PolySem: Efficient Polyglot Analytics on Semantic Data

Xinyu Liu[1], Venkatesh Emani[2,*], Avrilia Floratou[2], Joyce Cahoon[2], Philip Seamark[2] and Carlo Curino[2]

[1]*Georgia Tech. University (work done while at Microsoft)*
[2]*Microsoft*

Abstract
Data scientists and data engineers spend a significantly large portion of their time trying to understand, clean and transform their data before they can even start performing any meaningful analysis. Most database vendors provide business intelligence (BI) tools as an efficient and user friendly platform for customers to perform their data cleaning, preparation and linking tasks to obtain actionable *semantic data*. However, customers are increasingly interested in querying semantic data through various query modalities including SQL, imperative programming languages such as Python, and natural language queries. Currently, customers are limited to using either the visual interfaces provided by these tools or languages that are specific to the particular tool. In this proposal, we describe techniques to enable the execution of user queries expressed in different modalities on semantic datasets without having to export data out of the BI system. Our techniques comprise of automatic translation of user queries into a language-agnostic representation of data processing operations, and subsequently to the specific query language that is amenable to execution on the BI engine. Our evaluation results on business intelligence and decision support benchmarks suggest significant improvements in query performance compared to other popular data processing engines.

Keywords
semantic data, pandas, natural language, dax, sql

1. Introduction

Most popular database and cloud vendors provide business intelligence (BI) tools as an efficient and user friendly platform for customers to perform data cleaning, linking, visualization and reporting. Examples of such tools include Microsoft Power BI [1], Amazon QuickSight [2], Tableau [3], and others. These tools are popular among customers as they offer a “low-code/no-code” alternative to SQL-based systems for users to quickly start exploring their datasets and deriving meaningful insights.

These tools simplify data preparation by providing users a visual drag and drop interface (with optional textual querying capabilities) to perform operations such as linking tables, filtering,

Poly'23: Polystore systems for heterogeneous data in multiple databases with privacy and security assurances, Sep 01, 2023, Vancouver, Canada
*Corresponding author.
liuxy@gatech.edu (X. Liu); venkatesh.emani@microsoft.com (V. Emani); avflor@microsoft.com (A. Floratou); jcahoon@microsoft.com (J. Cahoon); phseamar@microsoft.com (P. Seamark); carlo.curino@microsoft.com (C. Curino)

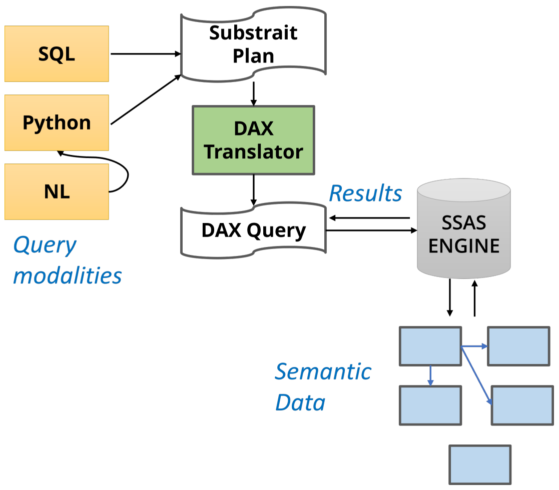


Figure 1: PolySem Architecture

projections, joins, sorting, grouping, aggregation, plotting and others. Further, users have the ability to add compound column information by combining existing columns and also retrieve related information (such as longitude to time zone mapping). The prepared datasets are stored in a linked tabular representation – referred to as *semantic data* – where relationships between various tables are baked into the representation. Working with semantic data greatly simplifies queries for users as they do not need to specify which tables to join in each query – the data processing engine leverages semantic information present in the underlying representation to automatically join the datasets appropriately and efficiently.

