Evaluating Query Languages and Systems for High-Energy Physics Data

Dan Graur
Department of Computer Science
ETH Zurich
dan.graur@inf.ethz.ch

Ingo Müller
Department of Computer Science
ETH Zurich
ingo.mueller@inf.ethz.ch

Mason Proffitt
Department of Physics
University of Washington
masonLp@uw.edu

Ghislain Fourny
Department of Computer Science
ETH Zurich
ghislain.fourny@inf.ethz.ch

Gordon T. Watts
Department of Physics
University of Washington
gwatts@uw.edu

Gustavo Alonso
Department of Computer Science
ETH Zurich
alonso@inf.ethz.ch

ABSTRACT

In the domain of high-energy physics (HEP), query languages in general and SQL in particular have found limited acceptance. This is surprising since HEP data analysis matches the SQL model well: the data is fully structured and queried using mostly standard operators. To gain insights on why this is the case, we perform a comprehensive analysis of six diverse, general-purpose data processing platforms using an HEP benchmark. The result of the evaluation is an interesting and rather complex picture of existing solutions: Their query languages vary greatly in how natural and concise HEP query patterns can be expressed. Furthermore, most of them are also between one and two orders of magnitude slower than the domain-specific system used by particle physicists today. These observations suggest that, while database systems and their query languages are *in principle* viable tools for HEP, significant work remains to make them relevant to HEP researchers.

PVLDB Reference Format:
Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T. Watts, and Gustavo Alonso. Evaluating Query Languages and Systems for High-Energy Physics Data. PVLDB, 15(2): 154 - 168, 2022.
doi:10.14778/3489496.3489498

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://doi.org/10.5281/zenodo.5569049.

1 INTRODUCTION

In the domain of High-Energy Physics (HEP), the well-known advantages of data processing platforms (data independence, declarative language, etc.) have not led to their adoption. This is surprising given the nature of the data and queries typical for that domain: Data sets in HEP are large but always fully structured. However, they are also heavily nested: they represent "events" registered by the sensors of a particle collider, where each event consists of a few scalar attributes as well as of numerous variable-sized sequences of relatively wide records. Queries follow a relatively simple pattern: they typically consist of a single scan over the input involving only a small subset of the available attributes, derivation of additional measures (potentially by joining and reducing the sequences *within the same event*), and selection of an interesting subset of events, which are then summarized using a reduction. HEP data is thus stored and analyzed in non-first normal form (NF^2)—a feature that early database systems did not support and thus the main reason why relational engines were rejected by physicists historically (along with the lack of support for used-defined code [40]).

Nowadays, most particle physicists work with a domain-specific system called the ROOT framework [4, 12], and increasingly so with its new RDataFrame interface [28]. In ROOT, queries are written in C++, requiring a non-trivial user effort, which can deter less experienced users [30]. Queries in ROOT entangle many aspects of the storage format and file system, the in-memory runtime format, the execution strategy, the target platform, and even the visualization. This makes many of the proven techniques known from data management difficult or impossible to apply. As one consequence, since ROOT does not support *distributed* processing out of the box, there is no standard scale-out solution for HEP analyses and various groups of physicists have built their own solutions for splitting up jobs into tasks, scheduling them across clusters, and combining the results. At the same time, many relational systems today offer rather complete support for nested data types like variable-size arrays, suggesting that the question of whether these general-purpose data processing systems are suitable or not for HEP should be revisited.

In this paper, we perform a comprehensive analysis in terms of expressiveness and performance of six general-purpose data processing systems that are *in principle* suited for HEP analyses: Postgres [68], as a representative of conventional database systems, Presto [64], a representative of systems for distributed data analytics with support for nested data, Google BigQuery [60] and Amazon Athena [62] as two Query-as-a-Service systems designed for large-scale analytics, as well as AsterixDB [2] and RumbleDB [49], two scale-out systems designed for document-oriented analytics. The baseline consists of the RDataFrames interface. We use the Analysis Description Languages (ADL) benchmark [58], created by physicists to evaluate languages and systems in their domain, and analyze typical query patterns occurring in the queries of the benchmark.

The result is a complex and rather intriguing picture: The two document-oriented systems, AsterixDB and RumbleDB, allow for

Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489498

the most natural and succinct query implementations. Their languages often even seem more elegant than those using RDataFrames, and, more importantly, their declarative nature also avoids most of the drawbacks of the ROOT framework mentioned above. They thus have the potential to increase the productivity of physicists significantly. To a slightly lesser degree, the same is true for the SQL dialects of BigQuery and Postgres, which implement the relevant parts of the SQL standard related to composite types and arrays. The conclusion is, hence, that there is no reason anymore to exclude SQL as a language for HEP analyses *per se*—there are, however, significant weaknesses in some of its dialects: Those of Athena and Presto are the most limited in the study; however, unlike many other systems that we excluded from the analysis, they *are* expressive enough to implement the queries even if it requires some effort. In terms of performance, the ranking is roughly reversed: AsterixDB, RumbleDB, and Postgres are the slowest systems in the comparison, about an order of magnitude slower than the fastest general-purpose system, Presto, which is another order of magnitude slower than RDataFrames. While their languages are well suited, these systems are hence not efficient enough to be practical. BigQuery has generally the lowest running times but the cost of individual queries is often an order of magnitude higher than that of RDataFrames. Overall, our study reveals significant weaknesses in all analyzed data processing systems in what could be considered their core competence and explains why physicists have resorted to developing their own solution.

2 HIGH-ENERGY PHYSICS DATA ANALYSIS

2.1 High-energy Physics

High-energy physics (or particle physics) studies the nature of the particles that constitute matter and radiation by observing the collisions of such particles in accelerators such as the Large Hadron Collider (LHC) run by CERN.

The nature of the particles and the need for high statistical confidence imply that a large number of similar collisions have to be analyzed. This makes the HEP community no stranger to large-scale data analytics. For example, the LHC in its latest configuration (Run 2), makes two particle beams intersect at a rate of 40 MHz, thus producing 40 million so-called "events" per second [36]. In each event, sensors register the presence and paths of particles resulting from the collision. The events are filtered on-site by a cascade of automatic filters until eventually about 1000 events per second are archived and shared [18] and subsequently analyzed by particle physicists around the world—the process we focus on in this paper. The amount of data archived is staggering: Run 2 produced data at a rate of about 8 GB/s, which amounted to 88 PB in 2018, and the next configurations, Run 3 and Run 4, scheduled for 2021 and 2027, will produce 2× and 10× that amount [14, 43, 76].

The information describing each event is shown in Listing 1. It consists of per-event metadata such as the run ID and the event ID, per-event measurements such as the missing energy (`MET`), and information about various types of observed particles.[1] Which particle types may occur depends on the underlying collision experiment as well as prior processing of the data set (called "reconstruction") and include jets, electrons, muons, tau particles, and photons. All

[1]For simplicity we denote particle-like objects such as jets simply as "particles."

```
struct MET  { float pt, phi, sumet /*...*/; };
struct Muon { float pt, eta, phi, mass /*...*/;
              int charge; };
struct Electron { /*...*/ };
// ...
struct Event { int event, run; MET met; // ...
    vector<Muon> muons;
    vector<Electron> electrons;
    /*...*/ };
```

Listing 1: Simplified schema of typical HEP data sets.

particle types have a common set of dimensions including transverse momentum (`pt`), pseudorapidity (`eta`), azimuth angle (`phi`), and mass, but some particle types have additional dimensions. For instance, electrons have a charge while jets do not. In each event, zero, one, or more particles may be observed of each type.

High-energy physics data is thus fully structured, contains no `NULL` values, and could be stored in normalized form in any RDBMS using one table per particle type with foreign keys to an event table. However, there is a strong ownership relationship between an event and its particles, and the particles are not analyzed outside of the context of their event. Physicists thus store HEP data *always* in non-first normal form (NF^2), i.e., particles are stored nested as part of each event. This makes it possible to store events in files without ever breaking foreign key constraints and eliminates both the mental effort and the execution cost of joins.

Data sets usually contain a large number of attributes (all available dimensions of all potentially present particle types, both measured or derived), at least several dozen and sometimes in the thousands. However, each query typically only accesses a few of them depending on what aspect the physicist is currently interested in. The data formats used for HEP data are thus all columnar formats in order to allow for pushing projections into the storage layer. For better or for worse, this columnar representation is also exposed by most programming abstractions commonly used for HEP analyses, which we discuss in more detail below.

In this paper, we concentrate on the final processing step ("analysis"). Virtually all queries in this step consist of two phases: (1) a sequence of transformations and filters applied *to each event in isolation*, and (2) one or several aggregations of the remaining events. The first phase consists of tasks such as establishing the presence of a particle type that sensors cannot detect directly but that can be derived from the presence of other particles or selecting events that contain a particular combination of interesting particles. The second phase consists of summarizing the selected events in form of a histogram of one or several dimensions of these events.

The analysis starts by determining some basic properties of the data set to confirm whether the data set is a good candidate for further exploration. Then, increasingly complex patterns are tried out, each of which is plotted in increasing detail to steer the subsequent search, until the result (i.e., the query itself as well as the plots) is eventually shared as code or as a materialized view.

2.2 The ADL Benchmark

The *Institute for Research and Innovation in Software for High Energy Physics* (IRIS-HEP) [32] has recently published the Analysis

Description Languages (ADL) benchmark [58], a sequence of high-level query descriptions designed to represent typical patterns in HEP data analysis. Its goal is to facilitate the test and comparison of languages and systems, and thus to guide the design of next-generation tools in the HEP domain. We use this benchmark in v0.1 as a running example for the remainder of the paper.

