Addressing the Nested Data Processing Gap: JSONiq Queries on Snowflake through Snowpark

Addressing the Nested Data Processing Gap: JSONiq Queries on Snowflake through Snowpark

Dan Graur
ETH Zürich

Remo Röthlisberger
ETH Zürich

Adrian Jenny
ETH Zürich

Ghislain Fourny
ETH Zürich

Filip Drozdowski
Snowflake

Choden Konigsmark
Snowflake

Ingo Müller
ETH Zürich

Gustavo Alonso
ETH Zürich

***Abstract*—Nested data is common in many use cases but querying it is still not well supported. Options available today include using: (1) SQL extensions, which are often unintuitive and error-prone; (2) user-defined functions, which limit portability and reusability, and often reduce performance; or (3) domain-specific query languages (DSQL), which often have limited scalability and performance. In this paper, we address the shortcomings of the latter approach by translating a language specifically designed for nested data, JSONiq, to a highly efficient, scalable, and feature-rich RDBMS, the Snowflake Database. For this purpose, we use the Snowpark API, a data-frame-based client library for writing applications on Snowflake, which allows us to translate each JSONiq query into a *single* native Snowflake SQL query. In contrast to previous approaches, this does not introduce any interpretation overhead or optimization barriers that may limit efficient execution in the target system. We evaluate the resulting system on an established benchmark for large-scale nested data from the high-energy physics (HEP) domain on up to 1 TiB as well as the SSB benchmark from the relational domain. Our approach is on par or better than handwritten SQL baselines while allowing for significantly more readable query formulations and typically outperforms the state-of-the-art systems specialized for nested data by an order of magnitude.**

***Index Terms*—databases, query languages, nested data**

I. Introduction

Nested data has grown in volume due to the ubiquitous nature of the internet, the digitization of businesses, and the ever-increasing diversity of data sources. Processing this type of data constitutes an important challenge for businesses and consequently represents a profitable market for database vendors. To gain more market share, vendors are in continuous competition with each other in adding new features that provide better support for storing and processing such data [1]–[6]. However, solutions for analyzing nested data are still sub-optimal. Practitioners essentially have the choice between (1) SQL with extensions, (2) SQL with user-defined functions (UDFs), or (3) domain-specific languages, all of which have significant downsides: While there *is* support for nested data in the SQL standard (via the `ARRAY` and **`ROW`** data types and related functions), a recent study [7] shows that only very few systems implement enough of the standard for even moderately complex queries because, among other limitations, they lack support for the most basic functionality like unnesting structures, defining custom structured types, or supporting nested queries. The systems that do support this part of the standard are most often not fully compliant or otherwise cumbersome to use. User-defined functions (UDFs) may allow users to express more complex logic on nested data but they are non-portable, usually sacrifice performance, and frequently disallow complex return types such as tables. In contrast, domain-specific query languages (DSQLs) are more expressive and provide a more readable interface; however, the performance of existing systems is one or more orders of magnitude worse than that of state-of-the-art RDBMSs [7].

In this paper, we address this gap in the processing of nested data by making the execution of DSQLs efficient and scalable. Concretely, we translate JSONiq [8]–[10], a language designed for semi-structured data with excellent support for nested data, to the SQL dialect of Snowflake [11], an efficient and scalable cloud-native RDBMS. This combines the best of two worlds: a natural, concise way for specifying queries on nested data and the benefits of a best-in-class RDBMS.

Translating a DSQL such as JSONiq to Snowflake SQL is a complex task. The key challenge is mapping each construct of JSONiq to one or several constructs of the target language (SQL) with identical semantics. While many constructs have a direct equivalent to Snowflake SQL, we show that a few require the careful combination of several target constructs that would be cumbersome and impractical to write manually. Unlike previous work [12], which falls back to imperative logic using UDFs or local single-node execution, we can translate any arbitrary JSONiq query in its entirety to a single SQL query that expresses its full semantics. This gives the SQL optimizer end-to-end visibility into the query and eliminates the overhead for language switching, data copying, intermediate result materialization, and UDF interpretation, thus improving performance substantially.

We exploit several other features of Snowflake. First, we use Snowpark [13], a data-frame-based API that makes assembling SQL queries programmatically safe and convenient. Second, we expose Snowflake's support for nested data formats (JSON, XML, Parquet, etc.) to allow for in-situ processing without manual schema definition or data loading, as well as processing of existing relational tables (with or without nested data). In addition to Snowflake's performance, we thus also benefit from its feature-richness and ecosystem.

```
{
  "EVENT": 263142897,
  "HLT": {"IsoMu24": false, ...},
  "ELECTRON":  [{"charge": -1, ...}, ...],
  "JET": [{"btag": 0.99983340, ...}, ...],
  "MUON": [{"pt": 12.777096, ...}, ...],
  ...
}
```

(a) Shortened element example.

```
{
  "EVENT": "long",
  "HLT": {"IsoMu24": "boolean", ...},
  "ELECTRON":  [{"charge": "integer", ...}, ...],
  "JET": [{"btag": "double", ...}, ...],
  "MUON": [{"pt": "double", ...}, ...],
  ...
}
```

(b) Shortened JSound schema.

Fig. 1: Schema and element example from the ADL dataset.

We evaluate our work on two established benchmarks: (1) the ADL benchmark [14], which stems from the high-energy physics (HEP) domain and consists of eight complex queries on a large-scale dataset that is naturally nested and (2) the Star Schema Benchmark (SSB) from the relational domain [15]. We show that our work can generate automatically translated queries with on-par performance to handwritten SQL queries while starting from queries that are significantly more concise and readable. Compared with other state-of-the-art systems with DSQLs for nested data, our approach is one to two orders of magnitude faster, thus greatly narrowing the nested data processing gap. Our work also contributes to the long-standing question of whether relational databases can efficiently store nested data and query it using DSQLs [16], [17]. This paper highlights how flexible cloud-native databases have become and how they can be further used to tackle similar challenges.

II. Background

A. Nested Data

Nested data, also known as homogeneous semi-structured data is a subclass of semi-structured data [18], [19]. In relational terms, nested data is a class of unnormalized or non-first normal form (NF^2) data. It is defined by a fixed tree-like structure, from which a schema can be derived. Figure 1a shows a shortened version of a nested element from the IRIS ADL dataset and Figure 1b shows the shortened JSound [20] schema of the dataset. In contrast, heterogeneous semi-structured data has a highly flexible structure (e.g., elements of different types and cardinality at the same paths). We focus on nested data. Still, we believe this approach is generalizable to heterogeneous semi-structured data as well, given Snowflake's out-of-the-box support for such data types.

Our work differs from related research that has focused on the XML data format and its associated query languages (e.g. XPath and XQuery), as XML is a document type with strong semantics and advanced features (e.g. ability to mix text and node elements, ordering guarantees, node attributes, various types of traversals, etc.). XML-related work has sought to abide by these semantics, tightly binding contributions to the format itself. By only focusing on the nestedness of data, we are not bound to any specific data format or format-specific semantics. As we show with this work, this enables much higher scalability and performance than document-centric querying. Furthermore, our work applies to any nested data format (e.g., XML serialization of records, JSON, Parquet, etc.)—and we only use JSON in the examples of this paper because it is widespread and human-readable.

B. The Snowflake VARIANT Type and Physical Layout

The VARIANT data type offers a set of features for efficiently storing and processing nested data at scale without requiring an explicit schema [11], [21], [22]. Nested data in a VARIANT column is transparently columnarized: elements at each unique path are physically stored in their own column and are transparently cast to their lowest common type to avoid runtime casting overheads. When no common type exists, fast runtime casting is used [11]. Snowflake infers the nested objects' schema and exploits it during query runtime to correctly address columns and rebuild objects. These features also ensure that large or deeply nested objects are not penalized, as the engine ultimately deals with flattened, columnarized data.

