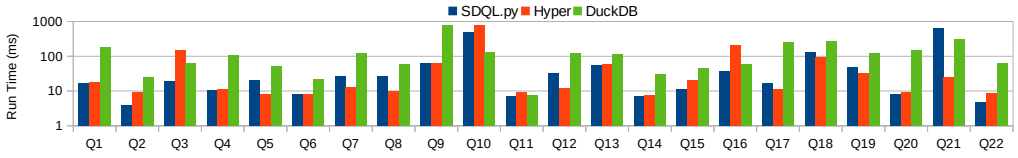


Figure 4.5: Performance results for TPC-H queries on 8 threads.

a scalar result and thus, it is trivial to be parallelized.

The right chart in Figure 4.3b is based on the standard Q14 with $60K$ join probe lookups while the left one depicts the results for a modified version of Q14 with $6M$ lookups. Similar to the results of aggregations in Figure 4.3a, the `cuckoo` hash table has an overall higher run time in comparison with `phmap`. Focusing on the `phmap` results, we see that its performance for different approaches to join build is almost the same. However, as the left chart shows, for a larger number of probes, the sequential version of `phmap` performs better. This is the reason behind the superior performance of the parallel-sequential approach.

Take-Away Message for Parallelization. By summarizing our observations on the results of Figures 4.3a and 4.3b, we decided to move forward with the two-phase parallel-sequential approach for parallelizing the group-by aggregates and hash join operators. As the output data structure of the second phase, we choose `phmap::flat_hash_map`, a sequential hash table. On the other hand, to generate a simpler sequential version of the code for single-threaded settings, we use a single-phase approach only by using the same hash table to eliminate any overhead introduced by the parallelization strategies.

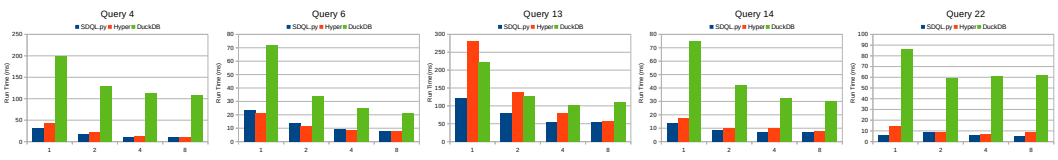


Figure 4.6: Scalability evaluation of TPC-H Q4, Q6, Q13, Q14, and Q22 on 1, 2, 4, and 8 threads.

4.7.3 Single-Threaded Performance

In this section, we show the end-to-end performance of SDQL.py. Figure 4.4 depicts the performance of SDQL.py, Hyper, and DuckDB for all TPC-H queries on a single thread. We observe that SDQL.py supports all TPC-H queries with a competitive performance in comparison to Hyper; the average speedup over all queries is 43%. Furthermore, we observe a better performance across all queries in comparison with DuckDB; the average speedup is 3.5×. As mentioned at the beginning of section 4.7, the binary of Hyper is no longer available and its code generation approach is not clear to us. However, based on an open-source version of Hyper for a subset of TPCH queries [47], the performance differences can be attributed to the efficiency of the underlying hash table. Hyper, in that version, uses a specialized parallel hash table with chaining. And we use the sequential version of `phmap` with an open-addressing collision resolution approach. The combination of this selected hash table and the two-phase (parallel-sequential) approach makes our approach faster than Hyper. In comparison with DuckDB, our performance gain comes from the pre-studied advantages of using compiled (push-based) engines instead of vectorized engines for OLAP workloads [47].

4.7.4 Multi-Threaded Performance

Figure 4.5 shows the performance of query engines in running TPC-H queries on eight threads. Similar to the single-threaded results, SDQL.py outperforms DuckDB with an average speedup of 1.85×. Although SDQL.py is faster than Hyper in most queries, we observe an average 5.5% slow-down. This is related to Q21 which contains correlated subqueries that result in nested dictionaries. The data layout design for nested dictionaries is not fast enough to make the queries with secondary indexes (e.g. Q21) competitive to the alternative approaches. By excluding this query, SDQL.py is on average 1.5× faster than Hyper.

And finally, in Figure 4.6, we evaluate the scalability of the parallelization in SDQL.py by running 5 TPC-H queries on an increasing number of threads. Similar to [86], we pick five queries that represent aggregates and join variants. It is shown that the performance of SDQL.py is on par with or better than the competitors, and it smoothly scales with respect to the number of threads. Moreover, we observe that the generated sequential code shows a better performance for a single-threaded setting, e.g., in Q22. In this query, the performance-critical part is the build phase of a join that

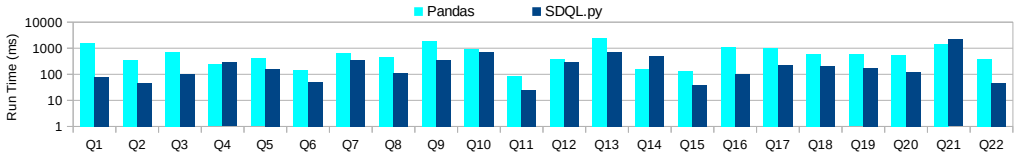


Figure 4.7: Performance results for TPC-H queries in Pandas and SDQL.py on 1 thread.

stores the intermediate results in a dense array (cf. Section 4.4.1). On 2 threads, the overheads of parallelization are higher than its benefits. By using more than 2 threads, the parallelization overhead is amortized and SDQL.py starts to improve again. Furthermore, in Q14, the build side of the join (which is a hash table) is small enough to fit into the L3 cache. As the memory access is critical for this query, the cache misses are very low and our optimized code runs fast even with the lower number of threads.

4.7.5 PyTond Integration Performance

Based on the integration discussion in Section 4.6, in Figure 4.7, we present an end-to-end benchmark that compares the performance of Pandas and SDQL.py. To perform this benchmark, we first converted the Pandas equivalent of TPC-H queries to SQL (using PyTond). Then, we used the generated query plan to generate optimized SDQL.py code. The results show that our approach can make Pandas workloads 2.36× faster on a single thread.

Chapter 5

Efficient Query Processing with Vectorized Hash Tables

In this chapter, we propose Vec-HT, an efficient batch hash table that is a useful data structure in different domains including the query engine design. We also discuss how to generate vectorized query engines by integrating this data structure to SDQL.py (cf. P3 in Figure 1.1). Most of the contents in this chapter are previously published [75].

5.1 Introduction

Hash tables are one of the most important data structures in programming. They are widely used in high-performance analytics workloads including database query processing, sparse linear algebra, graph processing, and computer networks. Besides the great efforts in algorithmic improvement of hash tables [30, 61, 65], the recent advances in modern processors, further motivated the research on high-performance hash tables that leverage the hardware characteristics including parallelization, prefetching, and vectorization.

Previous research [64] has shown that batch operations (e.g., batch lookups) on a hash table result in higher performance in comparison with ordinary scalar-parameter operations. This is because of the improved cache locality and the freedom given to the hash table designer for hand-tuning the code, which is not available when dealing with ordinary scalar-parameter operations over hash tables. Thus, hash tables with batch operations have gained more attention; both research [35, 53, 64, 71, 68, 88] and open-source projects [2, 4, 5] have proposed hash table designs with the support for batch lookups, insertions, and deletions.

The current literature on batch hash tables considers the following hardware features:

- Parallelization: The batch of inputs is divided into separate partitions that are processed in parallel [2, 64].
- Prefetching: The memory prefetching feature of processors is used to hide the latency of frequent memory accesses for a group of elements [2, 3, 29, 71].
- Vectorization: The Single Instruction, Multiple Data (SIMD) instructions of the processing unit are used for faster processing of a vector of inputs [2, 3, 29, 64, 71].

Although the existing approaches made noticeable efforts on improving the performance of batch hash tables, none of them fully exploits all of the three optimization

dimensions mentioned above (cf. Section 5.2). The existing approaches for vectorization can be categorized into two classes (cf. Figure 5.1): (1) *Horizontal Vectorization*, where the SIMD instructions are used for the operations on a single input over multiple hash table entries [2, 29, 71], and (2) *Vertical Vectorization*, where the SIMD instructions are used for the operations on an input batch over single hash table entries [64]. The first approach is not inherently batch based; it can be applied to ordinary hash tables. By conducting intensive benchmarks, Polychroniou et al. [64] and Shankar et al. [83] have shown that vertical vectorization is faster than the horizontal approach. However, prefetching has only been considered for horizontal vectorization [2, 71], and the only existing implementation of the vertically vectorized approach [64] does not support prefetching.

This chapter makes the following contributions:

- We present Vec-HT, the first batch hash table that is fully optimized in all three dimensions; Vec-HT is a multi-threaded, vertically-vectorized, and prefetching-enabled hash table that can be used for high-performance data analytics. We explain the architecture and the high-level batch API of Vec-HT in Section 5.3.
- Previous research has shown that in most high-performance use cases, optimizing the lookup performance is more important than the other operations (Section 2). Thus, the focus of this research is on batch lookups. We show the design decisions, the applicable optimizations, and the way we combine them for efficient batch lookups in Section 5.4.
- We consider several data analytics tasks that can benefit from batch hash tables, such as hash-join in query processing, set processing, and sparse vector operations (Section 5.5).
- We present the implementation challenges we faced (e.g., memory management and parallel iteration) and how we addressed them in Section 5.6.
- We showcase our design to vectorize SDQL.py join probes using Vec-HT. (Section 5.7).
- We experimentally evaluate (Section 5.8) the performance of our hash table on a set of micro benchmarks across different use case domains. Our results show that Vec-HT outperforms the state-of-the-art batch and non-batch hash tables.

Approach	Parallelization	Prefetching	SIMD Vectorization
Hirola [4], R hashmap [5]	○	○	○
DPDK [2]	●	●	◐ **(H)**
Cuckoo++ [71]	○	●	◐ **(H)**
Polychroniou et al. [64]	●	○	●
Vec-HT (this work)	●	●	●

Table 5.1: A summary of state-of-the-art batch lookups in hash tables. H: Horizontal Vectorization.

5.2 Background

In this section, we introduce the main concepts and techniques for building high-performance hash tables while summarizing the previous efforts in this area of research. Table 5.1 presents a summary of the state-of-the-art in batch hash tables.[1] To position the contributions of this chapter, our approach is also appended to the table.

Batch Hash Tables. Batch Hash Tables accept a vector of keys or key/value pairs as their API arguments and do the operation on the inputs in batch. By taking batch inputs, the batch hash tables benefit from the cache locality and are also amenable to the use of Single Instruction, Multiple Data (SIMD), parallelism, and prefetching. The SIMD is a hardware feature that allows the simultaneous execution of an operation on a vector of values. On the other hand, prefetching is a hardware feature that allows the program to request future memory accesses in advance and asynchronous to the other computations. We will cover the more-detailed definitions of these two concepts later in this section.

SIMD-Aware Batch Hash Tables. To use SIMD features of a CPU in an operation (logical, arithmetic, memory, etc.), we first need to construct a vector of operands that fit the CPU register size. Then, the prepared register could concurrently process the data by using SIMD instructions. Modern CPUs offer even more advanced SIMD instructions such as selective load/store and scatter/gathers. Selective load/store makes the parallel optional load/store from/to a contiguous memory location possible by accepting a mask register. The gather/scatter operations provide the ability to load/write from/into different parts of the memory in parallel. At the time of writing this chapter, the SIMD scatter is not widely adopted, and is only provided by specific hardware (e.g., Intel Xeon Phi).

[1] Although in the literature the terms batch and vectorized are used interchangeably, for the sake of consistency, in this chapter, we use the term batch to refer to a collection of elements, and vectorized to refer to SIMD vectors.