The data set of the benchmark consists of data obtained as a result of the Compact Muon Solenoid (CMS) experiment run on the Large Hadron Collider in the year 2012 [17]. It consists of roughly 54 million events (i.e., rows) and a total of 65 attributes (i.e., columns), totaling approximately 17 GB in the ROOT format.

The benchmark consists of the following eight queries. We make slight modifications to the query text in order to disambiguate some physics terminology:

(Q1) Plot the E_T^{miss} (missing transverse energy) of all events.
(Q2) Plot the p_T (transverse momentum) of all jets in all events.
(Q3) Plot the p_T of jets with $|\eta| < 1$ (jet pseudorapidity).
(Q4) Plot the E_T^{miss} of the events that have at least two jets with $p_T > 40$ GeV (gigaelectronvolt).
(Q5) Plot the E_T^{miss} of events that have an opposite-charge muon pair with an invariant mass between 60 GeV and 120 GeV.
(Q6) For events with at least three jets, plot the p_T of the trijet system four-momentum (i.e., any combination of three distinct jets within the same event) that has the invariant mass closest to 172.5 GeV in each event and plot the maximum b-tagging discriminant value among the jets in this trijet.
(Q7) Plot the scalar sum in each event of the p_T of the jets with $p_T > 30$ GeV that are not within 0.4 in ΔR of any light lepton (i.e., electron or muon) with $p_T > 10$ GeV.
(Q8) For events with at least three light leptons and a same-flavor opposite-charge light lepton pair, find such a pair that has the invariant mass closest to 91.2 GeV in each event and plot the transverse mass of the system, consisting of the missing transverse momentum and the highest-p_T light lepton not in this pair.

Note that "to plot" is short for "to plot an appropriate equi-width histogram," where 100 is a typical number of bins, the highest and lowest bins are typically set statically based on domain knowledge about the plotted metric, and under- and over-flows have their dedicated bins. Also, note that (Q6) consists of two different plots, i.e., a common sequence of transformations and filters is consumed by two distinct aggregations, as discussed above. We refer to the two plots as (Q6a) and (Q6b), respectively. Finally, note that some properties such as the mass of a trijet are calculated using involved mathematical formulae, which we do not include for conciseness. We refer to our query implementations [26] for details.

3 ANALYSIS OF HEP QUERY PATTERNS

We start with an analysis of functional requirements for data analytics in HEP with a focus on the query language. We interleave this analysis with a survey of existing general-purpose data processing systems that are, at least in principle, suitable for the HEP domain.

3.1 Methodology

Classification of requirements. We define the following classes of functional requirements:

(R1) Essential functionality, without which HEP queries cannot be reasonably expressed.
(R2) Important functionality, which has a major impact on readability and conciseness but acceptable alternatives exist.
(R3) Useful functionality helping to improve code quality but with limited impact.

In the analysis below, we label each requirement with (Ri.j), where i denotes one of the classes and j identifies the requirement. It is difficult to make such a classification fully objective as it depends on the user's taste, past experience, etc. However, it does help to interpret and summarize the findings.

Systems. We implement the queries in several SQL dialects as well as the documnet-oriented languages JSONiq and SQL++, and contrast them with state-of-the-art approach in the HEP domain, the*RDataFrames* interface of the ROOT framework, of which we use v6.24.02 [13]. For SQL, we consider the dialects of *PrestoDB* [64] v0.258 (or *Presto* for short), a modern system for large-scale analytics, *PostgreSQL* [68] version 13 (or *Postgres* for short), as a full-featured representative of conventional database systems, *Amazon Athena* [62] (engine version 2), a Query-as-a-Service system based on PrestoDB, and *Google BigQuery* [60], a Query-as-a-Service system built as the public version of Dremel [44]. We use PrestoDB rather than its fork *Trino* (previously called Presto SQL) because the latter does not support SQL-based user-defined functions. Presto does not make any claims of complying with the SQL standard, Postgres implements most features of SQL:2003 but does not aim at exact conformance, Athena uses "standard SQL" according to the product description, and BigQuery uses SQL:2011 with a few extensions.

We use JSONiq and SQL++ because they are designed to deal with the nested (and heterogeneous) JSON data model, which captures all aspects of HEP data naturally. We use our own implementation of JSONiq, *RumbleDB* [49] v1.11.0, which is built atop Apache Spark; however, the JSONiq queries run without modification on the two independent JSONiq implementations Zorba [77] and Xidel [73] as well. Both these systems are single-threaded and hence not optimal for HEP applications. SQL++ is the main query language of AsterixDB [2], a large-scale document store. We use the development version of AsterixDB with git hash `81c32493`, which is about two months older than the released version 0.9.7 and includes the fix to a performance bug we found during this study.

We also considered a number of additional systems; however, many of them lack the support of even the most basic features, so we excluded them from the comparison. For example, MySQL [53] and hence its hosted versions Amazon Aurora [75] and MySQL HeatWave [54], MS SQL Server [47] and hence Synapse Analytics [48] (which uses the same SQL dialect), and Actian Vector [1] do not have support for arrays and can thus not even represent the input data (unless normalized, which we exclude for the reasons given above). SAP HANA [23], and via their JSON type also MonetDB [10] and Snowflake [20], do have an array type but no suitable construct for unnesting or otherwise querying their elements. We also considered Spark SQL [5] but, in the end, decided to leave it as future work: it does support arrays and structs, but can only query the former through a set of array functions, against which we present arguments below. In contrast, we believe that the SQL dialect of *Apache Drill* [31] as well as the query language

```
df.Define("Jet_p4", make_p4, {"Jet_pt", "Jet_eta",
                              "Jet_phi", "Jet_mass"})
```
(a) RDataFrames.

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

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

```
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};
SELECT (SELECT a.x + b.x AS x, 42.0 AS y);
```
(f) SQL++.

Listing 2: Accessing and creating nested structs.

PartiQL [70] (based on SQL++, designed by AWS, and available for *Amazon Redshift* [63]) are suitable candidates for HEP analysis. Due to time and space constraints, we leave them as future work as well.

Data Format. Since none of the general-purpose systems supports reading from ROOT files directly, we convert the data to Parquet, which can represent ROOT files accurately and achieves a similar compression ratio. In this process, we also convert the data to a more natural representation: While the original ROOT files decompose the fields of structured attributes into distinct columns, both physically *and* logically, we represent them as logical `structs` and arrays thereof, which are both physically decomposed into scalar columns by Parquet. Instead of having to re-compose particle arrays from various attributes such as `Jet_pt`, `Jet_eta`, etc. and `nJet`, queries thus simply access the `Jets` attribute of type `array<struct<float pt, float eta, …>>`. Thanks to the separation of logical and physical representation of the data, the data is exposed in a more intuitive way to the user while maintaining the performance benefits of column decomposition.

3.2 Accessing and Creating Nested Structs

We now analyze basic language constructs for manipulating nested structs (or "objects"), starting with the availability of (R2.1) structured data types. This is a useful feature for handling several related attributes jointly, such as the various dimensions describing individual particles, and is thus ubiquitous in HEP queries. RDataFrames have no explicit support for structs but, since they allow arbitrary C++ types, users can create structured types manually or use existing types as they see fit. A typical example consists of assembling the library type for 4-dimensional space-time vectors (called "Lorentz-vectors") from their constituent atomic values (Listing 2a). The function `make_p4` is written by the user in C++ for the purpose of this query and applies the construction of Lorentz-vectors to arrays. Input data from ROOT files, however, is exposed in a columnar data model and not in the form of structs. Queries thus typically contain many columnar operations, where the correspondence between the dimensions of individual particles

```
df.Define("goodJet_pt", "Jet_pt[abs(Jet_eta) < 1]")
```
(a) RDataFrames.

```
SELECT j.pt FROM events
CROSS JOIN UNNEST(Jets) AS j
WHERE j.eta < 1
```
(b) BigQuery/Presto/Athena/Postgres.

```
$events.jets[][$$.eta < 1].pt
```
(c) JSONiq.

```
SELECT VALUE j.pt
FROM events AS e, e.Jets AS j
WHERE ABS(j.eta) < 1
```
(d) SQL++.

Listing 3: Simple unnesting of array elements.

is given only implicitly by matching indices of different columns. How this looks like becomes clearer with the examples below.

In contrast, the SQL standard describes the `ROW` type, as well as user-defined (composite) types, which were introduced in SQL:1999 for this purpose. BigQuery implements the `ROW` type under the type name `STRUCT`. As the first expression in Listing 2b shows, an instance can be constructed by (R3.1) defining a struct type inline. An additional syntax shown in the second expression allows constructing a struct where the types and optional names are defined by the expressions of the field values. If no name is provided for a particular field, that field cannot be accessed; however, since the whole struct can be coerced into another struct type with compatible field types (such as in a function call as we discuss in more detail below), such (R3.2) "anonymous" structs are still useful and very concise. Presto and Athena also implement the `ROW` type. Anonymous rows can be coerced into named ones as well; however, the only mechanism to create named rows is with a **`CAST`** expression as shown in Listing 2c, which is more verbose than BigQuery's inline declaration. In Presto, fields of anonymous rows can be accessed with their ordinal index; in Athena, fields of anonymous rows cannot be accessed at all. Postgres supports anonymous structs with the `ROW` type; however, only a subset of the expressions and functions on arrays can deal with them. For example, the **`ARRAY_AGG`** function accepts anonymous structs as input while the **`ARRAY_CAT`** function does not. It is thus often necessary to cast anonymous rows to (previously created) (R3.3) user-defined types as shown in Listing 2d—a possibity that otherwise only SQL++ provides.