Data in Snowflake is physically stored in horizontal table shards called micro-partitions. Each shard contains 50 MB to 500 MB of originally uncompressed row data. Within micro-partitions, data is stored compressed in a columnar fashion allowing for efficient granular scans. To enable scalability and to avoid maintenance overheads, Snowflake does not use indexes. Instead, it stores schema and micro-partition level statistics and metadata [21]. Schema-level statistics can be used to point to micro-partitions that contain specific values. Micro-partition-level metadata and statistics include zone maps and the unique value count for a micro-partition-level column. This information is used by the optimizer and runtime to effectively find and prune micro-partitions and columns.

Since VARIANT data is stored as many columns and reuses Snowflake's micro-partition logic and infrastructure, the optimizer and runtime can leverage nested data just like relational data and generally does not need to be aware of fine-grained information that is specific to semi-structured data. Some metadata can be used during query planning and execution, such as the objects' schema, types, and nestedness. The practitioner is allowed to stage nested data however they prefer: single or multiple VARIANT columns within one or more tables (§III-C).

C. The IRIS HEP ADL Benchmark

IRIS HEP ADL is a benchmark originating from the High-Energy Physics (HEP) domain [7], [14]. This benchmark is an excellent fit for the problem at hand: it consists of naturally nested, fully homogeneous data, its queries involve deeply nested logic, and performance and scalability are fundamental. Furthermore, it has been implemented across 13 systems, from BigQuery [23] to RDataFrames [24].[1] We thus use it as a running example throughout the document.

[1]ADL benchmark: https://github.com/iris-hep/adl-benchmarks-index

```java
DataFrame df = session.table("orders");
Column lower = Functions.lit(90000);
Column upper = Functions.lit(120000);
Column totalPrice = Functions.col("o_totalprice");
Column clerks = Functions.col("o_clerk");
df.where(totalPrice.between(lower, upper))
  .select(Functions.count_distinct(clerks))
  .collect();
```

(a) The query written in Snowpark's Java API.

```sql
SELECT count(DISTINCT "O_CLERK")
FROM (
  SELECT *
  FROM (SELECT * FROM (orders))
  WHERE (
    ("O_TOTALPRICE" >= 90000 :: int)
   AND ("O_TOTALPRICE" <= 120000 :: int)))
```

(b) The generated SQL code.

Fig. 2: Snowpark Java example on the TPC-H dataset.

ADL features eight complex queries and, at Scale Factor 1, uses a dataset of 17 GiB containing roughly 54 million rows (i.e., *events*), each consisting of event metadata and the high-energy particle properties observed during the event. Figure 1a shows a simplified event sample.[2] The queries and data stem from the 2012 Compact Muon Solenoid experiment at the Large Hadron Collider, which led to the discovery of the Higgs boson particle. ADL constitutes an important step forward for nested data research, as it is the only modern complete benchmark (data and queries) for this data type [7].

The ADL queries define challenging nested data workloads. At their simplest (Q1), they involve projection and histogramming. At their most complex (Q8), they involve unpacking several levels of nested data, applying complex HEP formulas, creating and manipulating new heterogeneous particle structures, before histogramming specific properties. For full benchmark details, we defer to [7].

D. The Snowpark Library

Snowpark is a library that offers a comprehensive set of API calls for programmatically building Create, Read, Update, Delete (CRUD) queries [25] that manipulate data stored in Snowflake. Snowpark currently offers APIs for Java, Scala, and Python. Snowpark exposes SQL operations as method calls rather than requiring its users to specify the SQL code themselves manually. Besides making query writing less error-prone and more maintainable, this additional level of abstraction allows Snowpark to transparently construct and optimize the generated queries to run on Snowflake.

Snowpark exposes three important classes: `DataFrame`, `Column`, and `Functions`. A `DataFrame` object logically encapsulates a fully executable SQL query. Transformations applied to it are lazy and generate a new `DataFrame` object. Table I shows some of the methods exposed by the `DataFrame` class. These methods generally correspond to SQL clauses (**SELECT**, **WHERE**, **GROUP BY**, etc.). `Column` objects represent subexpressions in queries (e.g. literals, arithmetic operations, logical operations, etc.). `Column` objects are not bound to any dataset and only represent partial pieces of SQL logic, meaning they are not executable. `Column` objects are composed with one another either via methods in their API or via static functions from the `Functions` class. Table I exemplifies some of these methods. In either case, new `Column` objects are returned as part of the composition process. `Column` objects are plugged in as arguments into methods from the `DataFrame` class, generating new `DataFrame` objects that represent the target query logic.

[2]The benchmark's repository offers a more complete, human-readable sample: https://github.com/RumbleDB/hep-iris-benchmark-scripts/blob/master/datasets/samples/

Return Type	Parent Class	Method
DataFrame	DataFrame	`select()`, `where()`, `drop()`, `withColumn()`, `groupBy()`, `flatten()`, `collect()`, ...
Column	Column	`add()`, `mul()`, `mod()`, `and()`, `lt()`, `in()`, `subField()`, ...
Column	Functions	`object_construct()`, `col()`, `lit()`, `sum()`, `seq()`, `atan()`, `array_agg()`, `abs()`, ...

TABLE I: Important Snowpark API classes and methods.

```
for $jet in collection("adl").Jet[]
where abs($jet.eta) lt 1
return $jet.pt
```

Listing 1: Simplified ADL Q3 JSONiq reference code.

As `DataFrame` objects are lazy and only reflect a logical query plan, the execution of a query is triggered only upon calling certain methods on the `DataFrame` (e.g. `collect()`, `take()`). This is similar to Spark [26]. We take full advantage of the lazy nature of the Snowpark API and ensure the query translation is lazy from start to finish, producing a single, fully native SQL query that completely represents the original query logic without using UDFs (i.e. only standard SQL). When `DataFrame` materialization is triggered, the query is directly and fully executed in Snowflake.

Figure 2a exemplifies how Snowpark can be used to query the number of distinct clerks associated with orders having a total price between 90000 and 120000 on the TPC-H dataset [27]. The generated SQL query is shown in Figure 2b.

E. The JSONiq Query Language

JSONiq is a fully specified, mature query language, tailor-made for the semi-structured data model. It employs the FLWOR expression set (**for**, `let`, **where**, **order by**, **count**, **group by** and **return**) [10]. FLWOR clauses have similar semantics to their SQL counterparts (e.g. **return** matches **SELECT**, **where** matches **WHERE**, **for** matches **FROM**, etc.), bar `let` which does not have a direct counterpart in SQL. `let` can be seen as the addition of a new column, and can, for instance, be expressed via a **SELECT** *, <**new-column**>. SQL **JOIN**s can be expressed as successive **for** clauses that read different tables. **where** can be used to represent **JOIN** conditions.

JSONiq is the successor to XQuery [28] and is inspired by the JSON format. This enables practitioners to express complex nested query logic concisely and with relative ease. This contrasts with SQL where certain patterns required for processing semi-structured data are not easily expressible [7]. JSONiq has an imperative feel to it, which stems from its FLWOR expression set. This is exemplified in Listing 1. In Line 1, the `Jet` entry in the `"adl"` dataset is unboxed. Each unboxed element is bound to the `jet` alias as part of the **`for`** clause. In Line 2, each `jet` element is discarded if its `eta` subfield's absolute value is less than 1. Finally, in Line 3, the `pt` subfield of the `jet` objects that passed the **`where`** clause are returned. While JSONiq is primarily designed for handling semi-structured data, it is not bound to this data model and can also be used to query relational data (§V-G) [9].

III. Design and Implementation

We show how the Snowpark API enables us to translate JSONiq queries to SQL for execution on Snowflake. The core challenge of doing so consists of mapping each language construct of JSONiq to one or several constructs in the Snowpark API that have the same semantics and result in SQL queries that use the full performance of Snowflake. To this extent, we employ a classical approach from compiler theory and lower a query from JSONiq to SQL via several intermediate representations. To facilitate this, we use RumbleDB [12], a state-of-the-art system for querying semi-structured data with excellent support for JSONiq. RumbleDB readily converts a query to an AST, expression tree, and finally, an iterator tree, where it defers execution in a hybrid manner to Spark and native Java. We reimplement the behavior of the iterator tree in Snowpark to generate a single SQL query that entirely encompasses the logic of the original JSONiq query. We continue to use the iterator naming for clarity, even though each iterator is visited exactly once during query translation.