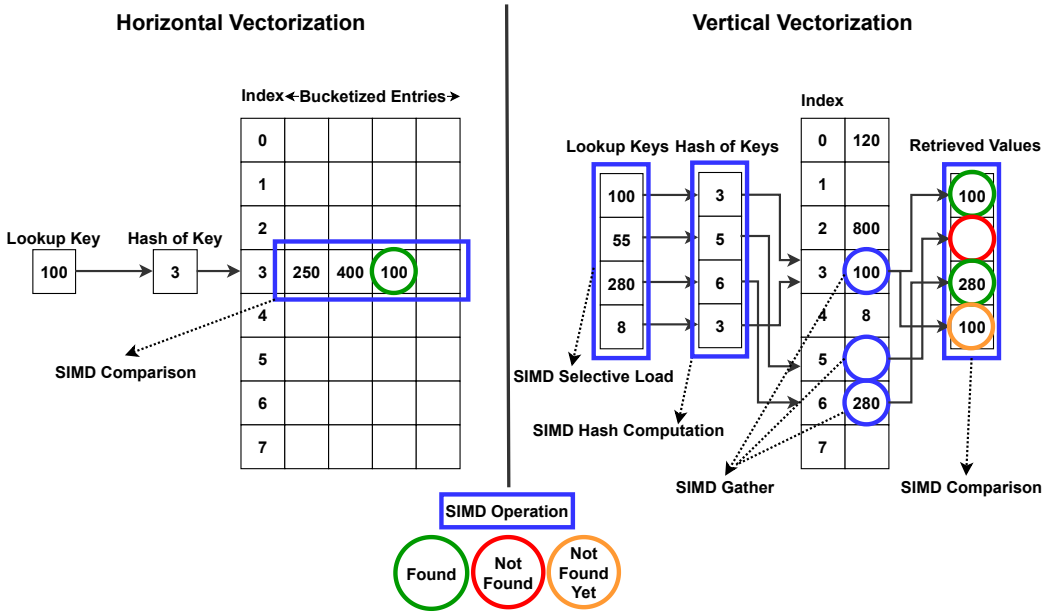


Figure 5.1: The comparison between horizontal and vertical vectorization.

As mentioned earlier, SIMD techniques can be used to improve the performance of batch hash tables. In high-level, the usage of SIMD in hash tables could be categorized in two: (1) Horizontal Vectorization and (2) Vertical Vectorization. A simplified visualization of these two approaches is depicted in Figure 5.1.

In horizontal vectorization, each cell of the hash-table entries array is bucketed into N inner cells. Then, while doing a lookup on the hash table and after computing the hash value, using the SIMD logical operations, the lookup algorithm can concurrently check the value of N bucket keys. This is much faster than having only one key per cell. By using this approach the hash table load factor can be improved without increasing the average lookup time. Regardless of the mentioned benefits of horizontal vectorization, it is wasteful if we expect to look up fewer than N buckets on average per probing key [64]. One of the famous open-addressing collision resolution algorithms in hash tables is Cuckoo Hashing [61]. A BCHT, as defined earlier, is actually the horizontally-vectorized version of cuckoo hashing. Many of the existing approaches on SIMD-aware batch hash tables [29, 68, 71] are in fact the improved version of BCHTs. Besides presenting a BCHT approach, Polychroniou et al. [64] also proposed and compared other horizontally-vectorized hash tables based on double hashing and linear probing hashing schemes.

Vertical vectorization [64] is a more generalizable but more complex approach to

benefit from SIMD in batch hash tables. It is more generalizable because it does not change the inner structure of a hash table. However, it is more complex as it needs the collision-resolution algorithm to be translated into SIMD operations. Contrary to horizontal vectorization, in this approach, the input of a lookup operation must be a vector of probing keys. In each vectorized lookup, the vertical approach will pass a vector (of register size) of inputs through the lookup process and by using mask registers and advanced SIMD features (like SIMD permutations) probe those keys at the same time. When the status of a key lookup is determined (found/not-found), its related CPU-register lane will be assigned to the next key in the batch of keys that are waiting to be processed. By conducting different experiments, Polychroniou et al. [64] and Shankar et al. [83] have shown that vertical vectorization yields higher performance than the horizontal approach. In vertical vectorization, since the hashing scheme must be translated into SIMD code, we need to use gathers and scatters to read/write from/to different entries of a hash table. As mentioned earlier, the scatter instruction is only available in limited types of processors hence the vertically-vectorized insertions can only be implemented on specific hardware.

The third category for SIMD-aware batch hash table design is a hybrid approach. However, the experiments in [83] show that the results of mixing the vectorized and horizontal vectorization approaches will not further improve the performance.

Besides the SIMD-aware vectorization methods discussed above, SIMD operations can also be used in the development of hash functions in any hash table [3]. Although this approach can improve the performance of hashing, it is orthogonal to the scope of this chapter.

Prefetching-Enabled Hash Tables. Modern CPUs support hardware and software prefetching. Prefetching improves the performance of a program by amortizing the costs of memory access over time (in parallel to running computations).

Hardware prefetching is automatically enabled by the compiler and executed by the CPU when long and contiguous access to memory (e.g. iteration on a large vector) is requested by the program. The developer does not have much control over the hardware prefetching. On the other hand, in software prefetching, the developers can issue on-demand asynchronous prefetching commands to prefetch their future memory accesses.

In hash tables, regardless of the hashing scheme, accessing entries is based on the value of the computed hash for each provided key. This is an example of random access to a non-contiguous memory that can be improved by using the software prefetching.

To have an effective prefetching in hash tables we need (1) a batch of operations and (2) a large hash table. The batch of operations provides enough computational tasks for the CPU while the prefetching instructions for future memory accesses are being processed. Also, if the hash table is not large enough, its content can entirely fit into the CPU cache. This cancels the benefits of prefetching and only puts its overheads on the CPU.

Parallel (Concurrent) Hash Tables. There is a long tail of research and open-source projects on parallel (i.e., multi-threaded) hash tables. The state-of-the-art systems [10, 8, 52, 54] have tried to enable concurrent insertion, lookup, and deletion on hash tables. These approaches can generally be divided into two categories: (1) the approaches that resolve the contentions using lock-based mechanisms, and (2) the lock-free hash tables that use atomic instructions, such as Compare-and-Swap (CAS) as their synchronization mechanism. Although these parallel hash tables offer better performance in comparison with the sequential hash tables, they are not fully exploiting the advanced features of modern hardware such as SIMD awareness and prefetching. This is due to the lack of a batch API.

Although most of the batch hash tables offer batch insertions or deletions, previous research has shown that in most of the high-performance use cases (e.g. join processing in relational algebra, vector/tensor processing in linear algebra, and packet processing in computer networks), the amortized cost of insertions is negligible in comparison with the overall cost of highly frequent (or even endless and continuous) lookups [29]. Thus, in this chapter, we only consider optimizing the lookup performance.

5.3 Architecture

In this section, we discuss the structure of Vec-HT and its high-level API.

5.3.1 Hash-Table Structure

The hash table consists of an array of `bucket` objects each of which contains a key (32 bits) and its related value (32 bits). As the hashing scheme, we use open addressing with linear probing (similar to [64]). We also use multiplicative hashing as our hash function.

Generally and without considering any optimization, to look up a key in the hash table, we first compute the hash of the key and then find its corresponding bucket in

the array. If the key of that bucket is empty (the value of the empty key is defined during hash table initialization) we return the empty key which means "not found". Otherwise, we check if the key in the bucket equals the probing key or not. If it is, we return the value, otherwise, we continue checking the next buckets to find an equal key or an empty bucket.[2]

5.3.2 High-Level API

To make a batch hash table more accessible to developers, we expose an easy-to-understand API. The Vec-HT namespace consists of three classes. A batch hash table class (`lp_map`) that is currently designed by having the open-addressing linear-probing hashing scheme (Figure 5.2), a batch-iterator class (`iter_batch`) that defines the data type containing the result of batch lookup, which also supports parallel iterations (Figure 5.3), and the `bucket` class (cf. Secion 5.3.1) that is related to each entry of the hash table.

Batch Hash Table Class. The constructor of `lp_map` takes three arguments. The first one (`size`) is for setting the maximum number of elements that will be inserted into the hash table. The second parameter sets the group size for the internal prefetching. And the third parameter (`threads`), determines the number of threads (cores when the Hyper-Threading is disabled) to enable concurrent batch processing.

The methods exposed by the API of `lp_map` are categorized into two sets: non-batch and batch methods. The non-batch methods include `insert`, `find` that are similar to the standard hash table interfaces such as `std::unordered_map`. These methods give the developers the freedom of using Vec-HT without batch processing.

There are five batch-based methods. The method `insert_batch` inserts an array of keys and their related values into the hash table. The remaining methods are related to vectorized lookup which is the main focus of this work (cf. Section 5.2). The method `find_batch` accepts three arguments: (1) an array of keys to look up, (2) the size of that array, (3) a boolean flag called `complement` that is used to request for the not-found elements instead of the successfully-found ones, and (4) an object of a Vec-HT-specific class called `iter_batch`. The `iter_batch` class is responsible for keeping the results of a vectorized lookup and making the (parallel) iterations over them possible. The other method in `lp_map` is `zip`. Although it is not a usual API for ordinary hash tables, we found it very useful in the case of batch hash tables. This method, similar

[2]Currently, due to the restrictions imposed by SIMD-vectorization, Vec-HT only supports 32-bit integer keys (similar to [2, 64, 71]).

```
namespace vec_ht
{
    using K = uint32_t;
    using V = uint32_t;
    using P = uint32_t;
    // ---- Linear-Probing Batch Hash Table Class -----------
    class lp_map
    {
        // ...
    public:
        lp_map (size_t size, size_t group_size=64, size_t threads=1);
        // ---- Non-Batch APIs -------------------------------
        inline bool insert (const K& key, const V& value);
        inline bucket* find (const K& key);
        // ---- Batch APIs -----------------------------------
        inline size_t insert_batch (uint32_t* keys, uint32_t* values,
        size_t size);

        inline size_t find_batch (uint32_t* keys, size_t size,
        bool complement, iter_batch* res_it);

        inline size_t find_batch_apply (uint32_t* keys, size_t size,
        bool complement,
        std::function<void(K& key, V& value)>const& f);

        inline size_t zip (uint32_t* keys, uint32_t* payloads,
        size_t size, bool complement, iter_batch* res_it);

        inline size_t zip_apply (uint32_t* keys, uint32_t* payloads,
        size_t size, bool complement,
        std::function<void(K& key, V& value, P& payload)>const& f);
    };
}
```

Figure 5.2: High-level API of Vec-HT in C++.

to the `find_batch`, does a lookup for the provided array of keys. However, it takes one additional argument; `payloads` assigns one value to each key in the keys array. When the method `zip` is called, the result also contains the related payloads of the found keys. The `iter_batch` class can also keep the results of a `zip` API. We show how `zip` can be used in practice in Section 5.5.

The remaining useful APIs are `find_batch_apply` and `zip_apply`. These APIs do the same job as their related discussed APIs. However, a user can pass their customized lambda function to be applied on the tuples of key-values (or key-value-payloads) whenever a match is found. Thus, there is no need to pass an `iter_batch` object to these APIs since it is the user's responsibility to handle the output. These two APIs

```
namespace vec_ht
{
    // Container and Iterator Class for find_batch/zip Results
    class iter_batch
    {
    // ...
    public:
        iter_batch (size_t max_size, size_t threads,
        bool for_zip=false)

        K** get_keys();
        V** get_values();
        P** get_payloads();

        inline void foreach
        (std::function<void(K& key, V& value)> f);

        inline void foreach
        (std::function<void(K& key, V& value, P& payload)> f);

        inline void foreach_parallel
        (std::function<void(K& key, V& value)> f);

        inline void foreach_parallel
        (std::function<void(K& key, V& value, P& payload)> f);
    };
}
```

Figure 5.3: High-level API of batched iterator in C++.

can improve the performance of pipelined analytical tasks because they eliminate the need for the materialization of intermediate results (`iter_batch`). In other words, using these APIs, the user can fuse the batch lookups with the following operations in the pipeline. This is especially useful in the context of pipelined analytical query processing [76, 80, 60].

Batch Iterator Class. The `iter_batch` class can be constructed by passing (1) an upper bound on the size of the results, (2) the number of threads (it must be the same as the one in `lp_map`), and (3) a boolean that shows if we want to pass this object to a `zip` or a `find_batch` API. By receiving these parameters, the memory needed for the storage of parallel-processed results will be allocated. Then, the class is ready to be sent to the methods `find_batch` or `zip` in a *destination-passing style* [77, 87]. The destination-passing style improves the performance of computational workloads by bringing the memory-allocation overheads out of the performance-critical part of the workload. The `iter_batch` class has also two overloads of `foreach`. After the execution

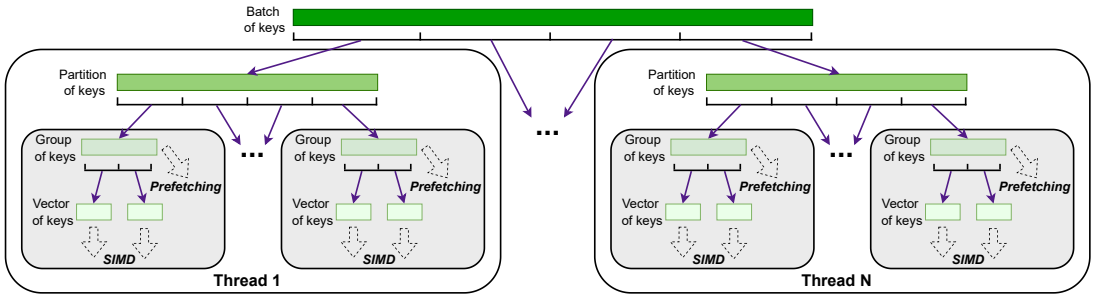


Figure 5.4: The architecture of batch lookup in Vec-HT.

of `find_batch` or `zip`, their relevant `foreach` method can be used for a sequential iteration over the results. The `foreach` method takes a lambda function that will be applied to each of the stored results in the `iter_batch`. Similar to the `foreach` methods, `iter_batch` also offers `foreach_parallel` methods that use multiple threads to apply the provided lambda function on the stored results. In the `foreach_parallel` methods, since they are internally implemented based on `tbb::parallel_for_each` [8], the developer can also use parallel containers such as `tbb::enumerable_thread_specific` or any other off-the-shelf parallel container to handle storing/aggregation of lambda outputs.