In JSONiq and SQL++, objects can be created with the `{...}` operator containing any number of pairs of unique field names and values. In SQL++, the rows produced by a nested **`SELECT`** statement are of the same object types, which thus offers an additional way to create (collections of) objects. No anonymous objects exist; in SQL++, columns without aliases are given generic names.

In all considered query languages, fields can be accessed with the `.` operator known from many other programming languages, with the particularity in Postgres that "you often have to use parentheses to keep from confusing the parser." [71]

3.3 Accessing and Creating Nested Arrays

Dealing with nested arrays is more involved and can have various degrees of complexity. We use Query (Q3) as an example: it

```
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 using `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 4: Querying unnested array elements.

filters the elements of a nested array (of structs), flattens them, and projects to one of the fields of the resulting structs. In RDataFrames, we can assemble this query from a large library of vectorized operations that work on nested arrays directly as shown in Listing 3a. These operations include mathematical functions like `abs()`, Boolean operations like `<`, and array selections based on bit-vectors like `[.]`, as well as many others. Unnesting for the purpose of aggregating across events is done implicitly.

In order to access array elements in SQL, `UNNEST(.)` was introduced in SQL:1999. Listing 3b shows how (R1.1) `UNNEST(.)` is combined with `CROSS JOIN`, which produces one row for each element of the provided array attribute where all other attributes are duplicated. The remainder of the query can thus filter and project on any of the attributes as usual. Note that this behavior is not strictly functional and thus somewhat hard to understand: `UNNEST(.)` on one attribute affects the number of occurrences of the other attributes. SQL++ also supports the `UNNEST(.)` construct, though without the `CROSS JOIN` keyword, and offers a short version with implicit unnesting as shown in Listing 3d.

In JSONiq, such a simple query can use concise operators for dealing with arrays, objects, and sequences thereof (all of which are strictly functional): The initial `.jets` extracts the `jets` member of each object contained in the input and produces a flat sequence of arrays; the subsequent `[]` operator extracts elements of each of these arrays and produces again a flat sequence of objects; the predicate expression `[...]` then applies a filter (using the context item `$$`); and the final `.pt` extracts a flat sequence of numbers.

However, most real-world HEP queries are far more complex, and the `CROSS JOIN` approach in SQL does hence not seem to be a great fit. Consider the only slightly more complex (Q4), which filters events having at least two jets matching some predicate. As Listing 4c shows, we can implement this query with a `CROSS JOIN`, which unnests the jets, filters them with the predicate, and then counts the matching jets per event. However, since the jets of all events are now flattened, the latter operation requires a `GROUP BY`, which essentially undoes the flattening again. Apart from potentially leading to a sub-optimal query plan,[2] this makes both reading and writing the query less natural than necessary. Furthermore, this only works for a single array—a query that should filter or transform the elements of two or more arrays has to be expressed as the join of two sub-queries or as the sequence of two common table expressions, which we discuss in more detail below. Listing 4d shows an alternative formulation based on the (non-standard) (R3.4) array functions `CARDINALITY` and `FILTER`. While this overcomes the aforementioned problems, it does not work for the more complex patterns we discuss below.

Consider Listing 4b as a contrast, which shows the alternative of using a (R2.2) nested subquery, which is part of SQL:1999. In terms of semantics, the subquery can be thought of to run once per row of the outer query, providing the outer row as a constant. Other than that, the two levels of the nesting do not influence each other; in particular, the subquery does not change the number of rows of the outer query (unlike a `CROSS JOIN`). Also, the subquery uses the same language constructs as the top-level query: `UNNEST(events.Jets)` produces a table expression that the remainder of the query uses as if it were a base table. While BigQuery, Postgres, and (with a slightly different syntax) AsterixDB support this construct, Athena and Presto do not. Since in SQL++ tables and arrays are both collections, which can be used as an input for array functions, SQL++ offers some more freedom to combine the two paradigms as shown in Listing 4f.

JSONiq has similar constructs illustrated in Listing 4e: The outer level uses a "FLWOR" expression, roughly speaking a generalization of SQL's SELECT-FROM-WHERE that has an imperative look-and-feel but is, in fact, declarative. The `for` clause produces a stream of tuples from the items in the input sequence `$events`, in each of which the item is "bound" to the given variable name. Subsequent clauses modify this stream; for example, the `where` clause filters out tuples from the stream. There is only one type of expression in JSONiq, i.e., the result of any expression can potentially be used as an input of any other expression. In the example, this means we can use similar expressions for unnesting and filtering of sequences as before in order to express the predicate of the `where` clause.

With RDataFrames, more complex query logic that cannot be assembled with the vectorized operations provided by the framework has to be written as UDFs in C++. The `make_p4` function above is such an example. Since the main query logic in the form of data frame operations is written in C++ as well and the framework runs both levels in the same (optimizing) C++ interpreter, these UDFs are reasonably seamless and efficient. However, as we illustrate in the remainder of this section, large fractions of the query logic end up being written in UDFs outside the RDataFrame API.

An interesting variation of this pattern is to (R3.5) produce a new array of particles in each event, typically derived from the existing

[2] *If* `event_id` is unique in the input, the unnesting produces runs of rows belonging to the same group, so a single-pass aggregation can be used. However, if the input is an external table made from files in Parquet or ROOT format, that information is not available.

```
ARRAY(SELECT AS STRUCT ...) AS new_particle
```
(a) BigQuery.

```
ARRAY(SELECT ...) AS new_particle
```
(b) Postgres.

```
SELECT event_id, ARRAY_AGG(...) AS new_particle
... GROUP BY event_id
```
(c) Presto/Athena.

```
[for $event in $events ...]
```
(d) JSONiq.

```
SELECT (SELECT ... FROM ...) AS new_particle
```
(e) SQL++.

Listing 5: Creating arrays.

particles in that event. While none of the queries in the benchmark explicitly ask to do that, we have found it to be extremely useful for debugging as well as for assembling complex chains of transformation and filter stages (see below). Listing 5 shows this pattern in the query languages: In BigQuery, Postgres, and SQL++, arrays can be constructed from any subquery (with minor syntactic differences), whereas Presto and Athena require the `UNNEST`/`GROUP BY` pattern using `ARRAY_AGG`. In JSONiq, the items returned by any expression can be turned into an array using the `[]` expression. With RDataFrames, UDFs can create arrays by returning an instance of the generic `ROOT::RVec` type.

3.4 Particle Combinations

While the patterns above deal with individual particles (or a certain number thereof), most real-world HEP queries actually involve *combinations* of particles. Combinations may be (R1.2) *asymmetric* (such as in "any electron-muon pair") or (R1.3) *symmetric* (such as in "any three jets") and may consist of combinations of two or more particles. Since only particle combinations within the same event are interesting, it is natural to think of one event in isolation. With that perspective, asymmetric combinations are simple Cartesian products, while symmetric combinations are only a "diagonal half" of such products, i.e., exactly one of (p_i, p_j) and (p_j, p_i) is in the result and (p_i, p_i) is not.

We illustrate how the various programming interfaces express this pattern in Listing 6. The RDataFrames API provides the vectorized `Combinations` operation for that purpose (Listing 6a), which produces indices of the desired combinations into the input array, which can in turn be used to access the actual particles. For use inside UDFs, the ROOT framework provides the semantically equivalent `VecOps::Combinations` function.

For the query languages, the patterns are similar as before: SQL subqueries can simply use several calls to `UNNEST` in their `FROM` clause, which produces the Cartesian product of their content as usual. In bot hcases, combining unnested particles also works outside of sub-queries, however, with the same drawbacks as discussed above. Duplicates can be eliminated with a filter on the indices, which are produced by the `WITH OFFSET` clause in BigQuery as shown in Listing 6b and by the `WITH ORDINALITY` clause in Postgres (Listing 6c). SQL++'s equivalent, the `AT` clause shown in Listing 6f, is even more concise, though currently not officially supported and thus undocumented. We still use it in our study because it produces

```
df.Define("indices", "Combinations(Jet_p4, 2)")
```
(a) RDataFrames.

```
... FROM UNNEST(events.Jets) j1 WITH OFFSET i,
         UNNEST(events.Jets) j2 WITH OFFSET j
    WHERE i < j ...
```
(b) BigQuery.

```
... FROM UNNEST(Jets) WITH ORDINALITY AS j1,
         UNNEST(Jets) WITH ORDINALITY AS j2
    WHERE j1.ordinality < j2.ordinality ...
```
(c) Postgres.

```
SELECT j1, j2, ... FROM events
CROSS JOIN
    UNNEST(Jets) WITH ORDINALITY AS j1(..., idx)
CROSS JOIN
    UNNEST(Jets) WITH ORDINALITY AS j2(..., idx)
WHERE j1.idx < j2.idx ...
```
(d) Presto.

```
... (for $jet1 at $i in $event.jets[]
     for $jet2 at $j in $event.jets[]
     where $i < $j ...) ...
```
(e) JSONiq.