A. System Architecture

1) Overview.: To execute a query, the user submits a JSONiq query to a local RumbleDB process via one of its interfaces (REPL client, command line client, or REST server). Our back-end translates the JSONiq query into a single native SQL query that fully encapsulates the original JSONiq logic. The translation happens in several phases, which can be split into two categories. The first phases are the construction of an abstract syntax tree (AST) and a subsequent conversion into an expression tree, which are back-end-agnostic. The third phase is the construction of a tree of iterators, which have back-end-specific execution modes.

2) Parsing layer.: RumbleDB's parsing layer converts JSONiq queries into logical query plans. This process is still back-end-agnostic. It does so by parsing the submitted query into an AST and then converting the AST into an expression tree using standard techniques. In the resulting expression tree, virtually every node represents a JSONiq operation that is specified in the query text. In that form, RumbleDB applies several query rewrites that optimize the expression tree (e.g. dead-code elimination, function inlining, etc.).

3) SQL Query Construction Layer.: RumbleDB then translates the expression tree and converts it into an iterator tree—a tree that has the same structure as the expression tree but whose nodes are "iterators". The default execution mode of these iterators is the traditional way of implementing iterators in database systems with an open/next/close interface. However, in RumbleDB, back-ends can extend iterators by implementing their own execution mode.

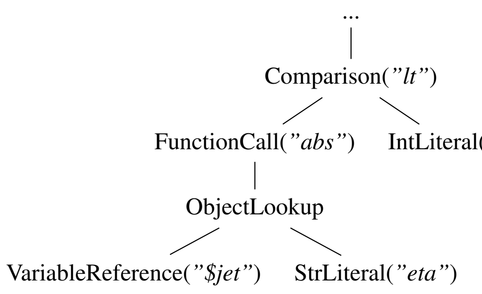
For our Snowflake back-end, we add a new execution mode that, for each node in the iterator tree, computes a `DataFrame` or `Column` object using the Snowpark API. We extend RumbleDB's iterator interface with the `processNativeSnowflake` method that computes a `DataFrame` or `Column` object. We do this for each iterator type in RumbleDB and give details for a selection of them in Section III-B. When the execution of the root iterator is triggered, it assembles a Snowpark API object by recursively asking its child iterators to do the same.

With this approach, the JSONiq query is translated to Snowflake SQL in its entirety. This is in contrast with previously existing back-ends of RumbleDB (e.g. Spark), which carried out pre- and post-processing in the local RumbleDB process and fell back to injecting slow Java-based iterators into UDFs for some of the more complex JSONiq constructs [12]. This often implies slow execution, data movement, and serialization overheads. In contrast, the Snowpark API and the underlying SQL dialect are expressive enough to natively express the subset of JSONiq that RumbleDB supports. This not only saves round-trips to and from the server but also gives the Snowflake optimizer the possibility to optimize the query holistically and without blackboxes caused by UDFs.

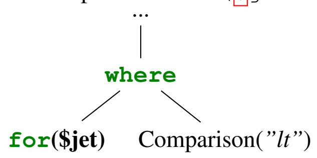
This design does not involve any modification in Snowflake but treats it transparently as any other client would. This implicitly means that the Snowflake optimizer is not aware of the translation layer, and the translation layer does not cater to the optimizer, beyond using the Snowpark API as intended. The Snowpark API exposes Snowflake's query language in a programmatic interface that enables us to express the logic of JSONiq queries with purely native constructs. As the evaluation shows, we can obtain on-par performance with handwritten SQL (§V). We posit, however, that if our back-end was moved *inside* Snowflake itself, such that users would send JSONiq queries to Snowflake directly, further performance improvements may be possible than what is currently achievable via the SQL interface.

B. The SQL Query Construction Layer

The SQL query construction layer generates an SQL query with identical behavior and semantics as the original JSONiq query. The query construction is done iteratively by traversing the iterator tree once in a post-order fashion and gradually building up the SQL query logic in each node by combining the SQL subexpressions of the children nodes using Snowpark calls. The final product of the translation is a single `DataFrame` capturing the original JSONiq query logic, which can be used to schedule query execution in Snowflake.



(a) Non-FLWOR expression: `abs($jet.eta) lt 1`



(b) FLWOR expression: `where` clause

Fig. 3: Listing 1 expression tree examples.

```
1  public Context processComparison(
2    Context context) {
3    Context lContext = this.children.get(0)
4      .processNativeSnowflake(context);
5    Context rContext = this.children.get(1)
6      .processNativeSnowflake(context);
7    Column lColumn = lContext.getColumn();
8    Column rColumn = rContext.getColumn();
9
10   Column resultColumn;
11   switch (this.comparisonType) {
12     case "lt":
13       resultColumn = lColumn.lt(rColumn);
14       break;
15     ...
16   }
17   return new Context(resultColumn);
18 }
```

Listing 2: `ComparisonIterator` simplified implementation.

The iterator tree features two important classes of nodes: (1) FLWOR clauses (i.e. the iterators representing **for**, let, **where**, **order by**, **count**, **group by** and **return**) and (2) non-FLWOR iterators (e.g. comparisons). We detail their high-level behavior and highlight how the Snowpark API is employed in the next sections (§III-B1,§III-B2). The more complex iterator behavior is discussed in Section IV.

1) Non-FLWOR Iterators: Non-FLWOR iterators represent partial query logic. Consider the subtree rooted in the `Comparison("lt")` iterator in Figure 3a. The logic of the subtree represents the comparison $|\text{jet.eta}| < 1$. The semantics of this subexpression become clear only when attached to a FLWOR clause. This is similar to how SQL subexpressions work: a subexpression in a **WHERE** clause represents a predicate, while in a **SELECT** clause it generates a new column.

This equivalence between JSONiq and SQL allows us to neatly exploit the Snowpark API, which is primarily designed with SQL in mind. Snowpark `Columns` occupy a similar role in the API as JSONiq's non-FLWOR expressions: they represent the partial pieces of query logic that are ambiguous until attached to an SQL clause.

In normal scenarios, non-FLWOR iterators receive `Column` objects from their children iterators. Incoming `Columns` are composed using Snowpark to implement the iterator's behavior. Each non-FLWOR iterator produces a single `Column` object that is returned to the parent iterator. The returned `Column` object encapsulates the partial query logic rooted in the non-FLWOR iterator. Listing 2 shows the implementation of the `ComparisonIterator`: the iterator fetches the `Columns` returned by the two children iterators (Lines 3-8) and applies the relevant comparison (Lines 11-16), generating a new `Column` that is returned to the parent iterator (Line 17).

This design of the non-FLWOR iterators produces concise SQL translations and decreases the amount of nesting in the translated SQL queries. For instance, the JSONiq predicate `where abs(delta) gt 1` ultimately gets converted to `SELECT * FROM source WHERE abs(delta) > 1 :: int`. The alternative approach of storing the results of non-FLWOR iterators directly in `DataFrames` generates nested verbose queries. The same JSONiq predicate would get translated to `SELECT * FROM (SELECT *, 1 :: int AS right FROM (SELECT *, abs(delta) AS left FROM source)) WHERE left > right`. Such translations would require more careful schema management to remove the partial results (e.g. **left**, **right** columns).

2) FLWOR Iterators: FLWOR clauses map closely to SQL clauses. For instance, a JSONiq **return** matches an SQL **SELECT**, a **for** matches **FROM**, a **where** matches **WHERE**, and so on. `let` clauses are without a direct counterpart in SQL, but can be straightforwardly expressed through **SELECT** clauses. In Snowpark, SQL clauses are exposed as methods in the `DataFrame` class. We design FLWOR iterators to manipulate `DataFrame` objects and implicitly generate the next query execution stages in the SQL translation.

