Evaluating query languages and systems for high-energy physics data

Dan Graur[1], Ingo Müller[1], Mason Proffitt[2,3], Ghislain Fourny[1], Gordon T. Watts[2,3] and Gustavo Alonso[1]

[1] ETH Zürich, Stampfenbachstrasse 114, 8006 Zürich, Switzlerand
[2] University of Washington, 3910 15th Ave. NE, Seattle, WA 98195-1560
[3] CERN, Espl. des Particules 1, 1211 Meyrin, Switzlerand

E-mail: {dan.graur, ingo.mueller, gfourny, alonso}@inf.ethz.ch,
{masonLp, gwatts}@uw.edu

Abstract. In the domain of high-energy physics (HEP), general-purpose query languages have found little adoption in analysis. This is surprising regarding SQL-based systems, as HEP data analysis matches SQL's processing model well: the data is fully structured and makes use of predominantly standard operators. To better understand the situation, we select six general-purpose query engines, from both the SQL and NoSQL domain, and analyze their performance, scalability, and usability in HEP analysis, employing standard HEP tools as baseline. We also identify a set of core language features needed to support HEP data analysis. Our results reveal an interesting and complex picture: several query languages provide a rich and natural query development experience, while others fall short. In terms of performance, our results reveal that many of the database systems are one or two orders of magnitude slower than the standard HEP analysis tools, while others manage to scale and perform well. These conclusions suggest that while the existing data processing systems are viable candidates for HEP analysis, there is still work to be done to improve their performance and ability to succinctly express HEP queries.

1. Introduction

In spite of the well-known advantages of modern-day data processing systems (data independence, declarative languages, etc.), such systems have seen little adoption in the HEP community for analysis use-cases. Instead, the HEP community mostly relies on a domain-specific system called ROOT [1, 2] and, more recently, its novel RDataFrames API [3]. While ROOT has indeed seen widespread adoption, its design ties together the storage format and the file system, the in-memory runtime format, the execution strategy, the target platform, and even the visualization. Such dependencies inhibit the development of parts of the system without affecting seemingly unrelated components, and make it difficult to transparently optimize queries at runtime — similarly to query plan optimizations carried out in databases, which are enabled by declarative languages.

Highly processed datasets, as predominantly used in HEP analyses, are always fully-structured and represent events as rows. Each event contains both scalar attributes and attributes that store relatively large sequences of values. The latter attribute type denormalizes the data into non-first normal form (NF^2). Such a data representation was one of the reasons why SQL-based query engines initially found little adoption in the HEP community for analysis

use-cases —in addition to the lack of support for User-Defined Functions (UDF) [4]—, as early engines did not provide support for nested data types. The database community has since made major strides, and many relational database systems provide mature support for processing nested attributes and defining custom user logic. Moreover, several NoSQL solutions are now available, which provide flexible data models as well as powerful and intuitive querying languages.

To test whether these new systems are viable solutions for HEP data processing, in terms of both performance, scalability, and querying languages, we evaluate six general-purpose querying engines using RDataFrames as the baseline. For the study, we employ the Analysis Description Languages (ADL) benchmark [5]. We choose two cloud-based Query-as-a-Serivce (QaaS) systems: BigQuery [6] and Amazon Athena [7], and four self-hosted systems: Presto [8], Postgres [9], RumbleDB [10], and AsterixDB [11]. The former two are representatives of conventional database systems, while the latter two represent document-oriented systems. Note that Athena is a fork of Presto, but the two systems have significant differences. BigQuery, Athena, Presto, and Postgres each implement an SQL dialect stemming from the SQL standard. RumbleDB implements the JSONiq query language [12]. AsterixDB implements SQL++ [13], an SQL-inspired query language designed to support queries on semi-structured data.