```
... FROM e.Muon AS m1 AT idx1,
         e.Muon AS m2 AT idx2
WHERE idx1 < idx2 ...
```
(f) SQL++.

Listing 6: Creating combinations of two jets.

correct results in the queries of the benchmark and is expected to be completed soon [38]. Presto follows the syntax from the standard, which enumerates the full list of fields names for specifying an alias. Athena, BigQuery, and Postgres, in contrast, allow to give an (R3.6) alias for the struct *as a whole* reducing the verbosity significantly. Postsgres supports both versions, SQL++ only the short one (but its name resolution scheme allows to access the struct's fields without specifying the alias of the struct in some situations).

In Presto, the alternative of using array functions extends to symmetric combinations thanks to its array function `COMBINATIONS`; however, it is non-standard, does not (easily) work for asymmetric combinations, and is not implemented by any other system we are aware of, so we do not discuss this alternative further. Interestingly, this function (like a few other array functions) is not supported by Athena even though it originates from the same code base.

In JSONiq, the FLWOR expression may contain several `for` clauses, which essentially produce the Cartesian product of their input sequences like in SQL. With `at $i`, indices can be produced and, of course, expressions can be nested arbitrarily as before. In all query languages, creating arrays is done as before (via `ARRAY(.)`, `ARRAY_AGG`, and the `[...]` operator, respectively).

3.5 Multiple Transformations and Filters

Most HEP queries do not consist of a single transformation or filter using the patterns above, but a *series* of them: physicists typically first compute some basic properties of each event, which they may use in an initial filter, and then iteratively add more of them to refine their search, often reusing properties of the previous steps.

```
WITH Leptons AS (...),
TriLeptonsWithOtherLepton AS (
    SELECT *, (...) AS BestTriL
    FROM Leptons AS l
    WHERE ARRAY_LENGTH(l.Leptons) >= 3),
TriLeptonsWithMassAndOtherLepton AS (
    SELECT *, TrMass(MET, BestTriL.other) AS trMass
    FROM TriLeptonsWithOtherLepton
    WHERE BestTriLepton IS NOT NULL)
...
```

(a) BigQuery.

```
for $event in parquet-file($input-path)
let $leptons := hep:concat-leptons($event)
let $best-tri-lepton := (...)
where exists($best-tri-lepton)
let $other := (...)
let $trMass := hep:TrMass($event.MET, $other)
...
```

(b) JSONiq.

Listing 7: Sequence of transformations in (Q8) (simplified).

With RDataFrames, the user can specify a sequence (or even a tree, if they specify more than one sink) of transformations and filters, which can be chained arbitrarily.

In contrast, due to its lack of (R2.3) "variables," the same task is somewhat cumbersome in SQL—it is not possible to define a column alias with `AS` and use that alias for the computation of other columns. For example, (Q6) and (Q8) combine several particles into one pseudo-particle, which consists of a vector space transformation, a piece-wise addition, and a reverse vector space transformation. If we want to use this pseudo-particle more than once, for example, to filter on one property and plot another one, we need to spell out the computation of the pseudo-particle *repeatedly*.

In the SQL implementations of the benchmark, we found that a sequence of common table expressions or CTEs (i.e., `WITH` statements) resulted in the most concise and readable code. Listing 7a shows an example: each CTE adds one or more attributes such as `BestTriL` and `trMass` and passes through all existing ones (with `*`) such that the subsequent CTEs or the final `SELECT` statement can use them. The other SQL dialects work the same way. While this is reasonably concise and avoids repeated code, it involves the *outer* level of relations even though the computations only concern the *inner* level of events. Furthermore, the necessary reference to the previous CTE(s) in the `WITH` clause is more verbose than necessary.

A special case of lack of variables in SQL is the fact that the standard does not allow to (R2.4) use a column alias in the `GROUP BY` clause. Since *all* queries consist of computing histograms, i.e., counting occurrences of values *per bin*, this is a frequent pattern. In the dialects where this is the case, we thus use at least one CTE to derive the desired property and a final `SELECT` statement just for the histogram (otherwise, the computation of the property would have to be repeated in the `GROUP BY` clause). Since BigQuery and Postgres *do* allow one to use aliases in the situation at hand, they allow for more concise queries than the other SQL systems (but diverge from the standard in this respect).

In JSONiq, variables are an integral part of the pseudo-imperative programming model of its FLWOR expression. Since in that expression, `for`, `let`, `where`, `return`, and a few other clauses can be chained in essentially arbitrary order, it is possible to assemble sequences of transformations and filters in a single, top-level FLOWR expression (which potentially uses nested FLOWR expressions on nested data). Listing 7b shows the same example as earlier. Each `let` clause introduces a new variable that any of the subsequent clauses and nested expressions can use—a concept that not only results in more concise code but is also familiar to programmers of virtually any background. SQL++ also has a `LET` clause that essentially solves the problems of SQL discussed above; however, it can be used somewhat less freely than that of JSONiq.

3.6 User-defined Functions

In all of the above, we give examples of query logic that may be reused both within the same query and across them, so a means to (R1.4) encapsulate query logic in user-defined functions (UDFs) or similar is essential. An example for the first type of reuse is (Q7), which asks for a "lepton" with a particular property, i.e., either an electron or muon with that property (these two particle types both being leptons). Since electrons and muons of an event are stored in two different columns, we either need to repeat the computation of the desired property or encapsulate it as a (temporary) UDF. With RDataFrames, the user can write UDFs in C++ for that purpose as described above. The second type of reuse is more common and much more important: the ROOT framework and similar tools for HEP analyses come with a large collection of what could be called "business logic," which encapsulate the computations of physical properties such as a certain derived property of a particle, the combination of several particles into one pseudo-particle, etc. These computations often include mathematical formulae and spelling them out for every query would be tedious and error-prone.

The support for UDFs in the SQL-based systems in our study is mixed. Athena does not support any type of UDFs suitable for our use case. The offered UDFs are based on serverless functions and thus need to ship all data to a different cloud service, invoke the UDF for each record at the time, incur further costs, and are subject to concurrency quotas—in short, they are not suitable for data-intensive tasks. Presto recently added experimental support for UDFs—currently with the severe limitation that UDFs cannot call other UDFs, making it impractical to implement real-world function libraries. However, given the current development effort on this feature, this limitation is likely to be lifted soon. BigQuery, Postgres, and SQL++ have mature support for both permanent and temporary UDFs. JSONiq allows declaring functions as part of the query text and to import functions and constants from external modules. Since the full name of such a module is a URI that typically hosts the code of that module publicly on the web, this mechanism can be seen as a simple built-in package manager.

UDFs in both Presto and BigQuery (R2.5) support structs as function parameters; however, the field names and types of the structs need to be specified in the declaration, and the value used in the invocation needs to have matching arity and compatible types.[3] This allows for providing anonymous structs as discussed above; however, the invoking code needs to (1) project away all fields of an existing `ROW` instance that the function does not list in

[3] In particular, it is not possible to access fields of a function argument declared as `ANY TYPE` in BigQuery.

```
CREATE TEMP FUNCTION AddPtEtaPhiM2(
   pepm1 STRUCT<Pt FLOAT64, Eta FLOAT64,
                Phi FLOAT64, Mass FLOAT64>,
   pepm2 STRUCT<Pt FLOAT64, Eta FLOAT64,
                Phi FLOAT64, Mass FLOAT64>) AS ...
SELECT AddPtEtaPhiM2(
     STRUCT(l1.Pt, l1.Eta, l1.Phi, l1.Mass),
     STRUCT(l2.Pt, l2.Eta, l2.Phi, l2.Mass)), ...
FROM UNNEST(Leptons) l1, UNNEST(Leptons) l2 ...
```

(a) BigQuery.

```
CREATE FUNCTION AddPtEtaPhiM2(
    IN pepm1 anyelement, IN pepm2 anyelement) ...
SELECT AddPtEtaPhiM2(l1, l2)
FROM UNNEST(Leptons) l1, UNNEST(Leptons) l2 ...
```

(b) Postgres.

```
declare function hep:add-PtEtaPhiM($p1, $p2) { ... };
for $l1 at $i in $leptons
for $l2 at $j in $leptons
let $mass := hep:add-PtEtaPhiM($l1, $l2).mass
...
```

(c) JSONiq.

```
DECLARE FUNCTION AddPtEtaPhiM2(p1, p2) {...};
FROM Leptons AS l1 AT idx1, Leptons AS l2 AT idx2
SELECT AddPtEtaPhiM2(l1, l2).mass ...
```

(d) SQL++.

Listing 8: Declaration and call of a UDF in (Q8) (simplified).

its parameter and (2) bring the remaining ones in the same order. Listing 8a illustrates how this affects verbosity in BigQuery: The full list of field names need to be specified twice, once in function declaration and once for assembling the call value. Listings 8b, 8c, and 8d show the corresponding example in Postgres' dialect, JSONiq, and SQL++, respectively.[4] Both the function declaration and the invocation site use objects without explicit enumeration of member names—the members accessed by the function body must be present (otherwise, depending on the body, an error is raised) and the superfluous members at the invocation are simply ignored (and may be physically removed by an optimizer). Furthermore, the order of the members does not matter (as it is undefined).

The SQL-based systems have additional limitations. They do not support (R2.6) UDFs that consume or produce a table. It is thus not possible to encapsulate the computation of histograms fully. Instead, the user needs to spell out the grouping with aggregation manually for every query as illustrated in Listing 9b. In JSONiq and SQL++, functions can work on any level so this logic can be fully hidden as illustrated in Listings 9c and 9d. Also with the RDataFrames API, only the built-in transformations can be applied to data frames; however, a large number of such transformations and sinks exists, in particular, domain-specific ones such as the `Histo1D` sink shown in Listing 9a, which computes the aggregation required for the histogram (and immediately produces the actual plot). Furthermore, SQL dialects do not allow declaring variables inside of UDFs, requiring a series of helper functions to avoid repeating common

[4]Notice in the example that SQL++ allows to specify the `SELECT` as the *last* clause of the statement.

```
df.Histo1D({"histogram name", "title;x-label;y-label",
            100, 0, 2000}, "values");
```

(a) RDataFrames.