Data cleaning, preparation and linking requires a significant amount of time and expertise and users often want to reuse these datasets as much as possible for all their analysis tasks. Increasingly, customers are interested in querying semantic data through other query modalities such as SQL, imperative programming languages like Python, and natural language (NL) interfaces. However, currently, access to these datasets is either (a) limited to the visual interfaces provided by these BI tools, (b) enabled through tool-specific query languages, or (c) provided by a tedious and insecure mechanism of getting the prepared data out of the BI system.

In this paper, we propose to alleviate this hurdle by executing queries expressed in popular programming languages such as Python, SQL or natural language directly on semantic data. An overview of our system is shown in Figure 1. Our work is in the lines of prior work on computation pushdown, however, our differentiated contributions are in supporting multiple query languages for semantic data through a common intermediate framework.

Using Microsoft's Power BI as an exemplar business intelligence tool, which is powered by Vertipaq (also called SSAS [4]) as the underlying query execution engine, and DAX (data analysis expressions [5]) as the query language, we develop techniques that automatically translate queries queries expressed in different modalities into the DAX query language, which is amenable to to execution on the Vertipaq engine. Our techniques comprise of first translating the query into a language-agnostic representation called a Substrait [6] , which is an emerging standard for specifying relational data processing operations that can represent most tabular data processing operations efficiently using protocol buffers. Substrait representations (or *plans*) are then optimized using pattern matching rules that we develop, and translated into DAX queries for efficient execution on the Vertipaq engine.

Our techniques have multiple benefits. Firstly, our system has practical utility as we do not require users to make any changes to their existing workloads to leverage our work. The PolySem system automatically translates these queries into DAX for efficient processing on Vertipaq. Secondly, users do not need to move semantic data out – the data processing happens in-situ, using tools that users already are subscribed to. This provides security as well as lowers costs for users. Thirdly, we support queries expressed in various languages and query abstractions (declarative SQL, imperative Python and natural language), so a wide variety of personas can leverage our system including data scientists, data engineers and business leaders. Finally, our techniques open up a broad new set of users that are traditionally unfamiliar with languages such as DAX, thus enabling a wider audience to benefit from these systems.

We evaluated the benefits of PolySem on various benchmarks and compared the performance of PolySem with other single node analytical engines such as DuckDB and Pandas. Our evaluation results (Section 5) show that our solution outperforms competing solutions in many benchmark queries, with speed ups over 100x.

In summary, our contributions in this paper are as follows:

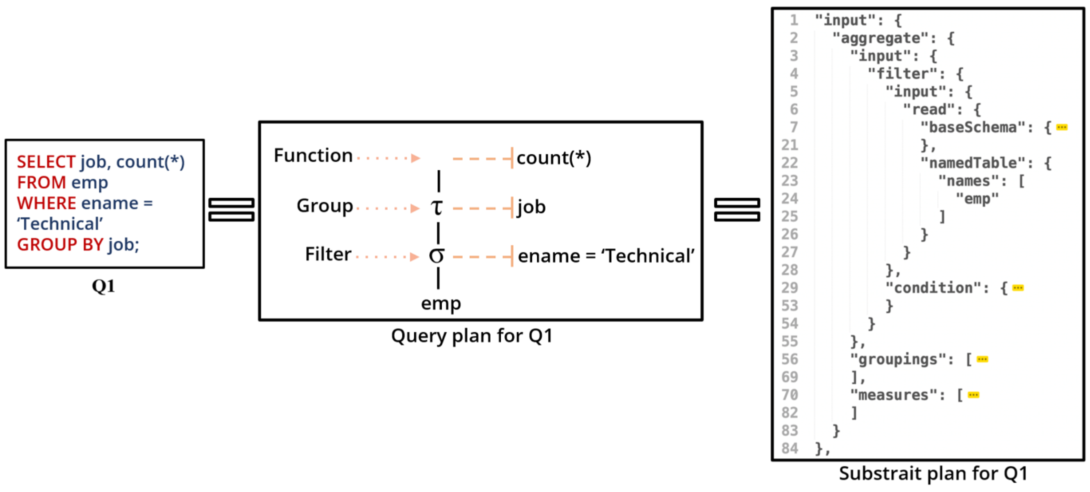
- Given a query specified as a Substrait plan, we develop techniques and a tool – PolySem – that implements these techniques to automatically translate the plan into a DAX query that is amenable to execution by the Vertipaq engine. Our techniques comprise of algorithms (Section 3) to generate optimized DAX queries by examining the structure of the Substrait plan.
- We discuss how our tool enables queries written in SQL, Python and natural language to run on Vertipaq (Section 4).
- We conduct a comparative analysis (Section 5) of our tool with Pandas and DuckDB on representative benchmark queries and present encouraging results.