Unlike non-FLWOR iterators, which can have an arbitrary number of children depending on what behavior they implement, FLWOR clauses possess exactly two (except the FLWOR clauses at the beginning of a JSONiq query). The left child points to the preceding FLWOR clause iterator of the original query, while the right child points to the subexpression attached to the clause. Figure 3b shows how the **where** clause in Listing 1 has a left child pointing to the **for** iterator and a right child pointing to the **where**'s subexpression (Figure 3a). FLWOR iterators fetch the `DataFrame` returned by the left child and the `Column` returned by the right child. The incoming `DataFrame` encapsulates the fully executable query logic up to the current iterator, while the incoming `Column` encapsulates the logic of the iterator's subexpression. The `Column` is applied to the incoming `DataFrame` using the appropriate `DataFrame` API call. This generates a new `DataFrame` that is returned to the parent FLWOR iterator.

In complex cases where the right subtree features a nested query, this pattern is changed. To enable such cases, the `DataFrame` incoming from the left child is always passed into the right child (i.e. the nested query) such that, if need be, it can be directly modified there. In such cases, the

```java
1  public Context processWhere(
2    Context context) {
3    // Execute child FLWOR clause
4    Context lContext = this.leftChild
5      .processNativeSnowflake(context);
6    DataFrame incomingDF = lContext.getDF();
7
8    // Execute right child subexpression
9    Context subContext = new Context(
10     context, incomingDF);
11   Context rContext = this.rightChild
12     .processNativeSnowflake(subContext);
13
14   DataFrame resultDF;
15   if (rContext.isSimpleQuery()) {
16     resultDF = incomingDF.where(
17       rContext.getColumn());
18   } else {
19     resultDF = cleanup(incomingDF, rContext);
20   }
21   return new Context(resultDF);
22 }
```

Listing 3: `WhereClauseIterator` simplified implementation.

FLWOR iterator generally performs cleanup operations on the `DataFrame`, such as eliminating temporary columns. We discuss situations where this pattern occurs in Section IV-D.

Listing 3 exemplifies a simplified version of the **`where`** iterator. The incoming `DataFrame` from the left child is captured (Lines 4-6). The right child is executed, passing the left incoming `DataFrame` to it in case it hosts a nested query (Lines 9-12). Finally, if there was no nested query rooted in the right child, the iterator applies the **`where`** clause of the incoming `DataFrame` (Lines 15-17); otherwise, it applies its cleanup logic (Line 19). In either case, the resulting `DataFrame` is returned to the parent iterator (Line 21).

C. Data Model

We do not require a schema and do not impose any restrictions on how nested data should be staged. The nested data can be stored in a single table or across multiple tables. Tables can have a single VARIANT-type column that fully encapsulates an object or several VARIANT columns for each nested top-level entry in an object (e.g., for ADL this would mean a dedicated column for `JET`, `MUON`, `ELECTRON` etc.) in combination with other Snowflake typed columns (e.g., `NUMBER(38, 0)` for `EVENT`) and anything in between. For the ADL evaluations, we stage the data in a single table, using the latter schema (i.e., the *multi-column* version). In all cases, each row in the Snowflake table represents an object.

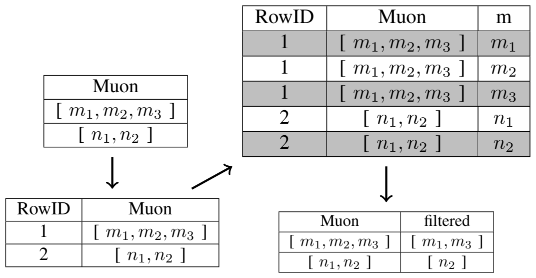
Data can also be staged in multiple tables that can be later joined. This data can be purely relational, nested, or even a hybrid combination of nested and relational data. To support these use cases, we implement support for **`JOIN`** operations, as described in the JSONiq standard [10].

IV. Complex Query Patterns

JSONiq and Snowflake SQL have a large amount of common ground between them. There are, however, several scenarios where the two diverge. We explore these cases here.

```
1  for $event in collection("adl")
2  let $filtered := (
3    for $m in $event.Muon[]
4    where predicate($m)
5    return $m
6  )
7  ...
```

Listing 4: A nested query example, where the Muon array in each top-level object is unboxed. Individual muons are filtered. The passing muons for each top-level object are aggregated into arrays and aliased by the name `$filtered`.



Muon
[m_1, m_2, m_3]
[n_1, n_2]

RowID	Muon
1	[m_1, m_2, m_3]
2	[n_1, n_2]

RowID	Muon	m
1	[m_1, m_2, m_3]	m_1
1	[m_1, m_2, m_3]	m_2
1	[m_1, m_2, m_3]	m_3
2	[n_1, n_2]	n_1
2	[n_1, n_2]	n_2

Muon	filtered
[m_1, m_2, m_3]	[m_1, m_3]
[n_1, n_2]	[n_2]

Fig. 4: Example trace of the subquery in Listing 4. Rows that pass the predicate are highlighted in gray.

A. Nested Data Access and Array Unboxing

Nested data access is a core functionality of JSONiq. We enable this common use case using the `Column::subField()` method which accesses VARIANT data subfields. Array unboxing is also an essential part of handling nested data. We employ the `DataFrame::flatten()` method. This is equivalent to a **`LATERAL`** `FLATTEN` operation in Snowflake SQL, which unboxes the elements of an array and replicates the other columns for each generated row [21].

B. Nested Queries

Nested queries are an intrinsic part of processing nested data. In JSONiq nested queries usually occur as part of a **`where`** or a `let` clause. Such queries usually unbox an array, filter and process its elements, and return results to the parent query. Listing 4 shows an example where a nested query is embedded into a `let` clause (Lines 2-5). Here, the Muon entry in each dataset object is unboxed (Line 3). The unboxed values are bound to the name `m`. Each muon then undergoes a filtering operation (Line 4). The passing muons are returned by the nested query (Line 5). At the end of the nested query, the semantics of JSONiq imply a transparent reaggregation of the returned values into an array for each parent object. The cardinality of a nested query's output is the same as its input. In other words, no objects can ever be fully removed by a nested query. Concretely, for Listing 4, this means that, for each dataset object (i.e., `event`), an array containing the object's muons that passed the predicate (i.e. reached the **`return`** clause) is produced. If none of the object's muons passed the predicate, an empty array should be returned.

We transparently deal with nested queries. We show this in Listing 4 and its trace in Figure 4. Upon entering a nested query, a unique ID is added to each row (`RowID`) to later reconstruct the table when exiting the nested query. The target structure (`Muon`) is flattened using `DataFrame::flatten()`. This creates a new column (`m`), and re-materializes the other columns of the original row (`RowID` and `Muon`). The nested query's logic is then applied. Tuples that pass the predicate are highlighted in gray. At the end of the nested query, the original table is reconstructed, with the nested query's results being packed in an array for each row. This is done by grouping using the row ID, using `Functions.array_agg()` on the **`return`**'s subexpression (`m`), and `Functions.any_value()` on the other columns. Auxiliary columns (e.g. `RowID`) are dropped.

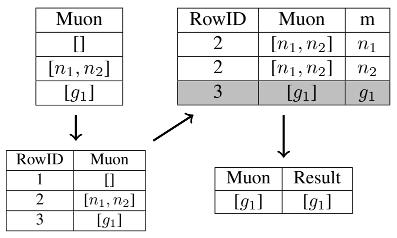
C. Erroneous Object Elimination

An important challenge that arises in nested queries is the erroneous elimination of dataset objects. The base pattern observed in nested queries involves several unboxing and filtering operations, ultimately followed by aggregation (§IV-B). In JSONiq, no parent object can be removed as a side-effect of a nested query. As Figure 4 shows, in Snowflake SQL, nested queries are handled through **`LATERAL`** `FLATTEN`. Subsequent subquery logic is then applied to such flattened top-level tables. This runs the risk of fully discarding objects when unboxing empty arrays or if all unboxed values of an object fail the nested query's predicates. When reaggregating at the end of the nested query, these objects no longer exist in the `DataFrame`. This conflicts with JSONiq's semantics of keeping all original objects and returning an empty array as the result of the subquery for such cases.