```
SELECT HistogramBin(value, 15, 250, 100) AS x,
       COUNT(*) AS y
FROM previousCTE GROUP BY x ORDER BY x
```

(b) BigQuery.

```
hep:histogram($values, 15, 250, 100)
```

(c) JSONiq.

```
histogram((FROM ... SELECT ...), 15, 250, 100)
```

(d) SQL++.

Listing 9: Histogram computation.

sub-expressions (see the CTEs discussed above). The UDFs in the SQL dialects studied are not standard and mutually incompatible.

3.7 Summary

We summarize the discussion of this section in Table 1. In addition to marking unsupported features with a dash (-), we also indicate "how well" a system supports the features it supports (more asterisks indicating better support). Again, this cannot be seen as a fully objective quantification but rather as an approximate visualization of the detailed discussion above. The table suggests that JSONiq and SQL++ are best suited for HEP analyses—since they were purpose-built for the JSON data model, which is also heavily nested (and heterogeneous), this is not completely surprising. BigQuery's and Postgres' SQL dialects implement all related features of the standard and have a few proprietary extensions, which together make them a good match as well; the only major missing constructs are the lack of variables and table-based UDFs. In contrast, we believe that Athena cannot currently be considered viable as the lack of UDFs makes it impossible to share library code between users. While Presto is on the brink of having this feature, it shares a number of missing or cumbersome constructs with its fork Athena that make it a less-than-ideal (though viable) system for HEP. RDataFrames are of course well suited for what they are built for; however, the fact that they make the columnar storage format part of the programming model requires a higher programming effort than that of a suitable declarative query language. Overall, 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.

We have implemented the full benchmark in the seven languages, dialects, and programming interfaces. Table 1 shows the overall implementation length in various metrics: the number of characters and lines (which exclude white space, blank lines, and comments), the number of clauses (to which we include calls to built-in functions), the number of unique clauses per query, the number of unique clauses (counting how many *different* language constructs are used overall), and the number of average unique clauses per query. All six metrics show the same picture: SQL++ and JSONiq are most concise; Athena, Postgres, BigQuery and Presto require more code. Such metrics always need to be interpreted with care: the metrics vary with implementation style, formatting, etc., more

Table 1: Summary of functionality of general-purpose data processing systems for HEP analyses.

	Athena	BigQuery	Postgres	Presto	JSONiq	SQL++	RDataFr.
(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

concise queries are not always more readable, and readability depends to a large degree on the personal preference and experience of the programmer. Still, the numbers do provide a useful quantification of the previous discussion of this section and suggest a similar conclusion. We have also made the full implementations publicly available [26], as a reference for the reader.

4 PERFORMANCE EVALUATION

We now study the efficiency and scalability of the systems implementing the query languages of the previous section.

4.1 Experimental Setup

Platform. For the self-managed systems, we use virtual machines in Amazon EC2 from the `m5d` series. The largest size of that instance type, `24xlarge`, has 48 real CPU cores, 384 GiB of main memory, four NVMe SSDs of 900 GiB each configured as RAID 0, and 25 Gbit/s networking, and costs 6.048 $/h in the `eu-wests-1` region; all numbers are proportionally smaller for the smaller sizes.

Storage. We use the original ROOT files for RDataFrames and Parquet files for all other systems. For AsterixDB, Athena, BigQuery, Presto, and RumbleDB, we place them on cloud storage and process directly from there. For RDataFrames, we put the input files on the local disk, which gives a similar performance as the typically used `xrootd` network storage protocol, which, in turn, is more performant than S3 for the required access patterns. For Postgres, we use the experimental foreign-data wrapper for Parquet files [50] and place the files on SSD since it does not support network storage.

Input Data Size. We use the original data set defined by the benchmark [17]. Additionally, we define a *scale factor* (SF), which we use for simulating more realistic data set sizes of up to 2 TB. Even though the original files are from experiments from 2012, they are the most up-to-date *publicly available* files due to the regular shut-down periods of the accelerator and data retention policies of CERN. Since then, the amount of produced data has grown exponentially [72], and data set sizes used by physicists today as well as their expected size for the next decade, are about one and two orders of magnitude larger [14, 43, 76], respectively. We thus replicate the original file SF times for SF ≥ 1 and take the first fraction of SF events for SF ≤ 1.

4.2 End-to-End Comparison

We start with a comparison among the systems under test on their end-to-end performance and monetary cost. For the self-managed systems, we report numbers for all instance sizes from `large` to `24xlarge`. We compute the query cost as the product of the number of wall-time seconds and the per-second price of the underlying instance. We use SF = 1 with the exception of RumbleDB, where we use the largest data set size that the system can handle in at most 10 min and extrapolate from that number. We report a single configuration for the QaaS systems since we do not have any control over the amount of resources.

Figure 1 shows the result for all queries except (Q6b), which has nearly identical results as (Q6a). On all queries, BigQuery is the fastest system, often even when using external tables, answering all queries except (Q6) in less than 10 s and many in a low single-digit number of seconds. We discuss the special nature of (Q6) below. Using pre-loaded data improves performance further by about 2× in most queries. Athena is significantly slower: it comes close to BigQuery in some queries (e.g., (Q3)), but often has a significant margin, though response time is still in the order of 10 s to 20 s except for (Q6). The performance of the self-managed systems generally depends on the instance size, yielding lower running time using large instances but also a higher cost. The only exception is Postgres: despite extensive manual rewriting of the queries and tuning of the configuration, the system was only able to fully parallelize (Q1)—for all other queries, the system could not produce fully parallel plans or did not execute the plans in parallel. RDataFrames is a strong runner up; its fastest configuration outperforms BigQuery with external tables for some queries and only has a narrow gap on the other ones. The good performance of RDataFrames can be explained by its efficient, jit-compiled execution model. Presto is again significantly slower, in particular, for small instance sizes, though its fastest configuration comes close to Athena. AsterixDB, Postgres, and RumbleDB, however, are about one order of magnitude slower than the next system and up to two orders of magnitude slower than the fastest. Except for the most simple ones, queries take minutes or even hours rather than seconds on these systems, making them impractical for interactive analyses. We attribute the slower performance of these systems to their interpreted execution model; in particular, AsterixDB and RumbleDB are designed to work on heterogeneous data sets, where polymorphic operators on polymorphic data representations are hard to avoid.

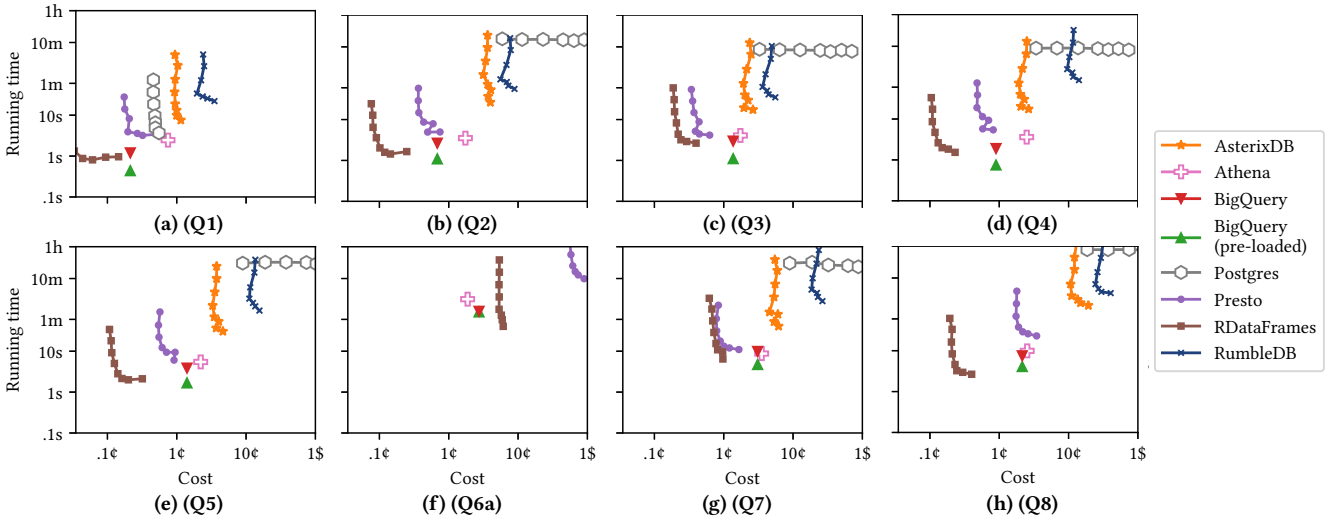


Figure 1: Running time/cost trade-off for various systems under test.

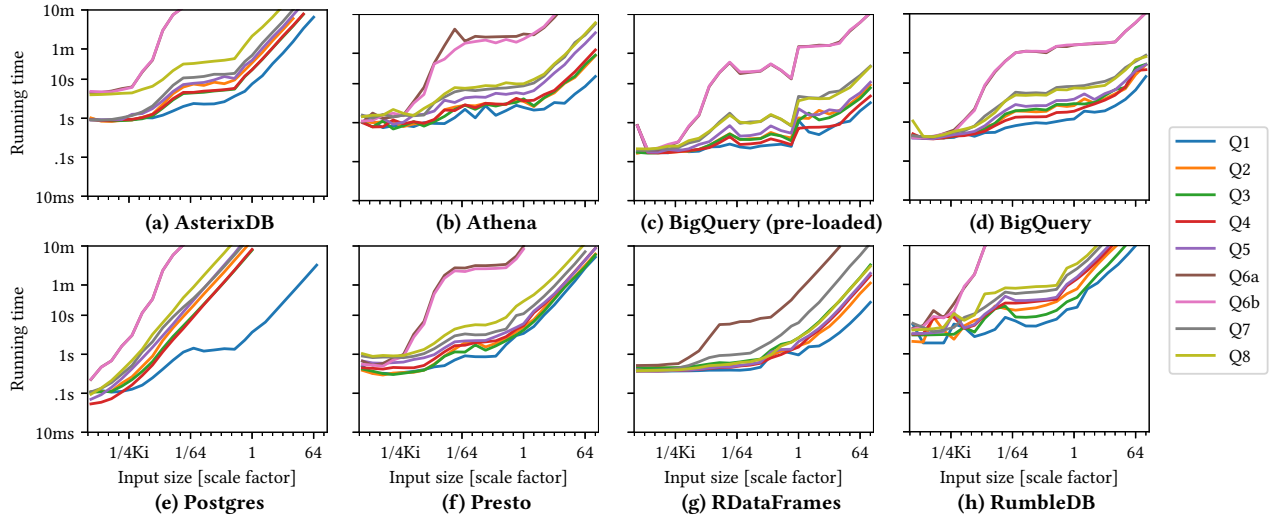


Figure 2: Impact of data size on end-to-end running time of various systems under test.

The monetary cost varies greatly among the systems as well. Among the two QaaS systems, BigQuery is not only faster but also cheaper for almost all queries. This is due to an interplay of various factors in the respective pricing models, which we discuss in more detail in the extended version of the paper [25].