We structure our paper as follows: We briefly discuss the benchmark and its queries in Section 2. We discuss some of the important query patterns as well as their applicability in the various systems we choose in Section 3. We study the performance and scalability of the chosen systems in Section 4, present related work in Section 5, and finally provide our concluding remarks and future work in Section 6.

For the full and detailed analysis and evaluation results, the reader can consult the extended version of this abstract [14].

2. Dataset and queries

The ADL benchmark consists of eight high-level query descriptions, representative of HEP data analysis. The benchmark is designed in order to understand the suitability of systems for HEP analyses, and guide the development of the next generation of HEP tools. The ADL dataset consists of roughly 54 million events obtained from the Compact Muon Solenoid (CMS) 2012 experiment [15]. A full description of the queries and the dataset is available at [5, 14]. Our query implementations are all open source [5, 16]. The dataset is originally in ROOT format and is approximately 17 GiB in size. We use the benchmark at version v0.1. As no system, apart from RDataFrames, can natively read the ROOT format, we convert the data to Parquet, as it offers a similar compression ratio and accurate data representation to ROOT. We use the Parquet version for those systems that cannot read ROOT.

ROOT represents the fields of structures and arrays as individual columns both physically and logically. In contrast, in the Parquet version, we choose to represent the structures and arrays logically together, while storing them column-wise physically. This provides a more natural view of the data to the practitioner, making querying more intuitive, with no compromise regarding storage format and performance.

3. Query patterns

We briefly discuss some of the main patterns that occur frequently in the ADL queries. For brevity reasons, we omit many of the details and the code examples shown in the full work [14].

3.1. Accessing and creating nested structures

Accessing and creating nested structures is a critical part of HEP queries. It allows practitioners to combine atomic attributes (e.g., p_T, η, ϕ, etc.) into logical structures (e.g., an object representing a muon) that make querying easier. Listing 1 shows the main ways of creating structs in the studied systems. RDataFrames does not provide native support for structures,

```
df.Define("Jet_p3", make_p3, {
  "Jet_pt", "Jet_eta", "Jet_phi"})
```

(a) RDataFrames

```
STRUCT<x INT64, y FLOAT64>(
  a.x + b.x, 42.0))
```

(b) BigQuery.

```
CAST(ROW(a.x + b.x, 42.0)
  AS ROW(x BIGINT, y DOUBLE))
```

(c) Presto / Athena

```
CAST(ROW((a).x + (b).x, 42.0)
  AS userDefinedPair)
```

(d) Postgres

```
{ "x": $a.x + $b.x, "y": 42.0 }
```

(e) JSONiq

```
SELECT {"x": a.x + b.x, "y": 42.0};
```

(f) SQL++

Listing 1: Accessing and creating nested structs.

but does allow for arbitrary C++ defined types. The user must define and use their own UDF (e.g. `make_p3` in Listing 1a) which returns a `ROOT::RVec` type. BigQuery allows for the direct use of the `STRUCT` keyword. The types and the names of the fields can be optionally omitted, thus creating an 'anonymous' struct. Presto, Athena, and Postgres use a combination of `CAST` and `ROW`. Among these three systems, Athena cannot access the fields of anonymous rows. JSONiq and SQL++ allow the user to create structs via the square brackets {...} operator, where the practitioner specifies the name and values of the fields. The types of the fields are inferred based on the values. In all cases, struct fields are accessed via the `.` operator.

3.2. Accessing and creating nested arrays

Operating on arrays of structs is a core requirement of a query system working on HEP analyses. Listing 2 shows parts of the implementation of ADL Query 4 across the systems studied. The query finds the events having at least two jets with $p_T > 40$ GeV.