This scenario is exemplified in Figure 5a. Here, the object with row ID 1 fully disappears after the unboxing as its Muon array is empty. After filtering, both unboxed elements stemming from the row ID 2 object fail the predicate. When the result is reaggregated, only the object with row ID 3 remains. The nested query consequently produces incorrect results. To ensure nested queries work as expected, we propose and implement two approaches: (1) a flag column approach (§IV-C1) and (2) a **`JOIN`**-based approach (§IV-C2).

1) Flag Column Approach: In the flag column approach, we add a special column to the `DataFrame` passed into the nested query that indicates whether the row should be considered for aggregation in the nested query's **`return`** clause. We name this column `KEEP`. In addition to this, we ensure the unboxing operation is performed on empty arrays as well by setting `outer=`**`true`** in `DataFrame::flatten()`. This ensures that objects with empty arrays are retained.

Figure 5b shows how the flag column approach would work on a trace of the nested query in Listing 4. The `KEEP` column is added immediately after the unboxing. Rows stemming from objects with empty arrays have a **`false`** in their `KEEP` column; all other rows have **`true`**. When passed through **`where`** clauses, rows failing the predicate will have the `KEEP` column set to **`false`**. Rows that pass the predicate have their



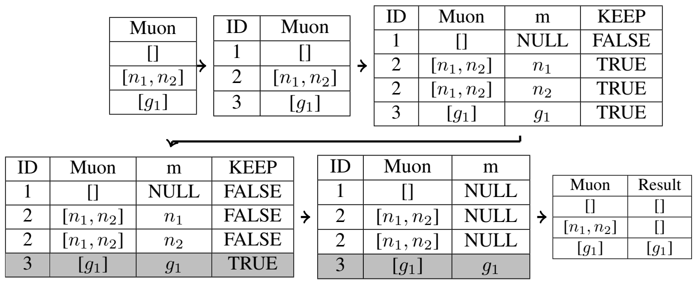
Muon
[]
$[n_1, n_2]$
$[g_1]$

RowID	Muon
1	[]
2	$[n_1, n_2]$
3	$[g_1]$

RowID	Muon	m
2	$[n_1, n_2]$	n_1
2	$[n_1, n_2]$	n_2
3	$[g_1]$	g_1

Muon	Result
$[g_1]$	$[g_1]$

(a) Erroneous object elimination example



Muon
[]
$[n_1, n_2]$
$[g_1]$

ID	Muon
1	[]
2	$[n_1, n_2]$
3	$[g_1]$

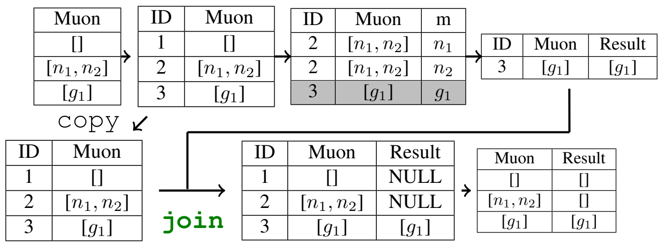
ID	Muon	m	KEEP
1	[]	NULL	FALSE
2	$[n_1, n_2]$	n_1	TRUE
2	$[n_1, n_2]$	n_2	TRUE
3	$[g_1]$	g_1	TRUE

ID	Muon	m	KEEP
1	[]	NULL	FALSE
2	$[n_1, n_2]$	n_1	FALSE
2	$[n_1, n_2]$	n_2	FALSE
3	$[g_1]$	g_1	TRUE

ID	Muon	m
1	[]	NULL
2	$[n_1, n_2]$	NULL
2	$[n_1, n_2]$	NULL
3	$[g_1]$	g_1

Muon	Result
[]	[]
$[n_1, n_2]$	[]
$[g_1]$	$[g_1]$

(b) Flag column approach



Muon
[]
$[n_1, n_2]$
$[g_1]$

ID	Muon
1	[]
2	$[n_1, n_2]$
3	$[g_1]$

ID	Muon	m
2	$[n_1, n_2]$	n_1
2	$[n_1, n_2]$	n_2
3	$[g_1]$	g_1

ID	Muon	Result
3	$[g_1]$	$[g_1]$

ID	Muon
1	[]
2	$[n_1, n_2]$
3	$[g_1]$

ID	Muon	Result
1	[]	NULL
2	$[n_1, n_2]$	NULL
3	$[g_1]$	$[g_1]$

Muon	Result
[]	[]
$[n_1, n_2]$	[]
$[g_1]$	$[g_1]$

(c) **`JOIN`**-based approach

Fig. 5: Erroneous object elimination example on Listing 4 and solutions. Rows that pass the predicate are highlighted in gray.

`KEEP` column unaffected. In the **`return`** clause, the rows where `KEEP` is **`false`** return a NULL, which is ignored in the array aggregation. This approach ensures that every object has at least one row stemming from it in the nested query, guaranteeing that the nested query produces the correct results.

While we do optimize **`where`** clauses to remove all failing rows, bar the last row stemming from an object, this technique has a slight data volume overhead.

*2) **`JOIN`**-based Approach:* In the **`JOIN`**-based approach, the `DataFrame` passed into the nested query is copied immediately after the row ID column is added. The nested query then freely manipulates the incoming `DataFrame` as described in Section 4, eliminating objects with empty arrays during unboxing operations and rows that get filtered in **`where`** clauses. The nested query produces its partial result, which gets joined with the copy of the `DataFrame` on the row ID column. The two tables are connected using a left outer join, where the left table is the copy `DataFrame`. **`NULL`** values in the result are replaced by empty arrays to ensure the nested query semantics are maintained. Figure 5c provides an example of how this would work for the subquery in Listing 4.

The benefit of this approach is that it proactively eliminates rows and does not require flag columns. The downside is the **`JOIN`** operation, which is fast in Snowflake [21] but can become costly for large tables. This approach can outperform the flag column approach (§IV-C1) for subqueries with many unboxing operations and filters.

D. Complex Non-FLWOR Iterators

Non-FLWOR iterators can return or modify `DataFrame` objects directly rather than returning `Column` objects to their parent iterator (§III-B1). This usually happens if non-FLWOR iterators are ancestors of a nested query. A frequent example of this is syntactic sugar expressions like **`where exists`**`(<subquery>)`. Here, the non-FLWOR iterator **`exists`** hosts a nested query that returns an updated `DataFrame`. For such cases, we take special care to orchestrate and maintain the original query semantics, which often involves decomposing the syntactic sugar expression into its fundamental clauses and subexpressions.

E. Limitations

Our work covers the major impedance mismatches between nested data querying and SQL. We however do not yet support recursive functions. We choose to not implement this feature for now as it is not often used in practice. In addition, the practitioner is tasked with choosing between the two solutions for erroneous object elimination (§IV-C). We've observed that the flag column approach fares well in most cases while the **`JOIN`**-based approach works best if there are numerous unboxing operations and **`where`** clauses in the nested queries. We plan to introduce an optimizer that can automatically decide between the two approaches based on the patterns observed in the nested query logic.

Our system takes a Snowflake-centric, data-format agnostic approach. We focus on deriving high performance and scalability from the translated query by focusing purely on the nested aspect of data and how it can be efficiently manipulated. This comes at the cost of two JSONiq semantics: (1) data ordering and (2) part of the JSONiq type system. JSONiq views the dataset as a stream, hence the output of a query should reflect the input order. Our translation processes the data in an arbitrary order thus compromising on (1). We could address this by adding an order number to each item to ensure the data gets processed in its original order. For (2), our system exclusively employs Snowflake types and discards special JSONiq types such as function items [10]. Some support for these could be added at the cost of query performance.