2. Background

Before we dive into the details of our techniques, we present an overview of the Substrait specification and the Vertipaq engine.

2.1. Substrait

Substrait [6] is a standard representation for query plans. It denotes *what should be done* to the data in a language-agnostic manner. Substrait provides relational built-in support for relational algebraic operations (select, project, join, aggregation, group by, order by, etc.) and common types (int, float, string, etc.) with an extension framework that enables representation of user defined functions and custom operators. The main use case of Substrait is to communicate the query plan between a query parser and an execution engine. Substrait uses protocol buffers for serializing query plans. Protocol buffers are a highly efficient binary format for transmitting data over the network.



```
"input": {
  "aggregate": {
    "input": {
      "filter": {
        "input": {
          "read": {
            "baseSchema": {
            },
            "namedTable": {
              "names": [
                "emp"
              ]
            }
          }
        },
        "condition": {
        }
      }
    },
    "groupings": [
    ],
    "measures": [
    ]
  }
},
```

Figure 2: Sample Substrait Plan

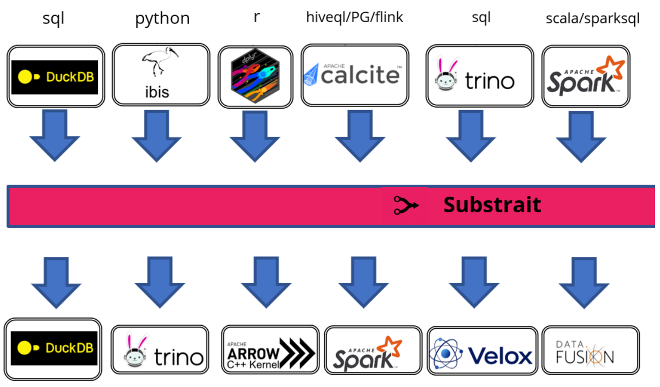


Figure 3: Substrait Adoption

Substrait plans can be examined for debugging using a JSON representation. Figure 2 shows a simple query, its relational algberaic representation and Substrait (JSON) representation.

Substrait plans may be generated by query interfaces as a machine readable representation of the queries, and consumed by data processing engines, which are responsible for parsing the Substrait plan and executing the specified operations. Substrait is gaining adoption rapidly and many data processing engines have added support for Substrait plans. Figure 3 shows a summary of Substrait's reach in popular open source and competing data processing engines. We note that by virtue of using a versatile and backend-agnostic representation such as Substrait, our techniques are easily extensible to other engines apart from Vertipaq.

2.2. Vertipaq

Vertipaq is the engine underlying various business intelligence tools provided by Microsoft, including Power BI and Excel Power Query. It provides a drag and drop visual interface for users to explore, analyze and report their data. Under the hood, these visual operations are translated into a query language such as DAX, and executed using the Vertipaq engine which is included as part of the Power BI platform. The linked tabular data is stored in Power BI efficiently using a compressed columnar format. Vertipaq contains techniques to directly perform query operations on the compressed datasets without decompressing it, leading to orders of magnitude improvements in query performance. The DAX language is targeted towards business intelligence queries and contains special optimized constructs for common BI query patterns such as filtering followed by grouped aggregation.