5.4 Design

In this section, we discuss the design decisions behind the optimizations in our approach and show how they relate to each other. Figure 5.4 shows the architecture of a batch lookup in Vec-HT. As it is shown in this Figure, the batch input is partitioned into smaller chunks on different levels and for different optimization purposes. In this section, these levels of input partitioning and the rationale behind them will be covered.

5.4.1 Parallel Processing

To make the most out of the multi-core processor, in case of a batch lookup, we partition the input batch of keys and assign each partition to a thread for parallel batch processing. Then, when the `find_batch` or `zip` methods are called, they call the `parallel_dispatcher` method internally. This lower-level method is responsible for managing the threads needed for the computation and passing them the contextual information. The interface of `parallel_dispatcher` is shown in Figure 5.5.

The `parallel_dispatcher` method takes six arguments:

```
template<typename FUNC_TYPE>
inline size_t parallel_dispatcher(uint32_t* keys,
                                  uint32_t* payloads,
                                  size_t size,
                                  bool complement,
                                  FUNC_TYPE func,
                                  iter_batch* res_iter)
```

Figure 5.5: The signature of `parallel_dispatcher`, a function used internally for parallelization.

1. The keys that we want to look up in the hash table.
2. The associated payloads (set to `NULL` if the caller method is `find_batch`).
3. The size of the `keys`.
4. The `complement` flag (cf. Section 5.3.2).
5. The lambda function passed when the user calls `find_bath_apply` or `zip_apply`.
6. The `iter_batch` which is passed in case of calling `find_batch` or `zip`.

5.4.2 SIMD-Awareness

In this section, we explain the vertical-vectorization approach for batch lookups at a high level and refer the interested reader to Polychroniou et al. [64] for more details.

Suppose that a number of `W` keys can be stored in a CPU register. When the batch lookup starts, `W` keys of the input vector will be fetched into the `keys` register. To load the input keys, we use the selective load SIMD operation. This operation uses a mask to select which lanes of the target register must be filled with the new values and which of them must be set to zero. In the beginning, we define a register (`invalid`) with all lanes activated and pass it to the selective load as the mask. This means that we plan to read `W` new keys from the input. Then using the SIMD operators, the hash value of all the keys in `keys` is computed simultaneously and stored in the `hash` register. In VecHT, as we use a simple multiplicative hash function, the computation of hash values consists of logical and arithmetic SIMD operations such as vectorized multiplication and shift.

By having the hash values, we use the SIMD `gather` operation to retrieve the needed hash table entries. The `gather` operation reads multiple memory addresses (stored in a

register) at the same time. Since the value of each computed hash shows the possible offset of each key in the hash table entries, we apply `gather` on the address of the hash-table entries array and the `hash` register. As a result of executing two `gather` operations, two registers for the retrieved keys (`tab_keys`) and values (`tab_vals`) will be created.

Next, based on the linear probing algorithm, we check the equality of key in `keys` and `tab_keys`. We do this check using SIMD logical operators. This check can have three different results for each lane: (1) the key is empty which means that the key is not found in the table (2) the keys are equal which means the key is found (3) the key is not empty or is not equal to the given input key and thus it needs further probing in the next rounds. For the not-found keys, we activate their relevant lanes in `invalid` register. For the found keys, we define and activate the relevant lanes in a new register called `output`. Finally, for the ones that need further probing, we create a new register called `offset`, initialize it with `0`, and increment its relevant lanes by `1`.

In this phase of the algorithm, we first add `out` to `invalid` and store the result in `invalid`. We do this since we are finished with both found and not-found keys and we want to fetch the new keys instead of them in the next round of lookups. Then, by using a static permutation table, we extract the permutation masks needed to align the active lanes of `invalid` and `output` to one side of the register. These permutation masks will be used in the SIMD `permute` operation, which changes the order of lanes in the register using a provided mask. First, we use the permutation mask of `out` on `out` itself and on `keys`. Now, we are ready to save the found keys to the target memory (reserved memory in `iter_batch`) using a selective store SIMD operation that its mask is the permutated `out`. The total number of output keys will also be updated at this stage. It is notable that the original vertical vectorization [64] uses a buffer to store the results temporarily and spills them to the output whenever the buffer is full.

After the work on the found keys is finished, we count the active lanes in `invalid` to know how many new keys are needed to be fetched. Then, we apply the `invalid` permutation mask on `keys`, `hash`, and `offset` to make them ready for the next run. By starting the next round of lookups, again the new keys will be fetched based on the updated `invalid` register. It is important to mention that this time all of the hashes are re-computed and the ones with inactive lanes in `invalid` will also be incremented by `offset` to point to the next entry in the hash table.

To return the value of found keys in the table and also their related payloads we need further considerations. For the hash table values, which are stored in `tab_val`, we can permute them using the `out` permutation mask and store them in their related mem-

ory in `batch_iter`. Similarly, the payloads can be selectively loaded exactly similar to the new key. Then they will be passed through the algorithm by similar permutations, and finally will be stored in the relevant output memory.

If the size of input keys is less than `W` or in the case of processing the last `W` keys, the algorithm switches to a normal scalar (non-SIMD) lookup in the hash table and stores the result into the `batch_iter`. This is because there are not enough keys to do a safe and efficient SIMD lookup.

5.4.3 Prefetching and Its Adaption Challenges

Vec-HT is a prefetching-enabled vectorized approach; we apply the prefetching on top of the parallel vertical vectorization. There is a large design space for combining prefetching and SIMD vectorization. We examined this design space through micro-benchmarking (cf. Section 5.8). The important design parameters for this combination are as follows:

- **Standard vs Group Prefetching:** putting the prefetching commands at the beginning of the main loop of the vertical vectorization is the standard solution for adding prefetching. We compare it with another approach (Group Prefetching) proposed by Chen et al. [32] in the context of databases.

- **Group Size for Group Prefetching:** Considering the group prefetching approach, the selection of the different group sizes might affect the performance of the system.

- **Optimistic vs Pessimistic Linear Probing:** Given that we use linear probing, there are two choices for prefetching for each key. Optimistic: we consider that the probe hits the correct location on the first try (or finds the location to be empty) and does not need to probe further; thus we only prefetch the hashed location. Pessimistic: we consider the case of not having a hit and thus prefetching the next location(s) as well.

- **Memoization of Computed Hashes:** We need the hash values in two places: (1) prefetching stage, and (2) vertical vectorization stage. We have the option of memoizing the hash value in the first stage and reusing/recomputing it in the second stage.

```
foreach group in array by GROUP_SIZE {
  // prefetching stage
  foreach vector in group {
    vector_h <- simd_hash(vector)
    if(HASH_MEMOIZE)
      mem_h += vector_h
    foreach h in vector_h {
      prefetch(buckets[h])
      if(!OPTIMISTIC)
        prefetch(buckets[h+1])
    }
  }
  // vertical vectorization stage using linear probing
  while(vec_elems not probed in group) {
    if(HASH_MEMOIZE)
      vector_h <- mem_h[vec_elems]
    else
      vector_h <- simd_hash(vec_elems)
    res <- vertical_vectorization(vec_elems, vector_h)
    if(OUT_BUFFER) {
      buffer += res
      if(buffer is full)
        flush(buffer)
    }
  }
  if(OUT_BUFFER) {
    flush(buffer)
  }
}
```

Figure 5.6: A generic algorithm showing the design space of combining prefetching with vertical vectorization.

- **Buffering:** In vertical vectorization, we can write the output into an output buffer and if it is not carefully adapted to the prefetching design, it might result in performance overheads.

Figure 5.6 depicts a generic and high-level algorithm for combining prefetching with vertical vectorization (based on the assumption that we take the group-prefetching approach instead of standard prefetching, which is a take-away message of micro-benchmarks in Section 5.8). In this algorithm, by setting the `GROUP_SIZE` parameter, we can enable the grouping loop that partitions the input keys into parts of size `GROUP_SIZE` and then run the algorithm on these smaller batches. By having a group of keys as input, before starting the vertical vectorization, we define a loop over the group keys (prefetching loop). In each iteration of the prefetching loop, we first compute the hash

```
// build phase
auto ht = vec_ht::lp_map(2*S_size, 64, 4);
ht.insert_batch(S_A, S_B, S_size);
// probe phase
auto res_it = vec_ht::iter_batch(R_Size/3, 4, true);
ht.zip(R_A, R_F, R_size, false, res_it);
// printing the output
res_it.foreach_parallel(
[](auto& key, auto& value, auto& payload){
    std::cout << "S_A/R_A: " << key << " | ";
    std::cout << "S_B: " << value << " | ";
    std::cout << "R_F: " << payload << std::endl;
});
```

Figure 5.7: Implementation of a hash join operator (on S and R relations) using Vec-HT.

of `W` elements using the SIMD approach mentioned before. By setting the `HASH_MEMOIZE` parameter to `true`, we can store the hash values and reuse them inside the vertical vectorization algorithm. After making a decision on memoization, we raise `W` software prefetch commands for the address of target entries in the hash table. Here we can do the prefetching also for the next bucket by setting the `OPTIMISTIC` parameter to `false`. After finishing the prefetching loop, all the related entries are prefetched. At this stage, the vertical vectorization algorithm will be executed for the current group and if the `OUT_BUFFER` parameter is enabled, the output buffering happens.

5.5 Use Cases

In this section, we show the usability of our proposed batch table, by showing several high-performance data analytics use cases.

5.5.1 Relational Hash Join

First, the code for a join on two relations (S and R) is shown in Figure 5.7. We assume that the relations are stored in columnar format (i.e., struct of arrays) which is a popular design decision in high-performance query engines [60]. The code is executed using a prefetching group size of 64 on 4 threads. We keep these settings for all of the use cases covered in Section 5.5.

Build Phase. In the beginning, the batch hash table is initialized with the table size, group size, and the number of threads. The size is set to twice the number of the elements in the relation on the build side of the hash join (`S`). We do so to keep the fill

```
auto ht = vec_ht::lp_map(2*S2_size, 64, 4);
for (int i=0; i<S2_size; i++) ht.insert(S2[i], 1);
auto res_it = vec_ht::iter_batch(S1_Size/5, 4, false);
ht.find_batch(S1, S1_size, true, res_it);
res_it.foreach_parallel([](auto& key, auto& value){
    std::cout << "Item: " << key << std::endl;
});
```

Figure 5.8: Implementation of a Set-Difference operation (S1\S2) using Vec-HT.

ratio of the hash table less than or equal to 50%. Then, using the `insert_batch` method, all the key/value pairs from `S` are inserted into the hash table in a batch style.

Probe Phase. Before running the batch lookups on the hash table, we first prepare the `iter_batch` for the results. This object is initialized by setting three parameters. The first one is an upper bound for the join result size. The more precise this estimation is, the less memory allocation time is spent during the batch lookups. The second parameter is the number of threads that must be equal to the one already passed to the hash table. The last parameter is a flag that shows if the `iter_batch` object will be used in a `zip` or `find_batch` API. Next, by having the `res_it` object, we can run our `zip` method to join the relations based on the `R_A` and `S_A` columns and by considering the `R_F` column as the payload.

After the `zip` execution is finished, we use the `foreach_parallel` method of `res_it` to iterate over the join results and print them. The desired functionality (printing) is passed to the `foreach_parallel` using a user-defined lambda function.