RDataFrames provides a sizable library of vectorized operations that work directly on arrays of structs (e.g. the comparison operator `>`). BigQuery, Presto, Athena and Postgres all employ a combination of `CROSS JOIN` and `UNNEST`. As its name suggests, `UNNEST` breaks up the array into its individual elements. `CROSS JOIN` takes each individual element produced by `UNNEST` and generates a new row from it, replicating all other fields of the initial row and adding a new column containing the aforementioned element. SQL++ also features a more concise mechanism which only requires `UNNEST`. JSONiq offers a more intuitive way of accessing and filtering elements. First, a stream of `events` is generated via the `for $event in $events` instruction. Then, the `jets` entry is extracted from each `event`. `jets` is then flattened into a list of jet structs via the `[]` operator. Finally all jets with an unsuitable p_T value are filtered out via `[$$.pt > 40]`. JSONiq uses the powerful FLWOR expression set, composed of the `for`, `let`, `where`, `order by`, `return` clauses. The semantics of FLWOR are similar to SQL's `SELECT-FROM-WHERE`, where one seemingly iterates through the entire dataset and applies various filtering operations. FLWOR provides, what initially appears to be an imperative programming model, that is in fact declarative under the hood. This can feel intuitive to practitioners experienced with the imperative programming paradigm. In Listing 2f we show a potential implementation for SQL++ which employs the `ARRAY_LENGTH` function together with a subquery that is directly embedded in the function call itself.

3.3. Complex queries and particle combinations

HEP queries are generally more complex and may require the inspection of particles tuples, which can be of the same type, different type, or both, in those cases where larger particle

```
df.Filter("Sum(Jet_pt > 40) > 1")
```

(a) RDataFrames

```
... WHERE (SELECT COUNT(*)
      FROM UNNEST(events.Jets) AS j
      WHERE j.pt > 40) > 1
```

(b) BigQuery/Postgres with nested sub-query

```
SELECT event_id, MET.sumet
FROM events
CROSS JOIN UNNEST(events.Jets) AS j
WHERE j.pt > 40
GROUP BY event_id, MET.sumet
HAVING COUNT(*) > 1
```

(c) Presto/Athena/Postgres with `CROSS JOIN`

```
... WHERE CARDINALITY(FILTER(
  events.Jets, j -> j.pt > 40)) > 1
```

(d) Presto/Athena using array functions

```
for $event in $events
where count(
  $event.jets[][$$.pt > 40]) > 1
...
```

(e) JSONiq

```
... WHERE ARRAY_LENGTH((
        SELECT * FROM e.Jets AS j
        WHERE j.pt > 40)) > 1
```

(f) SQL++.

Listing 2: Querying unnested array elements

tuples are generated. Consequently, SQL queries often have to use a `GROUP BY` clause which undoes the `CROSS JOIN` and `UNNEST` clauses, a step which feels counter-intuitive and inefficient.

More complex queries also have a negative effect on RDataFrames, where complex logic for vectorized operations needs to be defined in UDFs written in C++ outside of the RDataFrames' system logic. While some of the other studied systems, such as Athena or Presto lack in terms of UDF support, engines like BigQuery, Postgres, AsterixDB, and RumbleDB offer mature UDF support. For instance, RumbleDB allows the practitioner to define UDFs in source files that can later be imported by arbitrary query files. A downside, however, of some SQL systems is that one cannot define tables as valid input or return types in UDFs.

3.4. Summary

We present the results of our usability study in Table 1. Each row specifies a required feature identified by us for HEP analyses. Requirements are encoded as R$i.j$, where i denotes their importance and j is an index. Requirements $R1.j$ are critical, $R2.j$ are secondary, while $R3.j$ are tertiary. The full paper [14] gives a more precise definition of each feature. Missing features are marked with '-'. We also indicate how well a system does in a specific feature using '$*$' characters. Finally, we present several other metrics, such as the number of characters, lines, clauses, etc. required to implement all the queries in the studied systems. While this quantification cannot be interpreted as fully objective, we have, to the best of our abilities, striven to produce a complete image of the usability requirements of HEP analyses and the degrees of suitability of the studied systems.