V. Evaluation

A. Methodology

We evaluate across several dimensions: (1) query translation time from JSONiq to SQL (§V-B) (2) query compilation time in Snowflake (§V-C) (3) query execution time in Snowflake (§V-D) (4) end-to-end time (i.e. the sum of (2) and (3) §V-D) (5) the number of bytes scanned (§V-E), (6) scalability of the queries in terms of the end-to-end time, as a function of the dataset size (§V-F), (7) the SSB relational benchmark [15]. For (4) and (6), we compare against AsterixDB and RumbleDB with a Spark backend, two state-of-the-art NoSQL systems for semi-structured data that use query languages tailored for such data (SQL++ and JSONiq).

For most experiments, we use the IRIS HEP ADL benchmark (§II-C). For experiment (1) the results are independent of the dataset size. For experiments (2) through (5) we use Scale Factor 1 (17 GiB of data with 54 million top-level objects). For experiment (6) we use 23 different scale factors: $\{2^{-16}$, 2^{-15}, …, 2^{5}, $2^{6}\}$, covering dataset sizes from 1000 events (approx. 300 kiB) to 3.42B events (approx. 1 TiB). For (7), we use four scale factors for the SSB data: $\{1, 10, 100, 1000\}$, ranging from approx. 1 GiB to approx. 1 TiB. For Snowflake, we use pre-loaded tables; for AsterixDB and RumbleDB we store the data in Parquet. Parquet has similar features to VARIANT: transparently columnarizing nested data (via Dremel's columnar storage representations for nested records [29]), using horizontal partitioning, and allowing the push-down of predicates for more efficient data pruning [30]. For a fair comparison, all systems use AWS infrastructure, and all data is stored in S3. We disable both the data and result cache in Snowflake. For experiments (1) - (6), data is staged in a single table; we choose to stage top-level entries in their own column (i.e., the multi-column schema). This applies to all systems. The alternative of storing the data as a single VARIANT column should not affect the presented Snowflake results and insights as the data's physical layout would be identical [22], but might negatively affect RumbleDB-on-Spark and AsterixDB as they rely on Parquet's shredding capabilities. For experiment (7), we use the official version of SSB. As baseline, we use RumbleDB-on-Spark, AsterixDB, and the handwritten Snowflake SQL queries publicly available in the official IRIS HEP ADL benchmark GitHub repository, which are written using the same query patterns used for the other relational databases in [7]. We use the reference queries for JSONiq (for our work and Rumble-on-Spark) and SQL++ (for AsterixDB), which are available in the official IRIS HEP ADL benchmark GitHub repository. We refer to the queries translated by our system as *Automatically Generated SQL*, and the handwritten Snowflake SQL references (which are executed on Snowflake) as *Handwritten SQL*. For Q6, we use the **`JOIN`**-based approach for nested queries (as it yields better performance); for the rest, we use the boolean flag-based approach (§IV-C). All translations are publicly available[3].

For each final value in experiment (1), we average a total of 100 runs with 10 warm-up runs. For each value in experiments (2) to (6), we average across a total of three runs with three additional warmup runs, while for (7), we average 20 runs. We impose a 10 minute query time limit for (6). For (1), we employ an AWS `z1d.xlarge` VM with 4 vCPUs and 32 GiB of memory. Our Snowflake deployment across experiments (2) to (7) consists of a single Standard edition LARGE virtual warehouse on an AWS backend. RumbleDB and AsterixDB are deployed on an `m5d.24xlarge` AWS VM (96 vCPUs, 384 GB RAM). For RumbleDB, we use AWS EMR [31] to ensure the Spark configuration is well-tuned.

[3]Translations repository: https://github.com/DanGraur/JSONiq-Snowflake

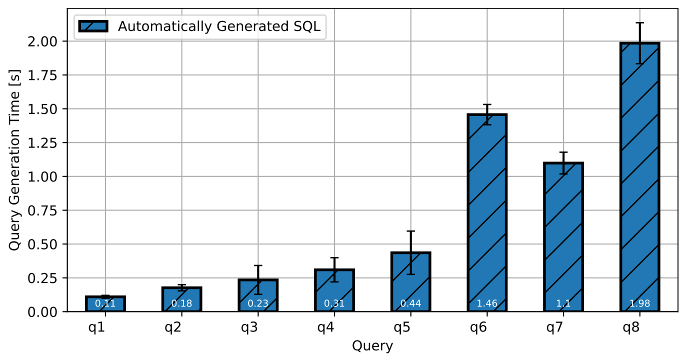


Fig. 6: Query translation time (JSONiq to SQL)

Type	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
FLWOR Iterators	14	14	17	21	24	61	24	72
Other Iterators	70	71	79	90	148	460	213	492
Total Iterators	84	85	96	111	172	521	237	564

TABLE II: The number of runtime iterators generated by RumbleDB for each query in the ADL benchmark.

B. Query Translation Time

Our results for the query translation times are shown in Figure 6. We count the number of iterators for each query (Table II) to understand the relationship between the iterators and the translation time. The translation time is directly proportional to the number of iterators, as more iterators imply more logic to go through to generate the SQL query.

The translation times are relatively low especially when compared to the query runtimes at larger scales and the benefits they can bring to developer productivity. This is our target use case: complex query logic that is not easily expressible in SQL executed on large-scale data. Nonetheless, we could translate faster by introducing a translation cache and by making RumbleDB's codebase leaner.

C. Compilation Time

The compilation time represents the time required by Snowflake to parse, optimize, and compile the SQL into an execution plan. It does not include any execution time. Figure 7 shows our results. For all queries except Q8, the compilation of the automatically generated queries is lower than the handwritten versions. This is likely because Snowpark expresses more verbose but conceptually simpler queries.

The compilation times are especially high for Q6 and Q8. This problem occurs both for the handwritten and the automatically translated queries, meaning the problem is not due to the translation. Q6 and Q8 are the most complex and compute-intensive queries in the benchmark [7]. Both generate dozens of triplets of particles from each dataset object and heavily manipulate them by applying complex HEP formulas that access many fields within nested structures and combine them to produce scalar results or generate new nested structures. The overhead of optimizing the complex access patterns is what likely leads to high compilation times. Optimizing this time is outside the scope of our work as it focuses on Snowflake internals but reducing it would prove beneficial to us.

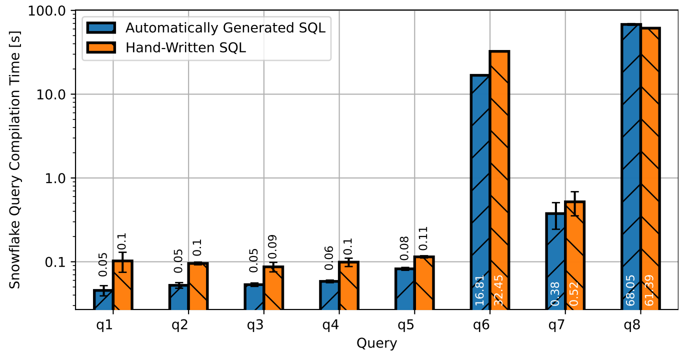


Fig. 7: Query compilation time in Snowflake

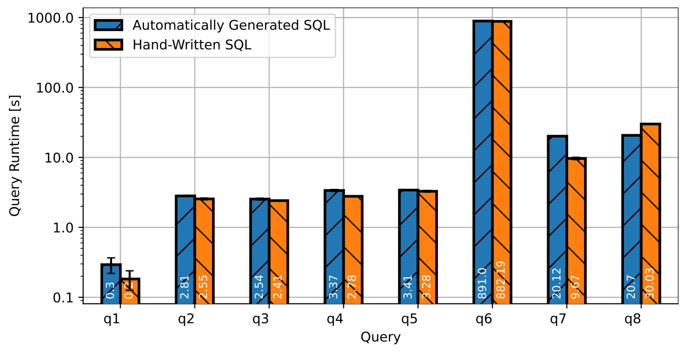


Fig. 8: Query execution time in Snowflake for SF1

D. Execution Time

We measure the Snowflake query execution time of the generated and handwritten queries on SF1 data (shown in Figure 8). The execution time corresponds to the execution of the query plan generated by the compilation phase. Apart from Q7, all the automatically translated queries are on par with the handwritten versions. The automatically generated version of Q7 is more expensive to run, as the original JSONiq query imposes four array unboxing-reaggregation pairs in the translation whereas the handwritten SQL version does this with four unboxing operations and two reaggregations. Moreover, the handwritten version uses the `BOOLAND_AGG`, which can apply a predicate and a conjunction in a single pass during aggregation. Snowpark API v1.9 does not expose this operation, hence the automatic translation must use less efficient approaches.