5.5.2 Set Operations

As our second use case, we show the implementation of a set difference operation (S1\S2) using our approach in Figure 5.8. The sets S1 and S2 are stored in two arrays. The code is almost the same as the one in the relational-join example. Its main difference is in using `find_batch` instead of `zip`. It is because there is no payload to be passed into the `zip` API. Furthermore, in this example, we see the usage of `complement` parameter as we need the elements of S1 that are not found in S2. To implement the set intersection we need to use the `zip` method, which is similar to the vector inner product, that is presented next.

```
auto ht = vec_ht::lp_map(2*V1_size, 64, 4);
ht.insert_batch(V1_idx, V1_val, V1_size);
auto res_it = vec_ht::iter_batch(V1_size, 4, true);
ht.zip(V2_idx, V2_val, V2_size, false, res_it);
uint32_t sum = 0;
res_it.foreach([](auto& key, auto& value, auto& payload){
    sum += value * payload;
});
```

Figure 5.9: Implementation of an Inner-Product operation (V1·V2) using Vec-HT.

5.5.3 Sparse Vector Operations

The last use case that we cover in this section is the inner product of two vectors (V1·V2). The related code is shown in Figure 5.9. In this implementation, we again use the `zip` method, since there are payloads on the `V2` side (values of `V2` for each index). After the `zip` execution is finished, this time we do a non-parallel iteration (`foreach` API) over `res_it` to prevent contentions on the `sum` shared memory. As mentioned in Section 5.3.2, a developer can easily use the concurrent containers (e.g. `tbb::enumerable_thread_specific<uint32_t>`) instead of `sum` here. However, in this case, we are interested in exhibiting the usage of non-parallel `foreach`.

5.6 Implementation

In this section, we give a more detailed explanation of the implementation behind Vec-HT.

Attributes of `lp_map`. The attributes of `lp_map` class are shown in Figure 5.10. All these attributes are initialized in the class constructor. The `size`, `threads_`, and `group_size_` attributes are set to the values that are passed by the constructor. The `hash_factor_` is set to a randomly generated number. The `empty_key_` attribute is set to the maximum possible value for `uint32_t` type. Lastly, we use an array of `bucket` structs (`entries_`) as the entries of our hash table. It has been shown that using an array of structs instead of a struct of arrays does not affect the performance of hash tables [67]. The memory of this array is allocated after the calculation of `size_` attribute.

Attributes of `iter_batch`. The attributes of the `iter_batch` class are shown in Figure 5.11. Here `max_size_` is passed by the constructor and shows an upper bound on the number of results for a `find_batch` or `zip` method call. The `threads_` and `for_zip` attributes are given by the constructor. The next three attributes are the storage for re-

```
namespace vec_ht
{
    class lp_map
    {
    private:
        size_t size_;
        size_t threads_;
        uint32_t group_size_;
        uint32_t hash_factor_;
        uint32_t empty_key_;
        bucket* entries_;

        template<typename FUNC_TYPE>
        inline size_t parallel_dispatcher(uint32_t* keys,
           uint32_t* payloads, size_t size, bool complement,
           FUNC_TYPE func, iter_batch* res_iter)

        template<typename FUNC_TYPE>
        inline size_t find_batch_inner(uint32_t* keys,
           uint32_t* payloads, size_t size, bool complement,
           FUNC_TYPE func, iter_batch* res_iter, size_t thread_id)
    public:
        // ...
    };
}
```

Figure 5.10: The internal of the `lp_map` class.

sults of a `find_batch` or `zip` method call. In the constructor, we allocate arrays of arrays to these pointers; this will allocate memory of size `max_size` for each thread. In the case of a `find_batch` method call, we do not allocate and use the `payloads_` pointer. As the last attribute, we have `threads_res_size_` that will be extended to the size of `threads_`. Each element of this vector is used by a thread to store the size of the results for that thread. By using this attribute, the iterations over the results will be more efficient (cf. Figure 5.12).

Implementation of `foreach_parallel`. In Figure 5.12, the implementation details of `foreach_parallel` are presented. In this method, we create a range of integers from `0` to `thread_-1` and assign each number in the range to a thread. Then, by execution of a `tbb::parallel_for_each` and passing the prepared range to it, we run a lambda function on each thread with `thread_id` as its single argument. In the lambda function, using the `thread_id` argument, the `max_size_` attribute of `batch_iter` class, and the vector of result sizes for each thread (`threads_res_size_`), we compute the boundaries of the result vectors that are assigned to the current thread. By having those boundaries, we

```
namespace vec_ht
{
    class iter_batch
    {
    private:
        size_t max_size_;
        size_t threads_;
        bool for_zip_;

        uint32_t** keys_;
        uint32_t** values_;
        uint32_t** payloads_;

        std::vector<size_t> threads_res_size_;
    public:
        // ...
    };
}
```

Figure 5.11: The internal of the `iter_batch` class.

can finally apply the developer-provided lambda (`func`) on each triple of `key`, `value`, and `payload` in the results assigned to this thread.

Implementation of `zip`. The implementation of the `zip` method is presented in Figure 5.13. To bypass the overheads of dispatching in the parallel scenario, this method (and other performance-critical methods such as `find_batch`), checks the `threads_` attribute of the current `vec_ht`. If it detects a sequential setting, then calls the internal method (`find_batch_inner`) which is responsible for the vertical vectorization and prefetching. Otherwise, the method calls the `parallel_dispatcher` (cf. Section 5.4) to partition and dispatch the work among the pre-determined number of threads. To call either of these two internal methods, the `zip` method provides them with the appropriate arguments or `null` types where required.

Implementation of `find_batch_inner`. Figure 5.14 shows a simplified implementation for `find_batch_inner`. This method is the most complex method in Vec-HT. It operates over a subset of batch input (the partition that is assigned to each thread) and is responsible to do the following tasks:

- To partition the input into group-sized batches.
- To compute and memoize the hashes for each group.
- To do the group prefetching for each group.

```
inline void foreach_parallel
(std::function<void(K& key, V& value, P& payload)> f)
{
    auto range = std::vector<size_t>(threads_);
    for (size_t i=0; i<threads_; i++) range[i] = i;
    tbb::parallel_for_each(range, [&](size_t thread_id)
    {
        for (size_t j=0; j<threads_res_size_[thread_id]; j++)
            func(keys_[thread_id][j],
                 values_[thread_id][j],
                 payloads_[thread_id][j]);
    });
}
```

Figure 5.12: The implementation of `foreach_parallel` in `iter_batch`.

```
inline size_t zip (uint32_t* keys, uint32_t* payloads, size_t size,
bool complement, iter_batch* res_it)
{
    if (threads_ == 1)
        return find_batch_inner<no_func_type>(keys, payloads, size,
        complement, nullptr, res_it);
    else
        return parallel_dispatcher<no_func_type>(keys, payloads,
        size, complement, nullptr, res_it);
}
```

Figure 5.13: Implementation of `zip` in `lp_map`.

- To run the entire vertical vectorization algorithm for each group.
- To buffer the found keys and their related values and payloads.
- To store the results into the `iter_batch` or apply `func` over them.

We present a brief overview of the above-mentioned steps in Figure 5.14. In the depicted pseudo-code, we highlighted our key differences to the approach of Polychroniou [64] et al. The main difference here is the introduction of the memoization and prefetching loop. In this loop, we first compute the hashes of a given key group vector-by-vector, using SIMD operations. Then, we store (memoize) the computed hashes and try to prefetch them for faster access in the near future. The interleaving of the hash computation, memoization, and prefetching is the main contributer to the efficiency of our approach. Note that in Figure 5.14, the `vector_size` is a global constant (8) which is a function of the selected data-type size (32 bits) and the SIMD vector size (256 bits), computed as vector size divided by data-type size.

As the last topic in this section, to implement the complement behaviour in Vec-HT, we have slightly changed the original vertical-vectorization algorithm. In the case of a complement, we replace the keys with an *invalid* status with the keys with an *output* status. In other words, the found keys are considered *invalid* and the not-found keys are the *valid* ones that must be stored in the output.

5.7 Integration to SDQL.py

In Section 5.5, by integration of Vec-HT into SDQL.py (cf. Chapter 4), we show the usefulness of Vec-HT for analytical query processing. Our focus is mainly on vectorizing the probe side of hash joins. By investigating this challenge, the SDQL.py specialized engines can benefit from all the advantages of Vec-HT in their join processing computations.

We start the discussion by showing the primary join of the TPC-H Q12 in Figure 5.15. In this example, the first summation iterates over the `orders` table and creates an index, a dictionary with `o_orderkey` as its keys and `o_orderpriority` as its values, for the build side of a join. Then, the second summation iterates over `lineitem` table, evaluates a complex condition, and creates the probe side of the join. It is important to mention that the output of this iteration, does the probing and group-by aggregation simultaneously (See loop fusion optimization in Section 4.6).

Now, we show the vectorized version of the same query (Figure 5.16) when the probe computation is done in a batch fashion. To achieve this, in the first summation, we pass a flag determining the type of output dictionary for that summation. Later on, our C++ code generator will declare and populate a `vec_ht::lp_map` instead of the other hash tables (cf. Section 4.4) for the results of this specific summation. This will enable us to use the advantages of Vec-HT while probing in the next step. Since the current version of Vec-HT only supports integer types as its keys/values, instead of materializing the exact values (`o_orderpriority` in this case), we store index of the related tuple to make the future access to all of its columnar values possible (early vs late materialization [20]).

Focusing on the probe side and considering the staging idea presented in [58], we introduce `ifbuf`, a new construct for SDQL.py that makes a foundation for the batch operations in the SDQL.py. This construct applies a condition (second argument) to an item iterator (first argument) and makes buffers of a specific size (default size is `1024`) if the item satisfies the condition. Whenever a buffer is ready to be consumed, it passes

the buffer to its body (third argument) and evaluates it. As it is shown in Figure 5.16, we use this new construct in the second summation of Q12 and prepare buffers for doing batch lookups on the build side. Now, we pass the populated buffers to a new API (`vfind`) that is only available when the caller (`ord_indexed` in this example) can do batch operations. This API extracts the lookup keys by applying a lambda function (second argument) on the input batch (first argument) and returns tuples that contain the matched keys, all values from the build and probe sides of the join. Back to our example, the result of `ord_indexed.vfind(buf, lambda q: q[0].l_orderkey)` contains values from `orders` and `lineitem` where `o_orderkey == l_orderkey`. As the next step, similar to our non-vectorized example (Figure 5.15), iterates over the joined results and prepares the output of the fused group-by aggreagtion.

We will cover the evaluation of this integration effort in Section 5.8.

5.8 Evaluation

In this section, we first present our experimental setup for the evaluation. Then, we show the performance of our approach in different use case scenarios and compare its performance with various competitors.

5.8.1 Experimental Setup

All experiments are done on a single machine equipped with 16GB of DDR4 RAM, and an Intel Core i5-10210U 1.6GHz with 4 cores and 256KB, 1MB, and 6MB of L1, L2, and L3 cache respectively. Hyper-threading is disabled for the experiments. We have used Ubuntu 20.04.3 as the OS. Our C++ code is compiled with G++ 9.4.0 using the `-O3` flag. To enable SIMD operations, we use the `-march=core-avx2` flag. All of the experiments were executed with 5 warmup rounds followed by 5 timed iterations. Then, we took the average of the timed iterations.

Workloads. To run the experiments, we use three different workloads. For the micro-benchmarks and the join experiments, we use the random data generator from [64]. By focusing on the notion of inner joins in databases, it generates two random data sets as inner and outer relations. The elements of the inner data set are inserted into the hash table creating the build side of the join. The elements of the outer data set shape the probe side of the join. The data generator accepts arguments for `inner_size`, `outer_size`, and `selectivity` of the join.

The second workload is used for the set and sparse vector experiments. The set/vector generator receives `size`, `density`, and `maximum_value` as input parameters. For each set of size `N`, it generates `N×density` unique random numbers from the range of `[0, N)` as the value of items in the set. Similar to the sets, for the vectors, it generates unique random numbers but uses them as vector indexes. Then, using the `maximum_value` parameter, it generates random integer numbers in the range of `(0, maximum_value]` as vector values.

As the last workload, we use the well-known TPC-H [19] benchmark with a scaling factor (SF) of 1 (1 GB of data) for the evaluation of our approach in analytical queries. It is important to note that in all of the benchmarks, we keep the fill ratio of all alternative hash tables less than or equal to 50%.

Alternatives and Competitors. In the micro-benchmarks, to evaluate our proposed approach, we compare it with (1) a scalar implementation of Vec-HT without any optimization (2) a scalar + prefetching version (3) and the vertical-vectorization approach by Polychroniou et al. [64]. For all alternatives, we consider sequential and parallel versions. As mentioned in Section 5.6, we reuse the code from [64] as the base for our implementations. We do not add the comparison with the approaches such as DPDK [2], as it is previously shown [64] that the BCHT approach is slower than vertical vectorization which is the basis for Vec-HT.