We conclude that JSONiq and SQL++ are best suited for HEP analysis, which is not entirely surprising as they have been designed for the nested, semi-structured JSON data model. BigQuery and Postgres are suitable candidates, too, as they provide a rather complete implementation of the SQL standard, with few proprietary extensions that would make porting between systems challenging. RDataFrames is also a good candidate, which is also not entirely surprising as it is designed for these kinds of workloads. However, coupling columnar storage with the RDataFrames programming model requires more effort when writing queries. Athena is not a viable candidate due to its lack of UDF support. Similarly, Presto is lacking in a large number of areas, including UDFs, making it an unsuitable candidate for HEP data analysis. Overall,

Table 1: Summary of functionality of general-purpose data processing systems for HEP analyses. "-", "*", "**", and "***" stand for no, poor, moderate and good feature support respectively.

	Athena	**BigQuery**	**Postgres**	**Presto**	**JSONiq**	**SQL++**	**RDataFrame**
(R1.1) unnest arrays	**	**	**	**	***	***	**
(R1.2) asym. combinations	***	***	***	**	***	***	**
(R1.3) sym. combinations	***	***	***	**	***	***	**
(R1.4) UDFs	-	**	**	*	***	***	***
(R2.1) structured types	**	***	**	**	***	***	**
(R2.2)nested sub-query	-	***	***	-	***	***	**
(R2.3) variables	-	-	-	-	***	**	***
(R2.4) group by variable	-	***	***	-	***	**	n/a
(R2.5) struct params in UDFs	*	**	***	**	***	***	***
(R2.6) tables in UDFs	-	-	-	-	***	***	-
(R3.1) inline struct types	-	***	-	-	***	***	-
(R3.2) anonymous structs	**	***	**	***	-	-	-
(R3.3) user-defined types	-	-	**	-	-	***	***
(R3.4) array functions	**	**	**	***	**	**	**
(R3.5) array construction	-	**	**	-	***	***	**
(R3.6) unnest whole structs	***	***	***	-	***	***	-
#characters	6.7k	7.6k	7.6k	7k	**3.8k**	**3.8k**	11k
#lines	343	280	286	274	**106**	175	236
#clauses	222	223	205	180	**56**	104	134
avg. #clauses/query	24.6	16	17	19	**6.2**	8.6	14.9
#unique clauses	24	20	20	27	**8**	16	15
avg. #unique clauses/query	12.1	8.3	8.6	10	**3.3**	4.9	7

we believe that the NF^2 support added with SQL:1999 as well as more modern languages like JSONiq and SQL++ have the potential to make general-purpose data processing systems a viable alternative to the domain-specific systems used today.

4. Evaluation

We evaluate the performance of the studied systems on the ADL benchmark queries at various scales of the data. We scale the data size using Scale Factors (SF) 2^i where i takes values from the integer interval $[-16..7]$. For the self-hosted systems, we employ a single AWS EC2 `m5ds.24xlarge` instance, which has 96 logical cores (and 48 physical ones). For BigQuery, we show an additional version of the experiments where the data is pre-loaded. Figure 1 shows the results of this experiment.

Parquet logically splits the data into row groups. In our case, each row group contains around 400k events. At SF=128, the dataset contains around 16k row groups. Since the systems which read from Parquet parallelize over the row groups, it follows that for small SFs, where there is only one row group, execution is implicitly single threaded. As the data sizes become bigger, and more row groups are present, processing can be parallelized. At this point plateaus are formed in the query running time. Once the SF is large enough, and the number of row groups