The cost of the self-managed systems is linear on the queries' running time, leading to a different picture: For the computationally simple queries, namely (Q1) to (Q5), Presto and RDataFrames are significantly cheaper than the QaaS systems, often by a factor $2\times$ to $6\times$. However, the gap narrows for (Q7) and (Q8), which are more compute-intensive as indicated by the higher running times. For (Q6), Presto and RDataFrames are more than one and about half an order of magnitude more expensive than the QaaS systems, respectively. This is mainly due to the pricing model of the latter, in which computation is essentially free. AsterixDB, Postgres, and RumbleDB are more expensive by the same factor they are slower: at least an order of magnitude compared to the next system. The potentially higher productivity of their query language thus currently comes at a significantly higher monetary cost.

The numbers presented in this experiment show only a partial picture of the total monetary cost. First, using spot instances has the potential to reduce the cost considerably, sometimes by up to $5\times$. Second, the user also has to pay for the idle time of their instances, which in turn could be reduced with a multi-tenant cluster and/or auto-scaling. The presented numbers, however, give *some* indication about how self-managed systems compare to QaaS systems.

4.3 Scaling with Data Set Size

We study the impact of different data set sizes. We run the benchmark on SF $= 2^i$ for $i = -16, \ldots, 7$. At the smallest scale factor, the input consists of about 800 events and requires about 250 KiB; at the largest scale factor, it consists of about 6.8 G events and requires

Table 2: Query complexity.

Query	Complexity	#Ops/event
(Q1)	1	1.0
(Q2)	J	3.2
(Q3)	J	3.2
(Q4)	$1 + J$	4.2
(Q5)	$1 + \binom{M}{2}$	1.6
(Q6)	$1 + \binom{J}{3}$	42.8
(Q7)	$(E + M) \cdot \sigma(J)$	1.5
(Q8)	$E \cdot M + E + M + 1$	11.6

slightly more than two terabytes. For the self-managed systems, we use the largest size (`m5d.24xlarge`); cloud-based systems use the resources assigned by the cloud provider. We exclude all configurations that take longer than 10 min to complete.

Figure 2 shows the result. The running time initially increases in all cases with the data size and then reaches a plateau. This is due to the granularity of parallelization: All systems using Parquet files only parallelize *across* row groups, not *within* them. Each row group has an average of 400 k events, which is the beginning of the plateau—execution is single-threaded for smaller data sets and increases with the number of events, and parallelized after that. For Postgres, this is only true for (Q1) due to its inability to parallelize the other queries as discussed above. The running time then increases again at the end of the plateau. This happens when there are more row groups than CPU cores: at SF = 128, the data set has in the order of 16 k row groups in Parquet while the instances have 48 real CPU cores (i.e., 96 logical cores including SMT), so the largest data sizes cannot be processed completely in parallel. Interestingly, this is even true for the QaaS systems, where our queries seem to exceed the amount of resources that the cloud providers are willing to dedicate to a single query for scale factors larger than about 2 and 8 for Athena and BigQuery, respectively. At the largest scale factor, all systems seem to reach a steady state where the running time is dominated by the raw processing throughput in terms of events per second. Due to the structure of the queries (a top-level aggregation with a small number of groups), we expect this to remain true for scale factors > 128.

4.4 Compute Intensity

We now study the balance of compute and I/O in more detail. We start with a complexity analysis of the queries. Table 2 gives a formula for each query indicating how many records or record combinations the query must explore for each event. Note that this number is 1 for all scans not involving arrays and hence for the traditional use cases with data in NF1. E, J, and M denote the number of electrons, muons, and jets, respectively; σ denotes the filter used in (Q7). While (Q1) does not access any particle array and (Q2) to (Q4) only accesses one of them, the remaining queries produce particle *combinations* as discussed in Section 3.4. From the formulae, we can understand that the compute intensity of a query depends on the distribution of the number of particles *per event*. Figure 3 shows this distribution for the data set of the benchmark and the three particle types used by the queries. It shows that electrons generally occur in the low single-digit numbers, muons generally

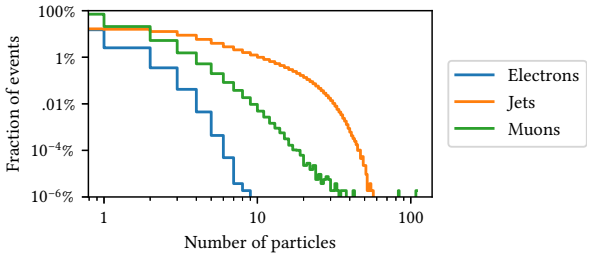
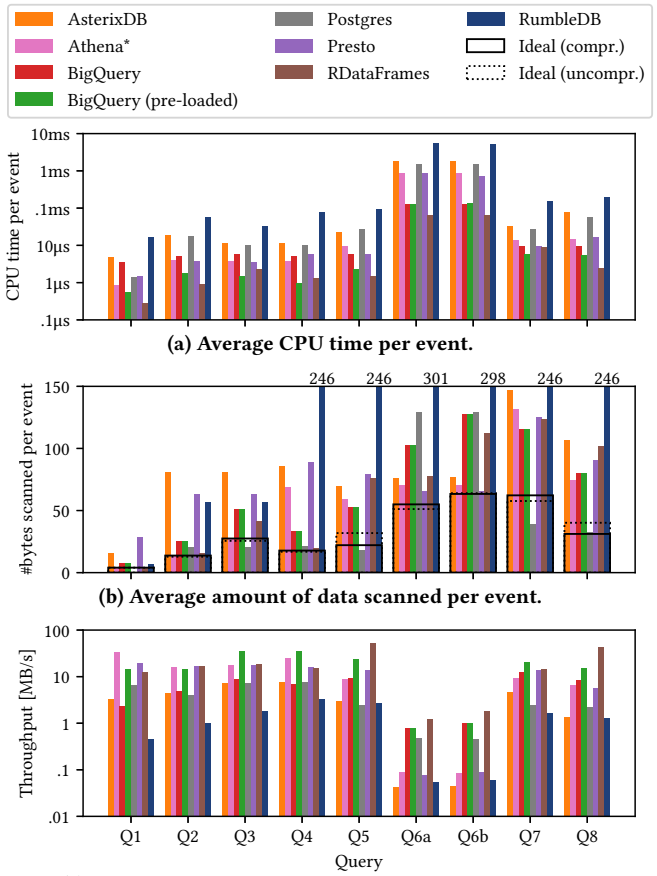


Figure 3: Distribution of number of particles per event.

occur more frequently and reach higher per-event occurrences, and a significant fraction of the events consists of several dozen jets. In these "large" events, exploring all combinations of three jets, like required by (Q6), may lead to a significant cost. For example, a single event with 50 jets has $\binom{50}{3} = 19\,600$ combinations of three jets, for each of which queries typically compute some distance using a sequence of expensive geometric functions. The last column of Table 2 shows how many records or record combinations the query must explore on average for each event in the benchmark data set. (Q6), in particular, and to some degree (Q8) are thus *intrinsically* compute-intensive and the running time of even highly tuned execution engines are likely to be dominated by these computations rather than by I/O.

To quantify the balance between I/O and compute further, we compare three metrics across the queries and systems in Figure 4. In Figure 4a, we show the average CPU time, which we compute as the total number of seconds any logical core spends on doing work for the query and which excludes the wait time of these cores. BigQuery, Presto, and RumbleDB report this metric in their statistics; for the remaining self-hosted systems, we use the CPU time as reported by the OS kernel. As a best effort, we run the queries on Athena against a version of the input that consists of a single row group such that the query has to be processed sequentially and then take the wall time as the CPU time; this approximation may not be accurate and has to be taken with a grain of salt (hence the * in the legend of the plot). The plot shows that the CPU time follows similar trends as the end-to-end running time reported above: the ranking among the systems is the same and (Q6), (Q8), (Q7), and (Q5) take the longest to compute. This is also in line with the computational complexity of the queries shown in Table 2.

In Figure 4b, we show the number of bytes scanned *per event*. Most systems report that metric as well; otherwise, we use statistics from the IO and networking subsystems of the OS. We also show the ideal value for that metric, once computed based on the column size as reported by the Parquet metadata, once based on the number of column entries times the size of the data type (4 B for most attributes). We see that all systems scan significantly more data than they should: BigQuery reports 2 × more than it actually reads from storage due to its pricing model. Also, AsterixDB, Athena, and Presto are not able to push down projections into structs; instead, they always read all fields of any struct attribute accessed. This is most likely due to a limitation of the Java implementation of the Parquet format, but it is not intrinsic to the format itself—the C++ implementation does not have this shortcoming. RumbleDB does not seem to push *any* projection into the scan and thus reads the full file for all but the simplest queries. RDataFrame also causes



(a) Average CPU time per event.

(b) Average amount of data scanned per event.

(c) End-to-end processing throughput per logical core.

Figure 4: Analysis of compute/IO balance.