We also compare our implementations with state-of-the-art hash tables. TBB [8] is a well-known parallel computation framework. We use its offered parallel hash table `tbb::concurrent_unordered_map` as one of our competitors. Libcuckoo [52] is a research project on fast parallel hash tables and we use its open-source implementation `libcuckoo::cuckoohash_map`. As the last competitors, from the open-source high-performance hash-table project phmap [10], we use its sequential and parallel data structures `phmap::flat_hash_map` and `phmap::parallel_flat_hash_map`. The implementation of Vec-HT that we use in the benchmarks has a group size of 64, taking an optimistic approach, with enabled memoization and simple buffering inside each group.

5.8.2 Benchmarks

In this section, we first consider the design space of combining prefetching with vectorization and show the best implementation. In addition, we evaluate the effectiveness of our implementation in comparison with scalar, scalar + prefetching, and pure vertical vectorization in a holistic micro-benchmark for hash join processing. Then, we

show its superiority over the existing hash table packages in different use cases. We consider benchmarks on set and vector kernels that are largely used in big data analytics frameworks such as query processors (e.g., BigTable, SparkSQL) and linear algebra frameworks (e.g., MLLib, SystemDS, distributed TensorFlow). We finally cover benchmarks on database query processing over a selected subset of TPC-H queries, the main benchmark for analytical queries.

Standard vs Group Prefetching Micro-Benchmark. The first micro-benchmark related to the design space (cf. Section 5.4.3) is shown in Figure 5.17. In these experiments, our focus is to show the performance difference between the standard prefetching and group prefetching approaches. The results show that even though the standard way of prefetching offers performance improvements compared to non-prefetched approaches, it cannot beat the performance and scalability of the group prefetching.

Optimistic vs Pessimistic Prefetching Micro-Benchmark. As mentioned in Section 5.4.3, by having a linear probing scheme in the hash table, for a given key, we can prefetch more than one bucket to improve the chance of a cache hit after an unsuccessful lookup. Although it seems to be an interesting strategy to take, the limited prefetching capability of CPUs, the variety in workload characteristics, and the parameters such as the fill ratio of the hash table can affect the benefits of this strategy. Figure 5.18 shows the performance results for prefetching with optimistic and pessimistic approaches. Both optimistic and pessimistic approaches perform better than the alternatives. However, for the smaller hash tables, the performance of the pessimistic approach is worse than the optimistic one and sometimes even worse than pure vertical vectorization. Thus, we decided to take the optimistic approach for our final implementation of Vec-HT.

Group-Size Micro-Benchmark. Figure 5.19 depicts group-prefetching performance with different group sizes while altering between the optimistic and pessimistic strategies, as well. Overall, we see a performance improvement by increasing the group size, however, this improvement no longer holds after the group size of 64. The results show that selecting a group size of 64 with the optimistic strategy is a reasonable choice.

Hash Memoization Micro-Benchmark. Our last micro-benchmark investigates the effects of memoizing the hash values during the prefetching stage. Figure 5.20 presents the results of these experiments. In selectivities of 0.1 and 0.5, it is clear that the memorized version performs better than the non-memoized one. With a selectivity of 1, the non-memoized version does better (with a narrow improvement compared to memorized version) for larger hash tables, however, the memoized version is faster for

smaller hash tables. Thus, we select the memoized version for Vec-HT.

Relational Inner Join for State-of-the-Art Hash Tables. In these experiments, using the first workload described in Section 5.8.1, we run a simplified relational inner-join operation using different hash table implementations to measure the performance of each approach.

As it is shown in Figure 5.21, our approach `vec_ht::lp_map` outperforms other approaches irrespective of the build size, join selectivity or concurrency level. It is on average 5× faster on both 1- and 4-cores.

Set Operations Use Case. To show the performance of our approach in set operations, we run experiments on the set intersection and set difference. To run the operations, we iterate over the first set (S1) and look up the values in the other one (S2). Since the hash tables in our experiments (except `tbb::concurrent_unordered_map`) do not support parallel iterations, to have a more comprehensive benchmark, we keep S1 in the vector format and only make a hash table for S2. For the set-difference operation, we use an implementation similar to Section 5.5.2. Figure 5.22 depicts the results of our set experiments. In these experiments, using a fixed size for S1 and S2 and a fixed density for S1, we observe the changes in the run time of each competitor while increasing the density of S2.

In the set-difference experiments, our approach outperforms the others with an average of 11× speedup on 1 and 4 cores. Similarly, in set intersections, our approach performs better than the others excluding `phmap::flat_hash_map`. By excluding `phmap`, the proposed approach is on average 6× and 5× faster than the others on 1 and 4 cores, respectively. For the small S2 sizes, when the log of S2 density is less than or equal to -4, the hash table can still be fitted into the L3 cache, thus the benefit of using our prefetched approach is not promising and the overall performance is near to what `phmap::flat_hash_map` offers. However, after passing that size limit, `phmap::flat_hash_map` run time goes higher while our approach keeps its good performance thanks to software prefetching.

Vector Operations Use Case. In Figure 5.23, the results of running experiments on vector inner-product (cf. Section 5.5.3) and pair-wise multiplication are shown. Similar to the set operations, here we keep V1 in the vector format and embed V2 into a hash table. The experiment parameters are also set to the values that we had in the set experiments. In both vector operations, our approach is still faster than the alternatives. However, since the scenario of these two vector operations is very similar to the set intersection, we see similar behaviour in the performance of our approach

Query	SQL Code

Q4	`select o_orderpriority, count(*) from orders`
	`where exists (select * from lineitem`
	`  where l_orderkey=o_orderkey and l_commitdate<l_receiptdate)`
	`group by o_orderpriority`

Q8	`select o_year, sum(case when nation = 'BRAZIL' then volume`
	`  else 0 end) / sum(volume)`
	`from (select extract(year from o_orderdate) as o_year,`
	`  l_extendedprice * (1 - l_discount) as volume,`
	`  n2.n_name as nation`
	`  from part, supplier, lineitem, orders,`
	`  customer, nation n1, nation n2, region`
	`  where p_partkey = l_partkey and s_suppkey = l_suppkey`
	`  and l_orderkey = o_orderkey and o_custkey = c_custkey`
	`  and c_nationkey = n1.n_nationkey`
	`  and n1.n_regionkey = r_regionkey and r_name = 'AMERICA'`
	`  and s_nationkey = n2.n_nationkey and o_orderdate between`
	`  date '1995-01-01' and date '1996-12-31'`
	`  ) as all_nations`
	`group by o_year`

Q12	`select l_shipmode, sum(case when o_orderpriority = '1-URGENT'`
	`  or o_orderpriority = '2-HIGH' then 1 else 0 end)`
	`  , sum(case when o_orderpriority <> '1-URGENT'`
	`  and o_orderpriority <> '2-HIGH' then 1 else 0 end)`
	`from orders, lineitem`
	`where o_orderkey = l_orderkey`
	`group by l_shipmode`

Table 5.2: Modified TPC-H queries we used in the experiments.

versus `phmap::flat_hash_map`. We explained the reason behind this behaviour in the set experiments. The experiments on set and vector operations show that our approach is a great choice when we deal with large amounts of data; while the performance of other approaches degrades with increasing the hash table size, our approach maintains good performance even for heavier workloads.

Analytical Queries Use Case. As the last set of experiments, we use the TPC-H benchmark and dataset. The queries we consider satisfy three criteria. (1) The join-build side of the query must result in a large hash table. (2) The join-probe part of the query must be a time-consuming part of it. (3) The build-side hash table can only have integers as keys and values. Based on these criteria we selected a modified version of Q4, Q8, and Q12 (cf. Table 5.2). Then, we used an automatically generated version of them in C++ by taking the query plans of HyPer [60] and using the integration approach discussed in 5.7. We used `phmap::flat_hash_map` as the competitor for

	TPC-H Query					
	Q4		Q8		Q12	
	Total	**Probe**	**Total**	**Probe**	**Total**	**Probe**
1-Core Run Time `vec_ht` (ms)	260	112	233	222	385	282
1-Core Run Time `phmap` (ms)	304	157	445	431	813	641
1-Core Total Run Time Speedup	**1.17**×	**1.4**×	**1.91**×	**1.94**×	**2.11**×	**2.27**×
4-Core Run Time `vec_ht` (ms)	191	48	91	79	187	31
4-Core Run Time `phmap` (ms)	131	40	139	121	366	243
4-Core Total Run Time Speedup	**0.69**×	**0.83**×	**1.53**×	**1.53**×	**1.96**×	**7.84**×

Table 5.3: Performance improvements of using `vec_ht::lp_map` instead of `phmap::flat_hash_map` in the probes of the most time-consuming hash-join of TPC-H queries 4, 8, and 12.

`vec_ht::lp_map`, because it is the hash table of choice behind existing query processing systems that use open source hash tables [74, 78]. We alternate between these two hash tables only in the most time-consuming join operation in the query.

Table 5.3 shows the results of this benchmark. Our approach performs better than its alternative almost in all these queries. However, in the 4-core setup of Q4, it resulted in a performance degradation. In this case, the speedup of probing could not be further improved (almost equal to the non-vectorized version) but the lack of parallel insertions in Vec-HT resulted in a faster hash table building by the competitor, which is a promising direction for the future.

```
inline size_t find_batch_inner (uint32_t* keys, uint32_t* payloads,
   size_t size, bool complement,
                                FUNC_TYPE func, iter_batch* res_iter,
                                size_t tid)
{
    // Partitioning the input keys into group-sized batches
    size_t inner_batch_size = group_size_;
    for(size_t i=0; i<size; i+=group_size_)
    {
        if (size-i<group_size_)
          inner_batch_size = size%group_size;
        // Hash memoization and Group prefetching
        uint32_t hashs[inner_batch_size];
        for (size_t j=0; j<inner_batch_size; j+=vector_size)
        {
            // Hash computation for keys using SIMD operations
            // ...
            // Hash memoization and Group prefetching
            for (size_t k=0; k<vector_size; k++)
            {
                // Storing the hash in hashs[j+k]
                // ...
                // Prefetching the computed and stored hash
                _mm_prefetch(&entries_[hashs[j+k]], _MM_HINT_T0);
            }
        }
        // Execution of vert. vect. using memoized hash values
        // considering the "complement" flag (if enabled) ...

        // Buffering the matched keys, values (and payloads) ...

        // Flushing the buffer into the iter_batch container or
        // Applying the lambda function on the buffered tuples ...
    }
}
```

Figure 5.14: A simplified representation of the `find_batch_task` implementation in `lp_map`.

```
ord_indexed = ord.sum(lambda p:
            {
                p[0].o_orderkey: p[0].o_orderpriority
            }, dict_size = 1500000)

li_probed = li.sum(lambda p:
                    {
                        record({"l_shipmode": probeDictKey.l_shipmode}):
                        record(
                                {
                                    "high_line_count":
                                        1
                                    if
                                        (indexedDictValue == "1-URGENT")
                                        or
                                        (indexedDictValue == "2-HIGH")
                                    else
                                        0,
                                    "low_line_count":
                                        1
                                    if
                                        (indexedDictValue != "1-URGENT")
                                        and
                                        (indexedDictValue != "2-HIGH")
                                    else
                                        0,
                                }
                        )
                    }
                    if
                        (
                            (p[0].l_shipmode == "MAIL")
                            or
                            (p[0].l_shipmode == "SHIP")
                        ) and
                        (p[0].l_receiptdate >= 19940101) and
                        (p[0].l_receiptdate < 19950101) and
                        (p[0].l_shipdate < p[0].l_commitdate) and
                        (p[0].l_commitdate < p[0].l_receiptdate)
                    else
                        {}
                )
```

Figure 5.15: SDQL.py version of the primary join in TPC-H Q12

```
ord_indexed = ord.sum(lambda p:
              {
                  p[0].o_orderkey: index(p)
              }
              , dict_type = "vecht"
              , dict_size = 1500000)

li_probed = li.sum(lambda p:
                  ifbuf(
                      p,
                      (
                          (p[0].l_shipmode == "MAIL")
                          or
                          (p[0].l_shipmode == "SHIP")
                      ) and
                      (p[0].l_receiptdate >= 19940101) and
                      (p[0].l_receiptdate < 19950101) and
                      (p[0].l_shipdate < p[0].l_commitdate) and
                      (p[0].l_commitdate < p[0].l_receiptdate),
                      lambda buf:
                          ord_indexed.vfind(
                              buf,
                              lambda q: q[0].l_orderkey).sum(
                              lambda r:
                                  {
                                      record({"l_shipmode": r[0].l_shipmode}):
                                      record(
                                          {
                                              "high_line_count":
                                                  1
                                              if
                                                  (r[0].o_orderpriority == "1-URGENT")
                                                  or
                                                  (r[0].o_orderpriority == "2-HIGH")
                                              else
                                                  0,
                                              "low_line_count":
                                                  1
                                              if
                                                  (r[0].o_orderpriority != "1-URGENT")
                                                  and
                                                  (r[0].o_orderpriority != "2-HIGH")
                                              else
                                                  0,
                                          }
                                      )
                                  }
                          )
                  )
              )
```