The translated version of Q8 is 1.5x faster than the handwritten version. This is because the handwritten query unboxes the Muon and Electron arrays into two separate tables, modifies the individual elements, and then reaggregates the modified elements into arrays. The resulting two tables undergo a **`UNION ALL`** immediately after to facilitate the concatenation of the two arrays into a single one. This is the approach taken for Q8 by some of the relational database implementations in [7]. The automatically translated version features SQL code where the result of each operation feeds into the **`FROM`** clause of the next operation. In the case of Q8, the results of nested

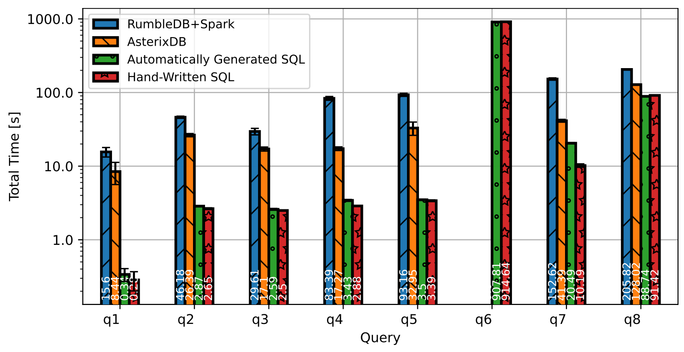


Fig. 9: End-to-end SF1 query runtime in evaluated systems

queries feed into each other, successively unboxing, filtering, and reboxing the partial result. This ultimately reduces the volume of processed data. Writing the latter version by hand is non-trivial. It requires several additional levels of nesting and care to avoid element duplication. This is encouraging and shows the practitioner can express query logic more naturally than in SQL for such use cases.

By adding the compilation and execution times, we obtain the total query runtime in Snowflake. We compare this to RumbleDB-on-Spark and AsterixDB. The results are shown in Figure 9. The takeaways are encouraging: our approach is on par with the handwritten queries for all cases bar Q7 where it is 2x worse. Moreover, the Snowflake deployments are generally one order of magnitude faster than AsterixDB and RumbleDB-on-Spark. AsterixDB and RumbleDB are both unable to finish executing Q6 within 10 minutes. As highlighted in Section III-A3, RumbleDB-on-Spark is slower than RumbleDB-on-Snowflake because of the limited support in Spark for JSONiq's operators. This implies UDF usage and partially lazy execution, which requires materialization and additional data movement, are adding noticeable runtime overheads.

E. Scanned Data

We track the volume of data scanned during execution (not plotted). In all cases, bar Q6, both versions scan roughly the same amount of bytes. In Q6, the translation ends up scanning more data due to the **JOIN** injected into the translation to deal with the object elimination problem. This triggers the re-scanning of the source table, roughly doubling the amount of data scanned when compared to the handwritten counterpart (7.4 GiB vs 3.9 GiB for SF1), which does not use **JOIN**.

F. Scalability

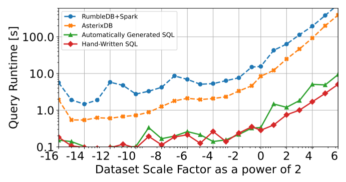
We evaluate the total time for all queries on 23 scale factors, ranging the dataset size between 1000 events (approx. 300 kiB) and 3.42B events (approx. 1 TiB) on Snowflake, AsterixDB, and RumbleDB-on-Spark. We impose a cut-off time of 10 minutes. Figure 10 shows our results. The automatically translated versions display very similar performance to the handwritten counterparts. Q1 displays noisy behavior at small scales due to its straightforward logic. Nonetheless, both versions have comparative performance. For Q6, the automatically translated version displays better behavior up to SF1, at which point the runtimes converge. The reason behind this is the **JOIN**-based approach used in Q6 for handling nested queries. This ensures the nested query can safely project away columns that are not used in the subquery, thus manipulating less data. In contrast, the handwritten queries manipulate a single table and run the risk of not being able to easily discard redundant columns. While joins are efficient, at higher scales they can become expensive, potentially explaining the time convergence at SF1. For Q8, both approaches are on par up to SF1. Up to SF1, the end-to-end time is dominated by the Snowflake compilation phase, taking upwards of 60s, regardless of dataset size. For larger scales, the **UNION ALL** operation in the handwritten version incurs large runtime costs (§V-D), losing in performance to the automatically generated version. When compared to AsterixDB and RumbleDB on Spark, Snowflake is consistently faster. Q8 is the only partial exception as Snowflake lags behind the other two systems up to SF1. As noted before, this is due to the high compilation time in Snowflake. At SF1, the handwritten version converges while the automatic translation outperforms the other approaches.

G. The Relational SSB Benchmark

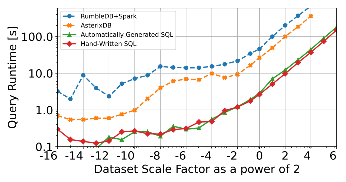
We evaluate our system on the SSB benchmark to show we can automatically translate relational queries expressed in JSONiq to SQL with identical performance as handwritten reference SQL implementations [15]. We run experiments on four scale factors: $\{1, 10, 100, 1000\}$ and on all of SSB's queries. Figure 11a shows a comparison between the performance of the automatically translated and the handwritten reference SSB SQL queries on Scale Factor 1000. In all cases, the total time (compilation and execution) is on par with the baseline. Figure 11b shows our results on query runtimes across the four tested scale factors. For legibility, we only show queries Q1.1, Q2.1, Q3.1, and Q4.1 but we confirm the same patterns can be observed for the rest of the queries. Across all tested Scale Factors, the two versions behave roughly the same. At lower scales, for queries Q2, Q3, and Q4, the automatically translated version can sometimes be slower than the baseline, as the JSONiq version must return a single object with several entries instead of several columns as the reference SQL. This introduces an additional `OBJECT_CONSTRUCT` in the query plan. The results show that JSONiq is a powerful query language capable of also expressing relational queries and that our system can support arbitrary queries spanning the nested and relational data paradigms with little overhead.