more bytes to be read than expected even though the projections are manually specified by the user; further work is needed to understand why. We could not determine why Postgres seems to read *less* data than ideal in some queries; it is possible that it caches part of the input in its buffer pool.

Finally, Figure 4c shows the scan throughput per core, computed as the amount of data scanned divided by the total CPU time. This number reflects the balance of compute and I/O. Comparing it with the typical I/O bandwidth obtainable from storage, typically around 50 MB/s to 200 MB/s per core, it indicates whether the queries are bound by I/O or by compute. If we discount the numbers of BigQuery by the 2× inflation of its pricing model, none of the systems comes close to the raw storage bandwidth: BigQuery, Presto, and Athena achieve only 15 MB/s to 30 MB/s on (Q1) to (Q4); BigQuery more or less maintains these numbers also for (Q7) and (Q8), while the others fall below 10 MB/s. On (Q6), BigQuery and RDataFrames achieve a mere 1 MB/s, while AsterixDB, Athena, Presto, and RumbleDB drop to as little as 100 kB/s. This indicates that the systems are heavily compute-bound, which is to some degree intrinsic to the queries as shown by the complexity analysis above; however, as the difference in performance among the systems indicates, most systems are also significantly less efficient than possible, often by one or even several orders of magnitude.

5 RELATED WORK

Over the decades, many researchers have studied the possibility of using database systems for data-intensive scientific applications in general [65] and high-energy physics in particular [6, 8, 11, 15, 19, 21, 22, 24, 27, 33, 34, 37, 39, 41, 42, 51, 56, 66, 74]. The domain where these efforts have been most successful is arguably that of astrophysics with the Sloan Digital Sky Survey [69]. In most other areas, real-world adoption seems rather limited.

However, researchers in the HEP domain do seem to see the need to look beyond their mainstream tools. For instance, Pheasant [3] is a visual query language for expressing decay queries more easily. A more recent effort aims to open up particle physics analysis tools to the broader scientific community via the Scikit-HEP project [59]. Other works have looked into exploiting the efficiency of GPUs for HEP analyses [57]. Also the ROOT file format is subject to investigation: There are also several studies [9, 55] of the performance of the ROOT data format that compare it with general-purpose alternatives such as Parquet, Avro, or Protocol Buffers, as well as propositions for new file formats [16, 29]. Finally, several authors have proposed integrations [7, 35, 45, 46, 61] with modern big-data systems such as Apache Spark.

There are several studies that compare query languages for certain domains. For example, Ong et al. [52] analyze eleven document-oriented query languages, from which they eventually derive the design of SQL++, the language that is part of the comparison of this paper. While for a different use case, the study follows a similar structure as ours. A similar study compares high-level query languages in the context of MapReduce [67].

6 CONCLUSION

We have evaluated several general-purpose data processing systems in terms of suitability of their query language, absolute performance, scalability, and query price in the context of High-energy Physics (HEP). With the support for structured data types and arrays, several SQL dialects can express HEP analyses reasonably well, and languages for nested and heterogeneous data allow for more natural query formulations. However, the general-purpose data processing systems are significantly less performant than the domain-specific ROOT framework—due to limited scalability and inefficient handling of the data and queries relevant to HEP.

The observations of the study suggest several avenues for future research: On the one hand, efficiency on nested but *homogeneous* data needs to be improved in order to make database systems competitive—otherwise, the potential of their many advantages remains untapped. On the other hand, to enable real-world adoption, some further questions need to be solved: how to expose the large body of physics libraries offered by ROOT and similar frameworks in query languages, how to integrate query languages with plotting and archival facilities, and how to convince and prepare domain physicists to adopt a new tool chain.

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] Actian Corporation. Columnar Database for Big Data | Vector Analytic Database. Retrieved Aug. 18, 2021 from https://www.actian.com/analytic-database/vector-analytic-database/.

[2] Sattam Alsubaiee et al. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.*, 7, 14. DOI: 10.14778/2733085.2733096.

[3] Vasco Amaral, Sven Helmer, and Guido Moerkotte. 2003. A Visual Query Language for HEP Analysis. In *Nuclear Science Symposium*. DOI: 10.1109/NSSMIC.2003.1351826.

[4] I. Antcheva et al. 2009. ROOT — A C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 180, 12. DOI: 10.1016/J.CPC.2009.08.005.

[5] Michael Armbrust et al. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. DOI: 10.1145/2723372.2742797.

[6] Andrew Baden, Chris Day, Robert Grossman, Dave Lifka, Ewing Lusk, Edward May, and Larry Price. 1991. Analyzing high energy physics data using database computing: Preliminary report.

[7] Zbigniew Baranowski et al. 2019. Evolution of the Hadoop Platform and Ecosystem for High Energy Physics. *EPJ Web of Conferences*, 214. DOI: 10.1051/EPJCONF/201921404058.

[8] Pavel Binko, Dirk Duellmann, Jamie Shiers, Pavel Binko, Dirk Duellmann, and Jamie Shiers. 1996. CERN RD45 Status Report - A Persistent Object Manager for HEP. In *CHEP*. DOI: 10.1142/9789814447188_0061.

[9] Jakob Blomer. 2018. A quantitative review of data formats for HEP analyses. *Journal of Physics: Conference Series*, 1085. DOI: 10.1088/1742-6596/1085/3/032020.

[10] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *CACM*, 51, 12. DOI: 10.1145/1409360.1409380.

[11] M. Bowen, Greg L. Landsberg, and Richard Partridge. 2000. The physics analysis server project. In *Computing in High-Energy and Nuclear Physics*.

[12] Rene Brun and Fons Rademakers. 1997. ROOT — An object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389, 1. DOI: 10.1016/S0168-9002(97)00048-X.

[13] Rene Brun et al. 2019. Root-project/root: v6.18/02. Version v6-18-02. DOI: 10.5281/zenodo.3895860.

[14] Paolo Calafiura, James Catmore, Davide Costanzo, and Alessandro Di Girolamo. 2020. ATLAS HL-LHC Computing Conceptual Design Report. Tech. rep. CERN-LHCC-2020-015. http://cds.cern.ch/record/2729668.

[15] Andrea Ceccarelli, Andrea Cioni, Maria Vittoria Garzelli, Piergiulio Lenzi, and Laura Redapi. Towards enhanced databases for High Energy Physics. In *Deep-Inelastic Scattering and Related Subjects*. 2019. ADS Bibcode: 2019disr.confE.223C.

[16] Jin Chang, Oliver Gutsche, Igor Mandrichenko, and James Pivarski. 2018. Striped Data Server for Scalable Parallel Data Analysis. *Journal of Physics: Conference Series*, 1085, 4. DOI: 10.1088/1742-6596/1085/4/042035.

[17] CMS collaboration. 2017. SingleMu primary dataset in AOD format from Run of 2012 (/SingleMu/Run2012B-22Jan2013-v1/AOD). CERN Open Data Portal. DOI: 10.7483/OPENDATA.CMS.IYVQ.1J0W.

[18] Compact Muon Solenoid - Wikipedia. Retrieved Feb. 16, 2021 from https://en.wikipedia.org/wiki/Compact_Muon_Solenoid.

[19] J Cranshaw, D Malon, A Vaniachine, V Fine, J Lauret, and P Hamill. 2010. Petaminer: Using ROOT for efficient data storage in MySQL database. *Journal of Physics: Conference Series*, 219, 4. DOI: 10.1088/1742-6596/219/4/042036.

[20] Benoit Dageville et al. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. DOI: 10.1145/2882903.2903741.

[21] Shaopeng Dai, Wanling Gao, Biwei Xie, Minghe Yu, Jia'nan Chen, Defei Kong, Rui Han, and Jinheng Li. 2018. Evaluating Index Systems of High Energy Physics. *Communications in Computer and Information Science*, 911. DOI: 10.1007/978-981-13-5910-1_2.

[22] Dirk Düllmann. 1999. Petabyte Databases. *SIGMOD Rec.*, 28, 2. DOI: 10.1145/304181.304229.

[23] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35, 1.

[24] A Fry and I Chow. 1993. Integrating PAW, a graphical analysis interface to Sybase.