3. Techniques

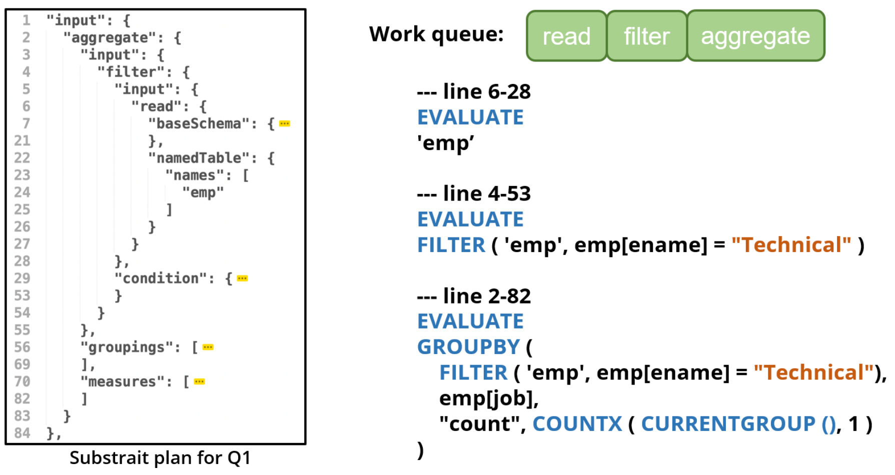
As mentioned in Section 1, our techniques comprise of two steps: translating the initial query into a Substrait plan, and then, translating the Substrait plan into a DAX query. Our system takes care of executing the DAX query on the Vertipaq engine and returning the results to the user in an appropriate format (for example, as a Pandas dataframe if the initial query was submitted as a Python Pandas script). We first discuss how a Substrait plan is translated into a DAX query. Then, we discuss how various query modalities can be supported through Substrait as the intermediate representation.

3.1. Substrait to DAX

The algorithm to translate a Substrait plan into DAX consists of two passes on the plan and can be summarized in the following steps:

- Step 1: Create a worklist (queue) by parsing the Substrait plan and extracting an ordered list of relational operators. This worklist enables PolySem to generate DAX queries using a bottom-top approach. That is, PolySem first generates an inner query and then build the outer query on top of the inner query.
- Step 2: Repeat Steps 3 through 5 if the worklist is not empty, otherwise go to Step 6.
- Step 3: Pop the first element from the worklist. This element represents the innermost relational operator to process.
- Step 4: Generates a DAX query based on two elements: (1) the element received from Step 3. (2) the DAX query generated from the previously executed Step 5 (if available).
- Step 5: Rewrite the DAX query from Step 4 into a more optimized one by using a set of query rewrite rules (described below).
- Step 6: Return the DAX query generated from Step 5 as output.

The algorithm above is illustrated in Figure 4.



```
1  "input": {
2    "aggregate": {
3      "input": {
4        "filter": {
5          "input": {
6            "read": {
7              "baseSchema": {⋯
21             },
22             "namedTable": {
23               "names": [
24                 "emp"
25               ]
26             }
27           }
28         },
29         "condition": {⋯
53         }
54       }
55     },
56     "groupings": [⋯
69     ],
70     "measures": [⋯
82     ]
83   }
84 },
```

Substrait plan for Q1

```
--- line 6-28
EVALUATE
'emp'

--- line 4-53
EVALUATE
FILTER ( 'emp', emp[ename] = "Technical" )

--- line 2-82
EVALUATE
GROUPBY (
   FILTER ( 'emp', emp[ename] = "Technical"),
   emp[job],
   "count", COUNTX ( CURRENTGROUP (), 1 )
)
```

Figure 4: Illustration of PolySem Query Generation Algorithm

Query optimizations

We observed in our experimental evaluation that a direct translation of Substrait plans into DAX queries results in suboptimal DAX queries. Unlike more sophisticated database engines like Postgres, SQL Server and others, which contain hundreds of query optimization rules thereby rewriting even complex queries into an optimal query rewrite, performance of DAX queries on Vertipaq is affected by the query structure. Thus, it is essential to generate an optimized query before submitting it for execution. We now describe two of our optimizations targeted at join optimization and groupby optimization. Section 5 will discuss the ramifications of these optimizations.