Figure 5.16: VSDQL.py version of the primary join in TPC-H Q12

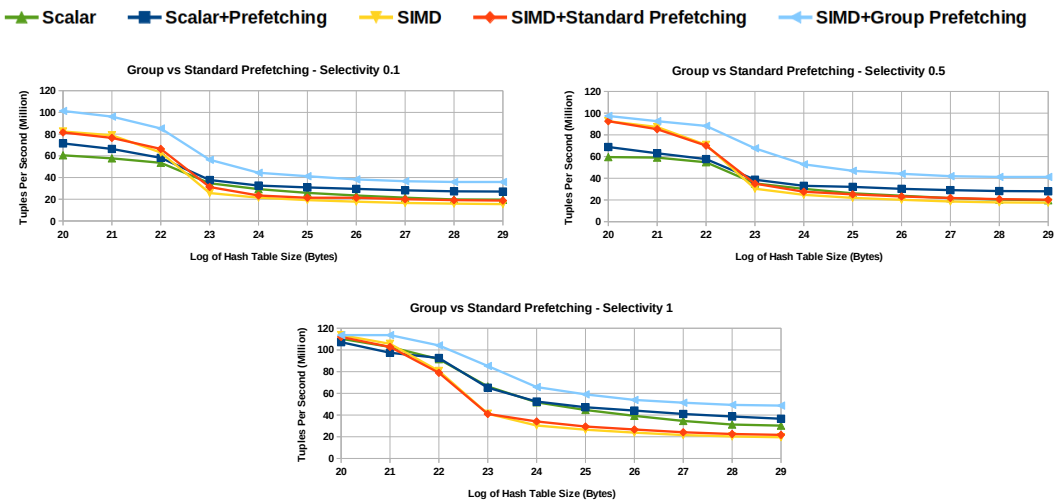


Figure 5.17: Comparing the performance of standard vs group prefetching for combining prefetching with vertical vectorization. The charts show the number of tuples processed per second (higher is better) for a simplified relational inner-join operation with a probe size of 1,500,000 over an increasing build-side size.

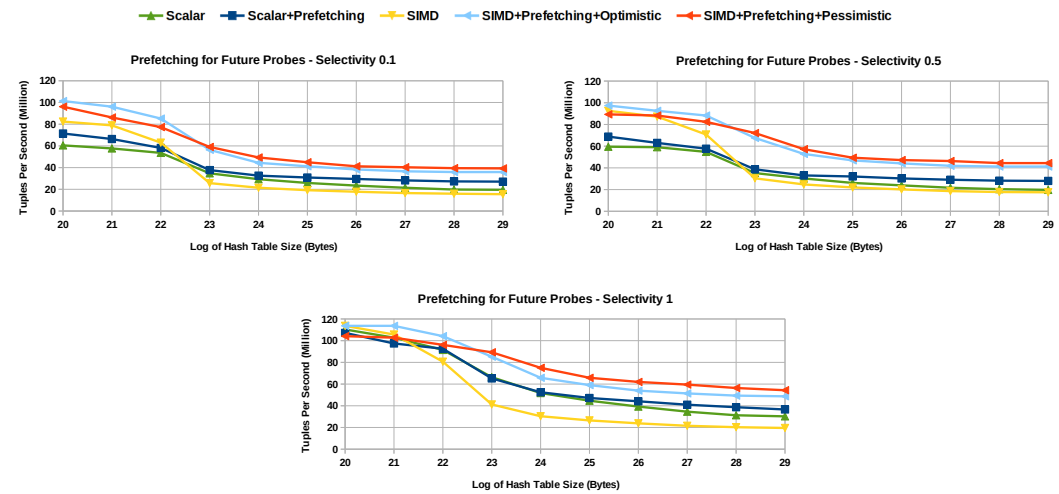


Figure 5.18: Comparing the performance of different alternatives by focusing on the optimistic and pessimistic approaches for prefetching. The workload is similar to Figure 5.17.

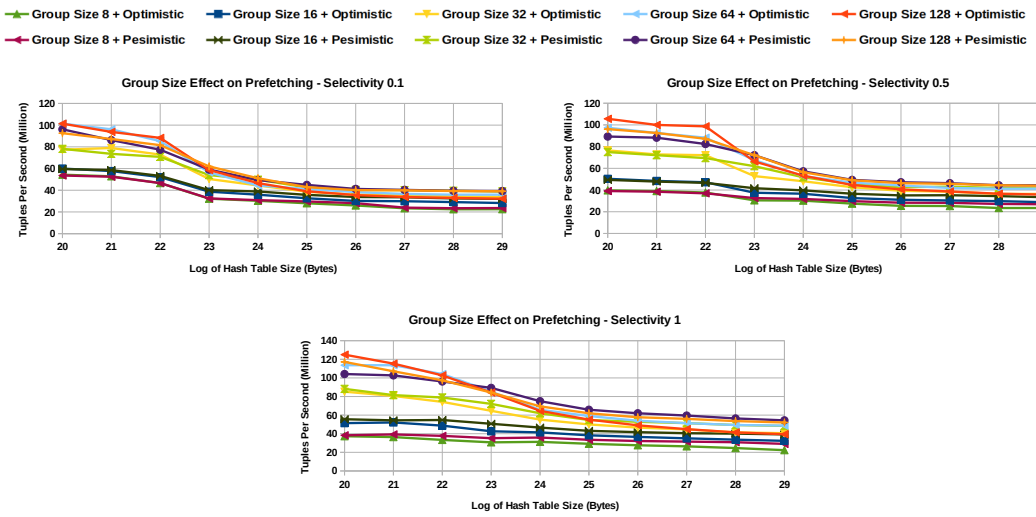


Figure 5.19: Comparing the performance of different combinations for group size and optimistic/pessimistic approaches for group prefetching. The workload is similar to Figure 5.17.

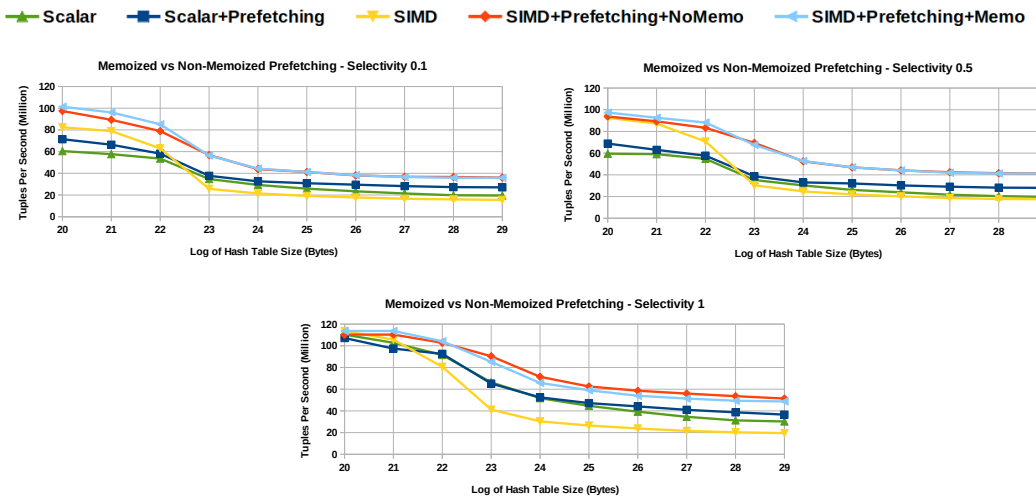


Figure 5.20: Comparing the performance of different alternatives by focusing on enable/disabled memoization for group prefetching. The workload is similar to Figure 5.17.

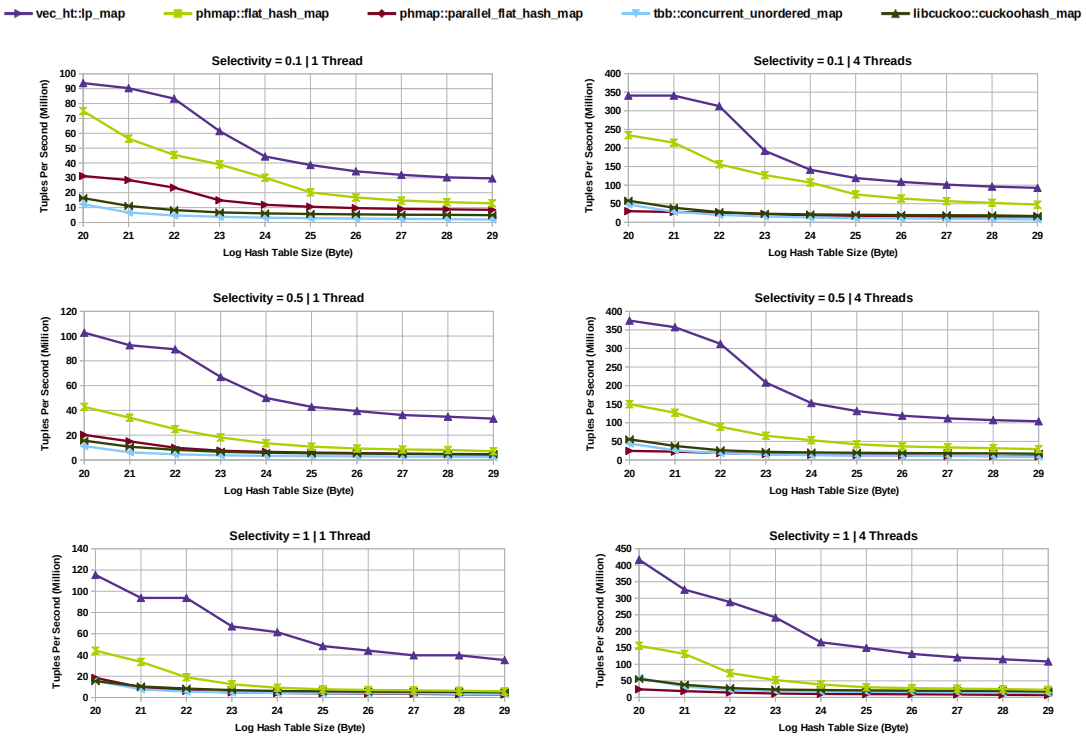


Figure 5.21: The number of tuples processed per second (higher is better) for a simplified relational inner-join operation with a probe size of 1,500,000 over an increasing build-side size. Charts on each side, represent the results for the join selectivities of 0.1, 0.5, and 1 on 1 or 4 threads.

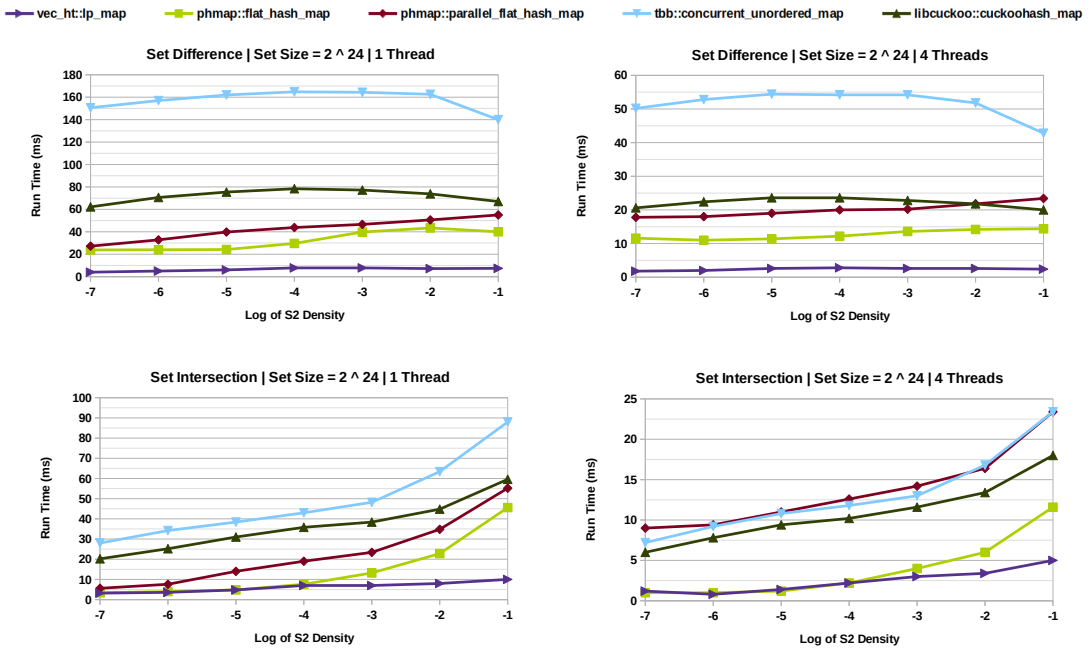


Figure 5.22: The run time (lower is better) for the set difference and intersection operations by varying the density of the second set on 1 and 4 threads. For both operands (S1 and S2), the size is 2^{24}. For S1, the density is set to 2^{-6}.