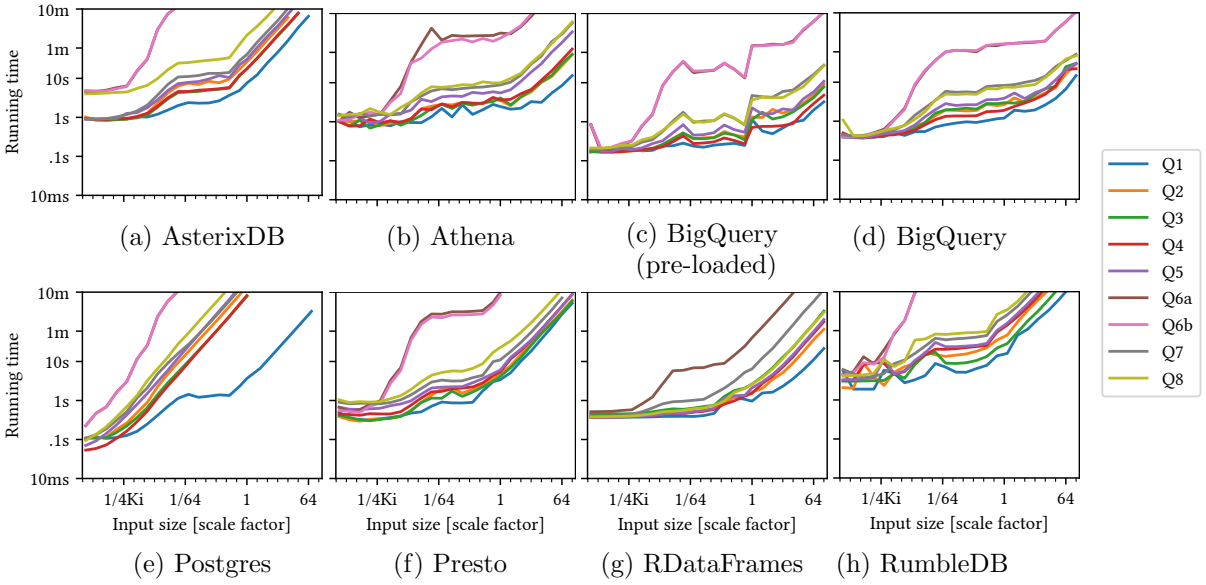

Figure 1: Impact of data size on end-to-end running time of various systems under test.

exceeds the number of virtual cores, the computation stops being fully parallelizable. This is the point where running times see a linear increase in the size of the data. For instance, for QaaS systems, where the hardware resources (and hence level of parallelism) is transparently handled by the cloud provider, we see that the saturation point is around SF=2 for Athena and SF=8 for BigQuery, while for the other systems it occurs at SF=0.5. A notable exception to this pattern is Postgres. In spite of our extensive efforts to parallelize the queries, only Query 1 succeeds at this. The rest end up running on a single core, hence the linear performance of the query runtime relative to data size across all SFs.

In summary, RDataFrames and BigQuery perform best. Presto and Athena also fare well, however their performance lags close to one order of magnitude behind the best systems. AsterixDB, Postgres and RumbleDB visibly underperform as they fail to scale up efficiently. Their performance is close to two orders of magnitude behind the best systems. Postgres displays the worst performance due to its inability to parallelize any query apart for query 1.

We present a complete overview of the experiments in an extended version of the paper [14].

5. Related work

Investigations into the possibility of using database systems for HEP analysis have been done in the past [17]. The HEP community has also investigated the development of novel analysis tools beyond the ROOT ecosystem, such as the Scikit-HEP project [18] as well as alternative data formats such as Parquet or Avro [19]. Finally, Ong et al. [13] survey document oriented query languages, and, from their observations, ultimately derive SQL++, one of the languages employed in this study. We present a more complete picture of the related work in [14].

6. Conclusion

We have evaluated the performance, scalability, and suitability of several query engines and their respective query languages in the context of HEP analyses. For our study, we have used the ADL benchmark, which consists of eight queries that present a high potential of parallelization and exploit the intrinsic nesting of the event data. Our query implementations are all open source [5, 16]. We show that the document-oriented query languages SQL++ and JSONiq provide a rather natural query development experience while several SQL dialects can express

HEP queries in a good manner. We also show that RDataFrames, BigQuery, Presto, and Athena perform and scale well, with the former two systems yielding the best results.