As mentioned in Step 5 of our algorithm, PolySem performs transformations to optimize the generated query. Our transformations are query rewrite rules tailored for DAX: for each rewrite rule, it rewrites a DAX query into a more optimized form if the query matches a certain pattern. If the DAX query does not match the certain pattern, the rule leaves the query unchanged. Rewrite rules are necessary because some DAX queries are more performant than others, even though they may be semantically equivalent. In our current implementation, we include two such rewrite rules, described below.

- Rule 1: Join optimization
  Input pattern:

  ```
  EVALUATE
  GENERATE (
    table1,
    FILTER (table2, table1.primary_key = table2.foreign_key)
  )
  ```

  Output pattern:

  ```
  EVALUATE
  GENERATE (table1, RELATEDTABLE (table2))
  ```

- Rule 2: GroupBy optimization
  Input pattern:

  ```
  EVALUATE
  SUMMARIZE (
    FILTER (
      GENERATE (
        table1,
        FILTER (table2, table1.primary_key = table2.foreign_key)
      ),
      <condition on table2.column3>
    ),
    table1.column1,
    table2.column2,
    <agg_name>, <agg_metric>
  )
  ```

  Output pattern:

  ```
  EVALUATE
  SUMMARIZECOLUMNS (
    table1.column1,
    table2.column2,
    FILTER (
      ALL (table2.column3),
      <condition on table2.column3>
    ),
    <agg_name>, <agg_metric>
  )
  ```

  where agg_name is any name for the aggregated column and agg_metric is an aggregate like SUM, MAX, etc. that uses any column from table2.

We observed that Rules 1 and 2 are practical in many decision support and business intelligence benchmark queries. We note, however, that this list is not exhaustive; we are exploring additional rules that may be incorporated.

4. Integrating Query Modalities

In this section, we briefly discuss how our techniques enable translation of queries in different languages into DAX.

- **SQL**: SQL is ubiquitously used by data engineers and developers of database-backed applications. For popular database query dialects such as Spark SQL, DuckDB etc., there are interfaces that can translate these queries into Substrait plans. For other engines including SQL Server, efforts are underway to enable this. Once the SQL query is translated into a Substrait plan, our tool proceeds with DAX translation as shown in Figure 1.

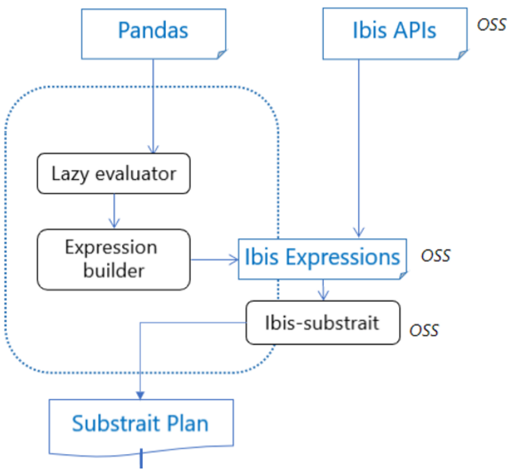


Figure 5: Architecture of Python to Substrait Translation

- **Imperative languages**: Most imperative programming languages provide libraries for data processing. Techniques to translate such imperative programs into declarative representations [7] can be used to translate them into Substrait plans. For one such language, namely Python and a library, namely Pandas, we build on the in-house developed PyFroid [8] translation library to generate Substrait plans. Pandas to Substrait translation happens in two steps: (1) translating Pandas APIs into Ibis [9] representations using PyFroid and (2) translating Ibis representations into Substrait using a publicly available library [10]. We note that alternatively, for Python users using the Ibis APIs, we could simply use step 2 to translate the program into Substrait. Again, once the Substrait plan is available, DAX translation happens as shown in Figure 1. The architecture for translating Python programs (Pandas or Ibis APIs) into Substrait is shown in Figure 5.
- **Natural Language**: For users who are not familiar with a programming language and/or would like to express their queries using simple English (natural language) sentences, our system provides a natural language (NL) query interface for semantic data. Recent advances in large language models (LLMs) such as Codex [11] have demonstrated powerful capabilities for generating Python code from natural language descriptions, even as their ability to generate accurate code for low resource languages such as DAX is evolving. We leverage these advances to first generate Python code from NL descriptions. From Python, we generate Substrait plans using our techniques discussed in the previous bullet point. Subsequently, Substrait is translated to DAX and executed on Vertipaq – thus realizing the goal of running natural language queries on Vertipaq. We note that as stronger accuracy guarantees are available for translation of NL to DAX using LLMs, we can use these models to directly generate DAX queries from NL descriptions without going through Python and Substrait.