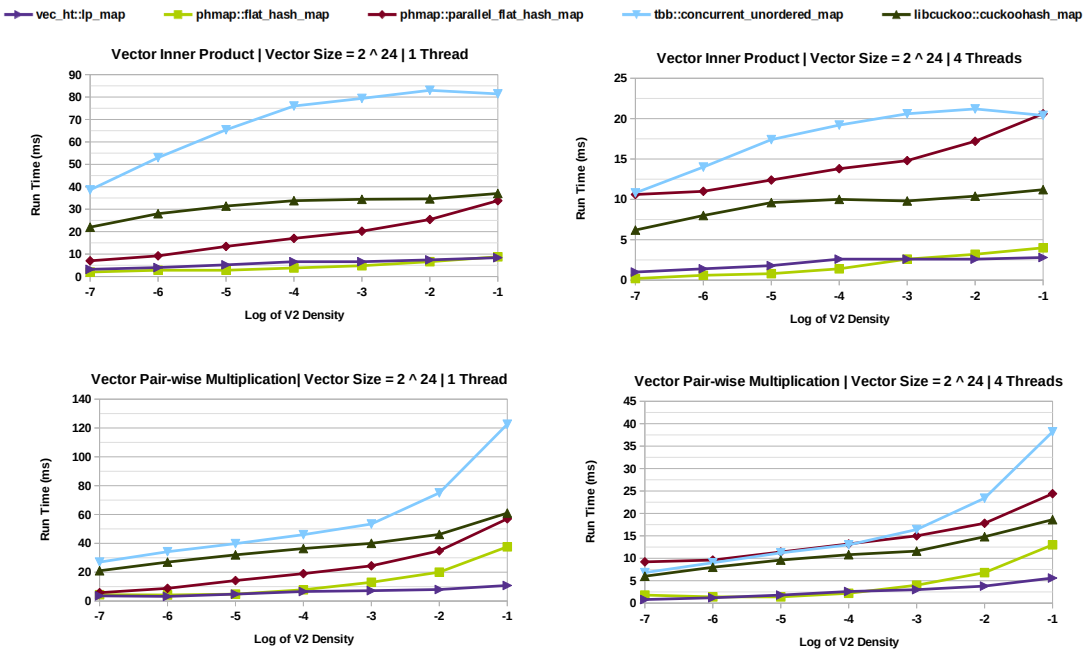


Figure 5.23: The run time (lower is better) for the vector inner-product and pair-wise multiplication operations by increasing the density of the second vector on 1 and 4 threads. For both operands (V1 and V2), the size is 2^{24}. For V1, the density is set to 2^{-6}.

Chapter 6

Related Work

In this section, we review state-of-the-art research related to the thesis topic.

Python to SQL. Approaches in this category [27, 26, 42, 38, 34, 63] aim to push the execution down into existing database engines by generating SQL code from source code. By taking this approach, the effort is mostly on a precise translation and then leveraging the proven optimization and execution power of databases. ByePy [38] automatically generates SQL for generic imperative Python code with no support for its popular libraries such as Pandas and NumPy. Blacher et al. [26] proposed their translation methods based on training simple machine learning models (e.g., logistic regression). They experimentally show the merits of a database-friendly (dense) data layout versus the COO (sparse) one for numerical analysis, however, their approach is not automated. In another work, Blacher et al. [27] presented a method to generate SQL for linear algebra workloads written in Einstein notation (`einsum` API in NumPy). While their approach can cover n-ary `einsum` and higher-order tensors, it can only operate on the data stored in a sparse format which, according to their experiments, is not always efficient. Grizzly [42] and PyFroid [34] generate SQL for Pandas. Our reconstruction efforts show that Grizzly APIs are not fully consistent with their Pandas equivalents and require the user to modify the source code. Moreover, Grizzly falls short in capturing different query operators, resulting in a limited coverage of standard TPC-H queries. It is also important to mention that its generated queries are not idiomatic SQL; they do not get the best performance from query engines. PyFroid [34] covers a broader set of APIs and incorporates optimizations on the generated code, the codebase of which is not accessible for experimentation.

Python to Low-Level Code. Solutions in this category [50, 85, 62, 48] generate the low-level equivalent of Python code (e.g. in C++ or LLVM) that will be compiled for the target machine. Numba [50] generates efficient low-level code for numerical operations written using the NumPy library. Numba users only need to add `@jit` decorator to their target functions to make them compiled. Tuplex [85] and Klabe et. al [48] propose code generation for UDFs (User Defined Functions) in Python. Weld [62] generates LLVM code by lazy construction of an IR after each library API call. As opposed to Numba and PyTond, Weld users need to slightly modify the library APIs to make them adaptable to its compilation pipeline. Daphne[33] provides an embedded DSL in Python for transforming integrated analytics to low-level code via MLIR. However, it lacks a compiler for translating Python data science code like Pandas and NumPy into its DSL, making user adaptation, benchmarking, and comparison difficult. LingoDB [45] introduces an MLIR-based framework that converts various interface

languages, particularly SQL, into low-level efficient code. While they demonstrate a use-case of their approach with a PyTorch example, to the best of our knowledge, there is currently no available implementation of a Python interface for LingoDB for comparison purposes. Another related research [44] around LingoDB has only shown limited support of translation from Pandas (three TPC-H queries), which contrasts with PyTond's broad coverage.

The mentioned approaches have shown performance improvements by leveraging a variety of optimizations (e.g. parallelization and vectorization). Nevertheless, they introduce compilation overhead and potential portability issues. They also need users to declare input types, a requirement for low-level code generation, which can reduce the overall user-friendliness of the approach.

Python Parallelization. Modin [63] introduces a Dataframe algebra designed to encapsulate Pandas code and enable parallel execution. However, its reliance on Pandas as the backend limits achieving optimal performance on each core.

IR for Data Science. There have been recent proposals on IRs for data science. DAPHNE [33] captures integrated data analysis (IDA) workloads with its own DSL and converts them to LLVM after multi-layer compilations based on MLIR. Weld [62] exposes a common IR aimed at cross-domain optimizations. However, there is no evidence of full support for the operators required for analytical queries, e.g., TPC-H queries. As an alternative IR, IFAQ [81, 82] and SDQL [78] can be used in their systems.

Vectorized/Parallel Query Execution. MonetDB [43] and DuckDB [66] are examples of DBMSes that use this technique. DuckDB is specially designed as an embedded vectorized DBMS. Fent et al. [37] have done a study on the best parallel designs for group-by and aggregation operators. Focusing on the join operator, Bandle et al. [24] investigated the effects of partitioning on the performance of this operator. Leis et al. [51] have proposed the morsel-driven parallelism to adapt query compilers to NUMA architectures. Menon et al. [57] use relaxed fusion to combine vectorization and compilation. The trade-offs between vectorization and compilation are investigated in Typer and Tectorwise [47].

Batch Hash Tables. Among the open-source projects [2, 4, 5], Hirola [4] presents a fast batch hash table written in C which is an alternative for `dict/set` in Python. Similarly, since most values in the R language are vectors and matrices, R-hashmap[5] presents a batch hash table for R, which is built over some existing ordinary hash tables in C. These two libraries are only using the cache-locality of the batch input as a way of

performance improvement. As a more advanced open-source project, DPDK [2] offers a specific-purpose batch hash table for networking use cases. It is both SIMD-aware and prefetch-enabled.

There is also state-of-the-art research [29, 35, 53, 64, 68, 71, 88] on batch hash tables. Some of the approaches [35, 53, 88] are only using batch processing to benefit from the cache locality and prefetching while others [29, 64, 68, 71] use SIMD techniques on a vector of inputs. Similar to DPDK [2], Horton [29] and Cuckoo++[71] have focused on improving the performance of batch hash tables by applying SIMD and prefetching techniques to a specific type of SIMD-aware batch hash table designs called Bucketized Cuckoo Hash Tables (BCHTs). On the other hand, Polychroniou et al. [64] present a generic design for SIMD-aware batch hash tables and compare the performance of different design decisions by doing intensive benchmarks. To answer some of the open questions in vectorized hash tables, [83] conducted a survey on state-of-the-art and conducted micro-benchmarks to position each work with respect to the others.

Prefetching-Enabled Hash Tables. The effects of using software prefetching in hash tables have been studied in [32, 71, 88]. In the networking community, Scouarnec et al. [71] and Zhou et al. [88] have shown the effects of using prefetching on network-specific batch hash tables. They proposed different approaches and improvements in applying prefetching on BCHTs. In the database community, by proposing two generic techniques of using prefetching, Chen et al. [32] have shown their impact on the performance of relational hash joins. Among the off-the-shelf open-source hash tables, we found phmap [10] as the one that provides a `prefetch_hash` API in its interface. However, it delegates the responsibility of using this API (and designing a good prefetching strategy) to the developer who is not necessarily a system-level developer. There exist challenges in combining software prefetching with vectorization in the context of hash tables. We cover these challenges and their related design decisions in Section 5.4.3.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented a framework for enhancing the efficiency of Python data science workloads using compilation and code generation. We showed that this framework captures the data science workloads from relational algebra and linear algebra domains (written using Pandas and NumPy libraries, respectively), converts them to a custom IR (Tond IR), optimizes them, and generates efficient SQL code that can be executed on the existing powerful relational query engines. Moreover, by taking the query plan of the generated SQL code, we showed how we craft specialized and fine-tuned query engines for the given input. This can be done by converting the plan to another IR (SDQL IR), optimizing the IR, and generating C++ code. Lastly, to make the specialized query engines even faster, we presented our design for embedding batch processing into our generated C++ code. We used a combination of parallelization, vectorization, and prefetching techniques in our proposed batch-processing strategies. We also evaluated our pipelines' performance using various micro and end-to-end benchmarks.

In our framework, we also highlighted the choice of decorator-based JIT compilation as a strategic design decision that improves the usability of our approach. It empowers users to use the framework without applying changes to their workloads or imported libraries. It also facilitates switching between prototyping and production modes. This high level of usability could help in a better adaption of the framework by users.

In summary, as we aimed for, our proposed framework can help data scientists develop their code in the popular Python ecosystem while gaining the benefits of fast and scalable computation with a minimal overhead of change or learning. It also reduces the need for increasing computing resources while working on big data which is an important step towards green and sustainable computing.

7.2 How to Use the Framework?

We open-sourced the main parts of our approach to make it accessible to interested researchers and practitioners. Users who are interested in the compilation of Python data science code into SQL can use the open-source version of PyTond[1] on GitHub. They can also refer to the open-source version of sdqlpy[2] for Python to C++ compilation. The instructions for running each tool are well-documented on their related GitHub page. It is worth noting that in both approaches, users can write their Python functions as usual and test them in Python's interpreted environment. Then, they can accelerate the execution of their verified code, only by adding the related function decorator (@sdql/@pytond).

[1] https://github.com/edin-dal/pytond
[2] https://github.com/edin-dal/sdqlpy

7.3 Future Work

The different directions and opportunities for the extension or improvement of this work are discussed below:

- The framework can be extended to capture other types of Python data science workloads with iterations, recursions, nested data, sparse linear algebra, and advanced numeric computations (e.g. higher-order tensor computations).
- Other vectorization opportunities such as vectorized selections, joins, and aggregates can also be embeded into the generated C++ code of specialized engines.
- The proposed vectorized hash table can also be developed to support complex key/values and parallel iterations that are required for broader workload coverage.

Bibliography

[1] Dask. https://dask.org.

[2] Dpdk. https://dpdk.org/.

[3] Highwayhash. https://arxiv.org/abs/1612.06257.

[4] Hirola. https://github.com/bwoodsend/hirola/.

[5] R hashmap. https://github.com/nathan-russell/hashmap.

[6] Ray. https://www.ray.io/.

[7] ska::flat_hash_map, 2018. https://github.com/skarupke/flat_hash_map.

[8] Intel® oneapi: Threading building blocks (tbb), 2021. https://github.com/oneapi-src/oneTBB.