Our study shows that there is plenty of future work for the HEP and database community. Understanding how to make general purpose systems more scalable and efficient on nested homogeneous data is key. Exposing existing HEP libraries (e.g. such as those in ROOT) to general purpose systems is also a promising avenue for future research, as well as understanding how to combine the functionality of query engines with auxiliary features (e.g. graph plotting). Finally, understanding how to best disseminate the use of such systems to the HEP community is essential to their widespread use.

Acknowledgments

We thank Jim Pivarski for establishing the connection among the authors and the various insightful discussions on HEP analyses, as well as the respective developer teams of ROOT and AsterixDB for their timely and thorough replies.

References

[1] Antcheva I, Ballintijn M, Bellenot B, Biskup M, Brun R, Buncic N, et al. ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization. Computer Physics Communications. 2009 dec;180(12).

[2] Brun R, Rademakers F. ROOT — An object oriented data analysis framework. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. 1997;389(1).

[3] Guiraud E, Naumann A, Piparo D. TDataFrame: functional chains for ROOT data analyses. Zenodo; 2017.

[4] Malon DM, May EN. Critical Database Technologies for High Energy Physics. In: VLDB; 1997. .

[5] Proffitt M, Müller I, Adamec M, David P, Guiraud E, Binet S. iris-hep/adl-benchmarks-index: ADL Functionality Benchmarks Index. Zenodo; 2021. Available from: `https://github.com/iris-hep/adl-benchmarks-index/`.

[6] Sato K. An inside look at Google BigQuery; 2012. White paper. Available from: `https://cloud.google.com/files/BigQueryTechnicalWP.pdf`.

[7] Services AW. Amazon Athena;. Available from: `https://aws.amazon.com/athena/`.

[8] Sethi R, Traverso M, Sundstrom D, Phillips D, Xie W, Sun Y, et al. Presto: SQL on everything. In: ICDE; 2019. .

[9] Stonebraker M, Rowe LA. The design of POSTGRES. ACM SIGMOD Record. 1986 jun;15(2).

[10] Müller I, Fourny G, Irimescu S, Cikis CB, Alonso G. Rumble: Data Independence for Large Messy Data Sets. Proc VLDB Endow. 2020 Dec;14(4).

[11] Alsubaiee S, Altowim Y, Altwaijry H, Behm A, Borkar V, Bu Y, et al. AsterixDB: A Scalable, Open Source BDMS. Proc VLDB Endow. 2014 Oct;7(14).

[12] Florescu D, Fourny G. JSONiq: The History of a Query Language. IEEE Internet Computing. 2013;17(5):86-90.

[13] Ong KW, Papakonstantinou Y, Vernoux R. The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases; 2014.

[14] Graur D, Müller I, Proffitt M, Fourny G, Watts GT, Alonso G. Evaluating Query Languages and Systems for High-Energy Physics Data. Proc VLDB Endow. 2021 oct;15(2):154–168. Available from: `https://doi.org/10.14778/3489496.3489498`.

[15] collaboration C. SingleMu primary dataset in AOD format from Run of 2012 (/SingleMu/Run2012B-22Jan2013-v1/AOD); 2017. CERN Open Data Portal.

[16] Graur D, Müller I, Proffitt M, Fourny G, Watts GT, Alonso G. Benchmark Scripts for Evaluating Query Languages and Systems for High-Energy Physics Data. Zenodo; 2021.

[17] Szalay AS. The sloan digital sky survey and beyond. SIGMOD Record. 2008 jun;37(2).

[18] Rodrigues E. The Scikit-HEP Project. EPJ Web of Conferences. 2019;214.

[19] Pivarski J. Survey of data formats, conversion tools. In: HEP Analysis Ecosystem Workshop; 2013. Available from: `https://indico.cern.ch/event/613842/contributions/2585787/`.