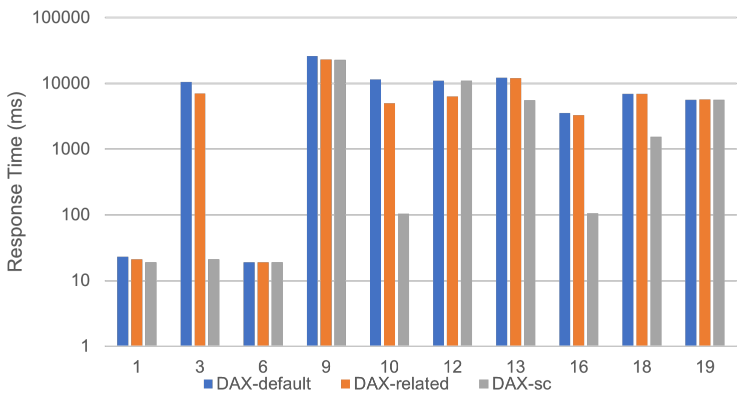


Figure 6: Impact of Rewrite Rules (TPC-H benchmark)

5. Evaluation

Before diving into the experiments, we first discuss our evaluation setup. We run all of our experiment on the same machine: it has 128G RAM and a CPU that has 10 cores and 20 threads. We evaluate PolySem on two benchmarks: the first one is TPC-H, which is a standard performance benchmark for relational databases; The second one is Public BI, as its name suggests, it is a benchmark for business intelligent. So, it targets a different kind of dataset and queries than TPC-H. Which, in our assumption, represents a scenario that PolySem could be best suited in. In our evaluation, we use the response time as the only metric for query performance. To do this, we issue queries from a client and measure the time it takes to receive the result.

For the TPC-H benchmark that we use, we use a dataset that has a scale factor of 1 since our target is datasets that comfortably fit on a single node. We select 10 out of 22 TPC-H queries. The remaining ones are the queries that PolySem can't support for now. Nevertheless, we think these 10 queries still cover a wide variety of SQL query operators and structures. For operators, they cover aggregations, sort, filter, case, and limit. For query structures, they cover joins, subquery and nested query. For the Public-BI benchmark, we randomly select one of a dataset, named "TrainsUK" , in our evaluation. The size of the dataset is around 1.3 GB. The schema is relatively simple compared to the TPC-H: it only has 1 table and does not have any index. The dataset comes with 5 queries, with relatively simple query operator and structures. They cover aggregations, sort, filter, and case operators. For query structures, they don't cover joins, subquery and nested query. We now discuss our results across our evaluation goals – one in each following subsection.

5.1. Impact of Rewrite Rules

We want to see whether the optimization rules that we introduce can help the DAX translator generates a more optimized query than the default ones. We call the default queries as DAX-default. Queries that use the RELATEDTABLE construct as DAX-related. Queries that use the SUMMARIZECOLUMNS constructs as DAX-sc. The results are shown in Figure 6.