[9] folly::AtomicUnorderedMap, 2022. https://github.com/facebook/folly.

[10] The parallel hashmap, 2022. https://github.com/greg7mdp/parallel-hashmap.

[11] robin_hood::flat_hash_map, 2022. https://github.com/martinus/robin-hood-hashing.

[12] tsl::hopscotch_map, 2022. https://github.com/Tessil/hopscotch-map.

[13] tsl::robin_map, 2022. https://github.com/Tessil/robin-map.

[14] Numpy einsum api, 2024. https://numpy.org/doc/stable/reference/generated/numpy.einsum.html.

[15] Numpy library, 2024. https://numpy.org/.

[16] Pandas library, 2024. https://pandas.pydata.org/.

[17] Ponder, 2024. https://ponder.io/.

[18] SQLite, 2024. https://www.sqlite.org.

[19] Tpc-h benchmark, 2024. https://www.tpc.org/tpch/.

[20] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented DBMS. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 466–475. IEEE Computer Society, 2007. doi:10.1109/ICDE.2007.367892.

[21] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[22] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/.

[23] Maximilian Bandle and Jana Giceva. Database technology for the masses: Suboperators as first-class entities. *Proc. VLDB Endow.*, 14(11):2483–2490, 2021. URL: http://www.vldb.org/pvldb/vol14/p2483-bandle.pdf, doi:10.14778/3476249.3476296.

[24] Maximilian Bandle, Jana Giceva, and Thomas Neumann. To partition, or not to partition, that is the join question in a real system. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 168–180. ACM, 2021. doi:10.1145/3448016.3452831.

[25] Ronald Barber, Guy M. Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi K. Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. Memory-efficient hash joins. *Proc. VLDB Endow.*, 8(4):353–364, 2014. URL: http://www.vldb.org/pvldb/vol8/p353-barber.pdf, doi:10.14778/2735496.2735499.

[26] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. Machine learning, linear algebra, and more: Is SQL all you need? In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p17-blacher.pdf.

[27] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. Efficient and portable einstein summation in SQL. *Proc. ACM Manag. Data*, 1(2):121:1–121:19, 2023. doi:10.1145/3589266.

[28] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237. www.cidrdb.org, 2005. URL: http://cidrdb.org/cidr2005/papers/P19.pdf.

[29] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. Horton tables: Fast hash tables for in-memory data-intensive computing. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 281–294. USENIX Association, 2016. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/breslow.

[30] Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin hood hashing (preliminary report). In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 281–288. IEEE Computer Society, 1985. doi:10.1109/SFCS.1985.48.

[31] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium*

on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA, pages 77–90. ACM, 1977. doi:10.1145/800105.803397.

[32] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17, 2007. doi:10.1145/1272743.1272747.

[33] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina M. Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios I. Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaz Kosar, Alexander Krause, Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Marcus Paradies, Bernhard Peischl, Gabrielle Poerwawinata, Stratos Psomadakis, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pinar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz P. Wrosz, Ales Zamuda, Ce Zhang, and Xiaoxiang Zhu. DAPHNE: an open and extensible system infrastructure for integrated data analysis pipelines. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf.

[34] K. Venkatesh Emani, Avrilia Floratou, and Carlo Curino. Pyfroid: Scaling data analysis on a commodity workstation. In Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani, editors, *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, pages 61–67. OpenProceedings.org, 2024. URL: https://doi.org/10.48786/edbt.2024.06, doi:10.48786/EDBT.2024.06.

[35] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 371–384. USENIX Association, 2013. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan.

[36] Azadeh Farzan and Victor Nicolet. Synthesis of divide and conquer parallelism for loops. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 540–555. ACM, 2017. doi:10.1145/3062341.3062355.

[37] Philipp Fent and Thomas Neumann. A practical approach to groupjoin and nested aggregates. *Proc. VLDB Endow.*, 14(11):2383–2396, 2021. URL: http://www.vldb.org/pvldb/vol14/p2383-fent.pdf, doi:10.14778/3476249.3476288.

[38] Tim Fischer, Denis Hirn, and Torsten Grust. Snakes on a plan: Compiling python functions into plain SQL queries. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 2389–2392. ACM, 2022. doi:10.1145/3514221.3520175.

[39] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

[40] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994. doi:10.1109/69.273032.

[41] Tim Gubner and Peter A. Boncz. Charting the design space of query execution using VOILA. *Proc. VLDB Endow.*, 14(6):1067–1079, 2021. URL: http://www.vldb.org/pvldb/vol14/p1067-gubner.pdf, doi:10.14778/3447689.3447709.

[42] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. Putting pandas in a box. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021_paper07.pdf.

[43] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012. URL: http://sites.computer.org/debull/A12mar/monetdb.pdf.

[44] Robert Imschweiler. Transforming data frame operations from python to mlir. Master's thesis, Technische Universität München, 2022.

[45] Michael Jungmair, André Kohn, and Jana Giceva. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow.*, 15(11):2389–2401, 2022. URL: https://www.vldb.org/pvldb/vol15/p2389-jungmair.pdf, doi:10.14778/3551793.3551801.

[46] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015. URL: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper8.pdf.

[47] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, 2018. URL: http://www.vldb.org/pvldb/vol11/p2209-kersten.pdf, doi:10.14778/3275366.3275370.

[48] Steffen Kläbe, Robert DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. Accelerating python udfs in vectorized query execution. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p33-klaebe.pdf.

[49] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 613–624. IEEE Computer Society, 2010. doi:10.1109/ICDE.2010.5447892.

[50] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python JIT compiler. In Hal Finkel, editor, *Proceedings of the Second Work-*

shop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015, pages 7:1–7:6. ACM, 2015. doi:10.1145/2833157.2833162.

[51] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754. ACM, 2014. doi:10.1145/2588555.2610507.

[52] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In Dick C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel, editors, *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 27:1–27:14. ACM, 2014. doi:10.1145/2592798.2592820.

[53] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 1–13. ACM, 2011. doi:10.1145/2043556.2043558.

[54] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general(?)! *ACM Trans. Parallel Comput.*, 5(4):16:1–16:32, 2019. doi:10.1145/3309206.

[55] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general (?)! *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–32, 2019.

[56] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. Compiling without continuations. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 482–494. ACM, 2017. doi:10.1145/3062341.3062380.

[57] Prashanth Menon, Amadou Ngom, Todd C. Mowry, Andrew Pavlo, and Lin Ma. Permutable compiled queries: Dynamically adapting compiled queries

without recompiling. *Proc. VLDB Endow.*, 14(2):101–113, 2020. URL: http://www.vldb.org/pvldb/vol14/p101-menon.pdf, doi:10.14778/3425879.3425882.

[58] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, 2017. URL: http://www.vldb.org/pvldb/vol11/p1-menon.pdf, doi:10.14778/3151113.3151114.

[59] Paul Mooney. 2020 kaggle machine learning & data science survey, 2020. URL: https://kaggle.com/competitions/kaggle-survey-2020.

[60] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011. URL: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf, doi:10.14778/2002938.2002940.

[61] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. URL: https://doi.org/10.1016/j.jalgor.2003.12.002, doi:10.1016/J.JALGOR.2003.12.002.

[62] Shoumik Palkar, James Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. A common runtime for high performance data analysis. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. URL: http://cidrdb.org/cidr2017/papers/p127-palkar-cidr17.pdf.

[63] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. Towards scalable dataframe systems. *Proc. VLDB Endow.*, 13(11):2033–2046, 2020. URL: http://www.vldb.org/pvldb/vol13/p2033-petersohn.pdf.

[64] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD vectorization for in-memory databases. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May*

31 - June 4, 2015, pages 1493–1508. ACM, 2015. doi:10.1145/2723372.2747645.

[65] Mihai Puatracscu and Mikkel Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):14:1–14:50, 2012. doi:10.1145/2220357.2220361.

[66] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1981–1984. ACM, 2019. doi:10.1145/3299869.3320212.

[67] Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9(3):96–107, 2015. URL: http://www.vldb.org/pvldb/vol9/p96-richter.pdf, doi:10.14778/2850583.2850585.

[68] Kenneth A. Ross. Efficient hash probes on modern processors. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 1297–1301. IEEE Computer Society, 2007. doi:10.1109/ICDE.2007.368997.

[69] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018. URL: http://arxiv.org/abs/1805.00907, arXiv:1805.00907.

[70] Maximilian Schleich, Amir Shaikhha, and Dan Suciu. Optimizing tensor programs on flexible storage. *Proc. ACM Manag. Data*, 1(1):37:1–37:27, 2023. doi:10.1145/3588717.

[71] Nicolas Le Scouarnec. Cuckoo++ hash tables: high-performance hash tables for networking applications. In Theophilus Benson and Noa Zilberman, editors, *Proceedings of the 2018 Symposium on Architectures for Networking and Commu-*

nications Systems, ANCS 2018, Ithaca, NY, USA, July 23-24, 2018, pages 41–54. ACM, 2018. doi:10.1145/3230718.3232629.

[72] Hesam Shahrokhi, Callum Groeger, Yizhuo Yang, and Amir Shaikhha. Efficient query processing in python using compilation. In Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia, editors, *Companion of the 2023 International Conference on Management of Data, SIGMOD-/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 199–202. ACM, 2023. doi:10.1145/3555041.3589735.

[73] Hesam Shahrokhi, Amirali Kaboli, Mahdi Ghorbani, and Amir Shaikhha. Pytond: Efficient python data science on the shoulders of databases. In *Proceedings of the 40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, Netherlands, May 13-17, 2024*. IEEE, 2024.

[74] Hesam Shahrokhi and Amir Shaikhha. Building a compiled query engine in python. In Clark Verbrugge, Ondrej Lhoták, and Xipeng Shen, editors, *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Montréal, QC, Canada, February 25-26, 2023*, pages 180–190. ACM, 2023. doi:10.1145/3578360.3580264.

[75] Hesam Shahrokhi and Amir Shaikhha. An efficient vectorized hash table for batch computations. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 27:1–27:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2023.27, doi:10.4230/LIPICS.ECOOP.2023.27.

[76] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push versus pull-based loop fusion in query engines. *J. Funct. Program.*, 28:e10, 2018. doi:10.1017/S0956796818000102.

[77] Amir Shaikhha, Andrew W. Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing style for efficient memory management. In Phil Trinder and Cosmin E. Oancea, editors, *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing,*

FHPC@ICFP 2017, Oxford, UK, September 7, 2017, pages 12–23. ACM, 2017. doi:10.1145/3122948.3122949.

[78] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–33, 2022. doi:10.1145/3527333.

[79] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. Building efficient query engines in a high-level language. *ACM Trans. Database Syst.*, 43(1):4:1–4:45, 2018. doi:10.1145/3183653.

[80] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1907–1922. ACM, 2016. doi:10.1145/2882903.2915244.

[81] Amir Shaikhha, Maximilian Schleich, Alexandru Ghita, and Dan Olteanu. Multi-layer optimizations for end-to-end data analytics. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*, pages 145–157. ACM, 2020. doi:10.1145/3368826.3377923.

[82] Amir Shaikhha, Maximilian Schleich, and Dan Olteanu. An intermediate representation for hybrid database and machine learning workloads. *Proc. VLDB Endow.*, 14(12):2831–2834, 2021. URL: http://www.vldb.org/pvldb/vol14/p2831-shaikhha.pdf, doi:10.14778/3476311.3476356.

[83] Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K. D. K. Panda. Simdht-bench: Characterizing simd-aware hash table designs on emerging CPU architectures. In *IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019*, pages 178–188. IEEE, 2019. doi:10.1109/IISWC47752.2019.9042069.

[84] Daniel G. A. Smith and Johnnie Gray. opt_einsum - A python package for optimizing contraction order for einsum-like expressions. *J. Open Source Softw.*, 3(26):753, 2018. URL: https://doi.org/10.21105/joss.00753, doi:10.21105/JOSS.00753.

[85] Leonhard F. Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. Tuplex: Data science in python at native code speed. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1718–1731. ACM, 2021. doi:10.1145/3448016.3457244.

[86] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. How to architect a query compiler, revisited. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 307–322. ACM, 2018. doi:10.1145/3183713.3196893.

[87] Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. Local: a language for programs operating on serialized data. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 48–62. ACM, 2019. doi:10.1145/3314221.3314631.

[88] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In Kevin C. Almeroth, Laurent Mathy, Konstantina Papagiannaki, and Vishal Misra, editors, *Conference on emerging Networking Experiments and Technologies, CoNEXT '13, Santa Barbara, CA, USA, December 9-12, 2013*, pages 97–108. ACM, 2013. doi:10.1145/2535372.2535379.