[25] Dan Graur, Ingo Müller, Ghislain Fourny, Mason Proffitt, Gordon T. Watts, and Gustavo Alonso. 2021. Evaluating Query Languages and Systems for High-Energy Physics Data [extended version]. arXiv: 2104.12615 [cs.DB].

[26] Dan Graur, Ingo Müller, Mason Proffitt, Ghislain Fourny, Gordon T. Watts, and Gustavo Alonso. 2021. Benchmark Scripts for Evaluating Query Languages and Systems for High-Energy Physics Data. DOI: 10.5281/zenodo.5569049.

[27] R. Grossman et al. 1994. Analyzing high energy physics data using databases: a case study. *Scientific and Statistical Database Management - Proceedings of the International Working Conference*. DOI: 10.1109/SSDM.1994.336938.

[28] Enrico Guiraud, Axel Naumann, and Danilo Piparo. 2017. TDataFrame: functional chains for ROOT data analyses. Version v1.0. DOI: 10.5281/zenodo.260230.

[29] Oliver Gutsche and Igor Mandrichenko. 2020. Striped Data Analysis Framework. *EPJ Web of Conferences*, 245. DOI: 10.1051/EPJCONF/202024506042.

[30] Oliver Gutsche et al. 2017. Big Data in HEP: A comprehensive use case study. *Journal of Physics: Conference Series*, 898. DOI: 10.1088/1742-6596/898/7/072012.

[31] Michael Hausenblas and Jacques Nadeau. 2013. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big Data*, 1, 2, 100–104. DOI: 10.1089/BIG.2013.0011.

[32] Institute for Research and Innovation in Software for High Energy Physics. Retrieved Feb. 16, 2021 from https://iris-hep.org/.

[33] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive Query Processing on RAW Data. *Proc. VLDB Endow.*, 7, 12. DOI: 10.14778/2732977.2732986.

[34] David Kernert, Norman May, Michael Hladik, Klaus Werner, and Wolfgang Lehner. 2015. From static to agile - Interactive particle physics analysis in the SAP HANA DB. DOI: 10.5220/0005503700160025.

[35] Viktor Khristenko and Jim Pivarski. 2017. diana-hep/spark-root: Apache Spark Data Source for ROOT File Format. Version v0.1.14. DOI: 10.5281/zenodo.1034230.

[36] Large Hadron Collider - Wikipedia. Retrieved Feb. 16, 2021 from https://en.wikipedia.org/wiki/Large_Hadron_Collider.

[37] Dr Maaike Limper. 2014. An SQL-based approach to physics analysis. *Journal of Physics: Conference Series*, 513, 2, 022022. DOI: 10.1088/1742-6596/513/2/022022.

[38] Dmitry Lychagin. 2021. Producing combinations of array elements in SQL++. https://mail-archives.apache.org/mod_mbox/asterixdb-users/202108.mbox/%3C9A48EB5A-E4EE-403B-BED7-7331D471BB7D%40couchbase.com%3E.

[39] D Malon, J Cranshaw, P van Gemmeren, and Q Zhang. 2011. Emerging Database Technologies and Their Applicability to High Energy Physics: A First Look at SciDB. *Journal of Physics: Conference Series*, 331, 4. DOI: 10.1088/1742-6596/331/4/042016.

[40] David M. Malon and Edward N. May. 1997. Critical Database Technologies for High Energy Physics. In *VLDB*.

[41] David M. Malon, Edward N. May, Robert L. Grossman, Christopher T. Day, and David R. Quarrie. 1995. Object Database Standards, Persistence Specifications, and Physics Data. In *CHEP*. DOI: 10.1142/9789814447188_0058.

[42] J. Marstaller. 1993. Comparative performance measures of relational and object-oriented databases using High Energy Physics data.

[43] Alberto Di Meglio. 2017. Facing up to the exabyte era | CERN. Retrieved Apr. 1, 2021 from https://home.cern/news/opinion/computing/facing-exabyte-era.

[44] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3, 1-2.

[45] Andrew Melo and Jim Pivarski. 2020. spark-root/laurelin: Allows reading ROOT TTrees into Apache Spark as DataFrames. Version v1.1.1. https://github.com/spark-root/laurelin.

[46] M Meoni, V Kuznetsov, L Menichetti, J Rumševičius, T Boccali, and D Bonacorsi. 2018. Exploiting Apache Spark platform for CMS computing analytics. *Journal of Physics: Conference Series*, 1085, 3. DOI: 10.1088/1742-6596/1085/3/032055.

[47] Microsoft. SQL Server technical documentation - SQL Server | Microsoft Docs. Retrieved Aug. 18, 2021 from https://docs.microsoft.com/en-us/sql/sql-server.

[48] Microsoft Azure. Azure Synapse Analytics. Retrieved Aug. 18, 2021 from https://azure.microsoft.com/en-us/services/synapse-analytics/.

[49] Ingo Müller, Ghislain Fourny, Stefan Irimescu, Can Berker Cikis, and Gustavo Alonso. 2020. Rumble: Data Independence for Large Messy Data Sets. *Proc. VLDB Endow.*, 14, 4. DOI: 10.14778/3436905.3436910.

[50] Ildar Musin. 2021. adjust/parquet_fdw: Parquet foreign data wrapper for PostgreSQL. Version 0.2.1. Retrieved Aug. 14, 2021 from https://github.com/adjust/parquet_fdw.

[51] M Nowak, Zoltán Kunszt, D Geppert, S Paoli, and D Düllmann. 2001. Object Persistency for HEP data using an Object-Relational Database. http://cds.cern.ch/record/518801.

[52] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases. arXiv: 1405.3631v4.

[53] Oracle Corporation. MySQL. Retrieved Aug. 18, 2021 from https://www.mysql.com/.

[54] Orcale. HeatWave. Retrieved Aug. 18, 2021 from https://www.oracle.com/mysql/heatwave/.

[55] Jim Pivarski. 2013. Survey of data formats, conversion tools. In *HEP Analysis Ecosystem Workshop*. https://indico.cern.ch/event/613842/contributions/2585787/.

[56] Jim Pivarski, David Lange, and Thanat Jatuphattharachat. 2018. Toward real-time data query systems in HEP. *Journal of Physics: Conference Series*, 1085, 3. DOI: 10.1088/1742-6596/1085/3/032044.

[57] Alexis Pompili and Adriano Di Florio and. 2016. GPUs for statistical data analysis in HEP: a performance study of goofit on GPUs vs. RooFit on CPUs. *Journal of Physics: Conference Series*, 762. DOI: 10.1088/1742-6596/762/1/012044.

[58] Mason Proffitt, Ingo Müller, Mat Adamec, Pieter David, Enrico Guiraud, and Sebastien Binet. 2021. iris-hep/adl-benchmarks-index: ADL Functionality Benchmarks Index. Version v0.1. DOI: 10.5281/zenodo.5131287. https://github.com/iris-hep/adl-benchmarks-index/.

[59] Eduardo Rodrigues. 2019. The Scikit-HEP Project. *EPJ Web of Conferences*, 214. DOI: 10.1051/epjconf/201921406005.

[60] Kazunori Sato. 2012. An inside look at Google BigQuery. White paper. https://cloud.google.com/files/BigQueryTechnicalWP.pdf.

[61] Saba Sehrish, Jim Kowalkowski, and Marc Paterno. 2017. Spark and HPC for high energy physics data analyses. *International Parallel and Distributed Processing Symposium Workshops*. DOI: 10.1109/IPDPSW.2017.112.

[62] Amazon Web Services. Amazon Athena. Retrieved Feb. 16, 2021 from https://aws.amazon.com/athena/.

[63] Amazon Web Services. Amazon redshift - cloud data warehouse. Retrieved Oct. 12, 2021 from https://aws.amazon.com/redshift/.

[64] Raghav Sethi et al. 2019. Presto: SQL on everything. In *ICDE*.

[65] Srinath Shankar, Ameet Kini, David J. DeWitt, and Jeffrey Naughton. 2005. Integrating Databases and Workflow Systems. *SIGMOD Rec.*, 34, 3. DOI: 10.1145/1084805.1084808.

[66] Jamie Shiers. 2011. Databases in High Energy Physics: A Critical Review. DOI: 10.1007/978-3-642-23157-5_9.

[67] R. J. Stewart, P. W. Trinder, and H. W. Loidl. 2011. Comparing High Level MapReduce Query Languages. In *APPT*. Vol. 6965 LNCS. DOI: 10.1007/978-3-642-24151-2_5.

[68] Michael Stonebraker and Lawrence A. Rowe. 1986. The design of POSTGRES. *ACM SIGMOD Record*, 15, 2. DOI: 10.1145/16856.16888.

[69] Alexander S. Szalay. 2008. The sloan digital sky survey and beyond. *SIGMOD Record*, 37, 2. DOI: 10.1145/1379387.1379407.

[70] The PartiQL Specification Committee. 2019. PartiQL Specification. https://partiql.org/assets/PartiQL-Specification.pdf.

[71] The PostgreSQL Global Development Group. PostgreSQL: Documentation: 8.16. Composite Types. Retrieved Aug. 14, 2021 from https://www.postgresql.org/docs/13/rowtypes.html#ROWTYPES-ACCESSING.

[72] Eric Torrence. 2019. Delivered Luminosity versus time for 2011-2018 (p-p data only). https://twiki.cern.ch/twiki/bin/view/AtlasPublic/LuminosityPublicResultsRun2.

[73] Benito van der Zander. Xidel repository. Retrieved Mar. 30, 2021 from https://github.com/benibela/xidel.

[74] V Vassilev. 2015. Native Language Integrated Queries with CppLINQ in C++. *Journal of Physics: Conference Series*, 608, 1. DOI: 10.1088/1742-6596/608/1/012030.

[75] Alexandre Verbitski et al. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. DOI: 10.1145/3035918.3056101.

[76] Worldwide LHC Computing Grid. About. Retrieved Feb. 16, 2021 from https://wlcg-public.web.cern.ch/about.

[77] Zorba. Zorba documentation. Retrieved Mar. 30, 2021 from http://www.zorba.io/documentation/latest.