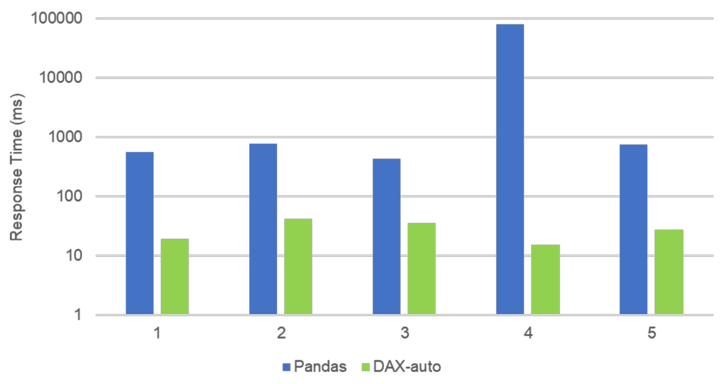


Figure 7: Comparison with Pandas (BI benchmark)

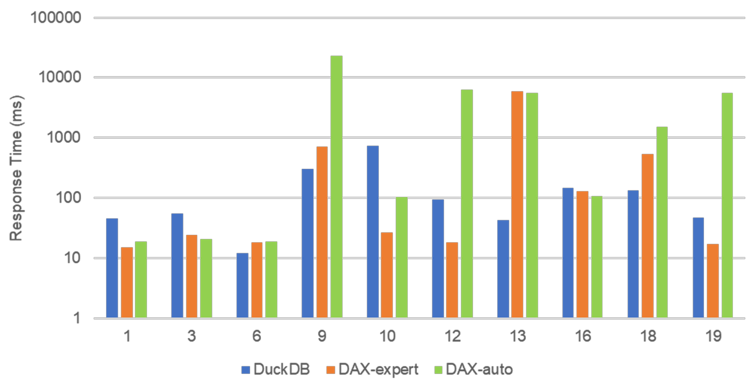


Figure 8: Comparison with DuckDB (TPC-H benchmark)

This plot shows the query response time on three versions of DAX queries. The x-axis shows the TPC-H query number and y-axis shows the total query runtime in log scale. The key observation from the graph is that there are 6 cases wherein either one of our optimization rules are applicable, the resulting queries always run faster than the default ones, with a speed up ranging from 1.8 times up to 498.4 times. In cases wherein both optimization rules are applicable, DAX-sc always runs faster than DAX-related.

5.2. Comparison with Other Engines

In our last set of experiments, we compare the performance of PolySem with two popular single node systems – Pandas and DuckDB. The results are shown in Figures 7 and 8. In summary, our solution outperforms Pandas in most cases with speed ups ranging from 2x-700x, and outperforms DuckDB in 20% of considered benchmark queries. As we see, for many queries, the expert-written DAX queries (DAX-expert) are significantly better than the queries auto-generated by our tool (DAX-auto); we are working on improving our system to close this gap.

References

[1] pbi, What is power bi, 2023. URL: https://learn.microsoft.com/en-us/power-bi/fundamentals/power-bi-overview.

[2] aqi, Amazon quicksight, 2023. URL: https://aws.amazon.com/quicksight/.

[3] tableau, Business intelligence and analytics software, 2023. URL: https://www.tableau.com/.

[4] vertipaq, Vertipaq - brain and muscles behind power bi, 2023. URL: https://data-mozart.com/vertipaq-brain-muscles-behind-power-bi/.

[5] dax, Dax reference, 2023. URL: https://learn.microsoft.com/en-us/dax/.

[6] substrait, Substrait: Cross-language serialization for relational algebra, 2022. URL: https://substrait.io/.

[7] K. Ramachandra, K. Park, K. V. Emani, et al., Froid: Optimization of imperative programs in a relational database, VLDB 11 (2017) 432–444.

[8] A. Jindal, K. V. Emani, M. Daum, O. Poppe, B. Haynes, A. Pavlenko, A. Gupta, K. Ramachandra, C. Curino, A. Mueller, W. Wu, H. Patel, Magpie: Python at speed and scale using cloud backends, 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_paper08.pdf.

[9] ibis, Ibis-substrait compiler, 2022. URL: https://github.com/ibis-project/ibis-substrait.

[10] ibis-substrait, Ibis-substrait compiler, 2022. URL: https://github.com/ibis-project/ibis-substrait.

[11] codex, Openai codex, 2022. URL: https://openai.com/blog/openai-codex/.