H. Takeaways

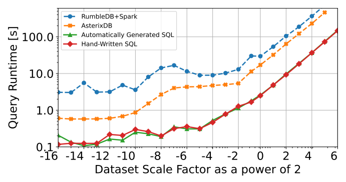
We show that we can build an effective query translation system that converts JSONiq queries to SQL using our Snowpark library for both nested (ADL) and relational (SSB) workloads. The translated queries offer on-par compilation and execution time to reference handwritten SQL queries. Both the automatic translations and reference implementations outperform state-of-the-art systems for nested data by one or two orders of magnitude. The results show it is possible



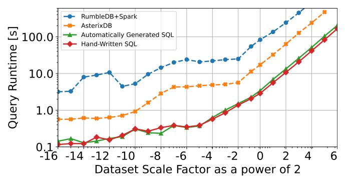
(a) Q1



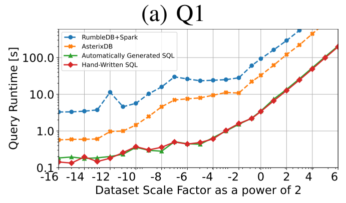
(b) Q2



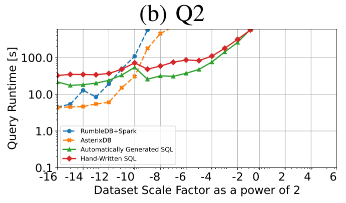
(c) Q3



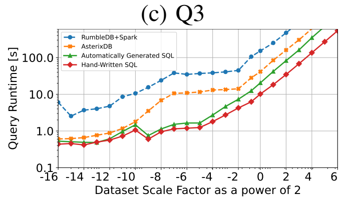
(d) Q4



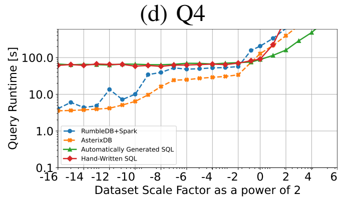
(e) Q5



(f) Q6

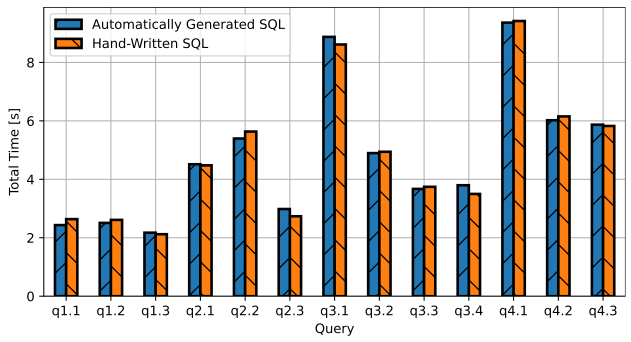


(g) Q7

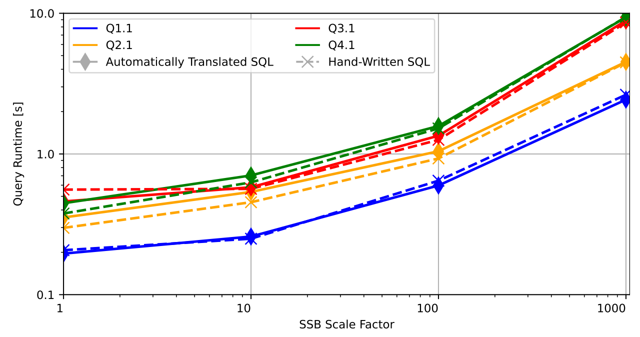


(h) Q8

Fig. 10: Query runtime for ADL queries on Scale Factors 2^{-16} (1000 events, ∼300 kiB) to 2^{6} (3.5B events, ∼1 TiB).



(a) For all queries at Scale Factor 1000.



(b) For subset of the queries on the tested Scale Factors.

Fig. 11: Full runtimes (compilation and execution) for the SSB queries.

to strive for both performance and usability for nested data querying, filling in the gap highlighted in [7].

VI. Related Work

Query Languages for Nested Data: The database community has developed numerous query languages targeted at nested data. To name a few: ArangoDB's AQL [32], SQL++ [33] (supported in Couchbase [34] and AsterixDB [35]), N1QL [36], and JAQL [37]. Our work focuses on JSONiq for two important reasons. (1) It has a mature, vendor-agnostic specification, designed to target heterogeneous data – which we plan to tackle in the future –, and (2) RumbleDB, the engine used for this study, is open-source and offers nearly complete support for JSONiq.

Query Engines for JSONiq: There have been a few solutions that implement support for JSONiq besides RumbleDB. Examples include Xidel [38], Zorba [39], and IBM WebSphere [40]. IBM WebSphere is vendor-dependent and, as a consequence, we do not pursue it for our study. As far as we are aware, Zorba is no longer maintained and Xidel is a command line tool written in Pascal, making an integration with Snowpark challenging. RumbleDB is actively maintained and provides a modern, extensible, iterator-based Java back-end that maps neatly to the Snowpark API.

Query Language Translation: The community has forwarded a large body of work around reusing RDBMSs to store semi-structured data and translate DSQLs [17], [41]. Previous work can be classified into two broad categories based on the type of data they target: (1) relational data [42]–[52] and (2) semi-structured data. We expand on the semi-structured data work as this is our target use case.

Previous work on storing nested data in an RDBMS and querying it using a DSQL has mostly focused on XML, XQuery, and XPath. Related contributions generally shred the nested data to map it over one or more flat relations and use `JOIN` operations to reconstruct it later at query time [17], [41]. To enable the shredding and reconstruction, one or more of three approaches is used [17]: (1) attaching an element ID to each shredded component, (2) attaching intervals to encode nestedness, and (3) storing the path to each shredded component. [53] maps semi-structured data to a relational mapping via the STORED language and uses a specialized SQL dialect to run queries against this data. [54] exploits the graph structure of an XML document to generate a schema where each edge and attribute are stored in their own table. [55] decomposes XML documents via vertical fragmentation and stores the root-to-leaf path and parent-child IDs in attribute tables. The authors propose a declarative algebra for XML document retrieval. XRel [56] uses a similar shredding strategy and focuses on translating a subset of XPath (i.e. XPathCore) to SQL via an eight-step algorithm. [57] uses a storage schema similar to XRel, but focuses on enabling ordering guarantees in the source data by embedding order values in the database.

Similar to XRel, it offers support for translating a class of XPath queries using several algorithms that enforce ordering information. [58] takes a different approach and proposes OQL, a query language for object-oriented data models. Work in this area has focused on converting path expressions to SQL [58]. Both [59] and [16] translate XQuery to SQL by using rule-based algorithms and composing fixed SQL templates associated with certain XQuery operators. [16] requires the addition of custom operators in the relational engine to obtain good performance. We take a different approach to many of these works by not explicitly shredding the data or requiring a schema. Our translation logic exploits intermediate representations of the original query to ultimately generate a single native SQL. This approach is similar to compiler theory but dissimilar to related work which generally translates DSQLs to SQL in one step.

Fundamentally, our approach contrasts with previous work, as we focus on a simpler data model: pure, format-agnostic, nested data. Related work has focused on XML, which is a document format rich in semantics (e.g. ability to mix text and node elements, providing ordering guarantees, node attributes, flexible traversal types, etc.). Research has focused extensively on XQuery and XPath, which are tightly bound to XML. Consequently, it has sought to abide by XML's semantics. As we focus purely on the nested aspect of data, we are not bound to a specific data model and can simplify our approach to strive for scalability and performance. We require no shredding or knowledge of the schema. We employ straightforward, well-known techniques from compiler theory to convert a query from JSONiq to SQL via several intermediate representations (AST, expression tree, iterator tree, and SQL code) and use Snowflake as an end-user to prove that it can be efficiently used for querying nested data without changing its internals. To the best of our knowledge, this is the first work that approaches the problem of translating a DSQL to SQL focusing purely on the nested aspect of data using well-known compiler techniques and without any need for shredding or schema.

VII. Discussion

A. Translation-Level Optimizations

We take a straightforward approach and convert JSONiq queries directly to SQL by implementing the behavior of iterators mapped to JSONiq expressions in Snowflake using Snowpark. We largely treat each iterator as independent and pass little cross-iterator information throughout the iterator tree. This approach leaves room for introducing translation-level optimizations that exploit contextual information and potentially synergize with the Snowflake query optimizer [60], [61]. For instance, our logic could identify common expressions in the JSONiq query and materialize them as temporary tables to facilitate the reuse of partial results across the query. Similarly, our work could benefit from identifying common query patterns that are often inefficiently expressed by practitioners in JSONiq or SQL and ensuring they are expressed using SQL constructs that efficiently exploit Snowflake. Q8's translation is an example of this, wherein a more efficient query pattern is non-obvious and non-trivial for a practitioner but easily translatable by a system like ours.

B. Integration with Snowflake

Our work shows that it is possible to reuse Snowflake as an end-user and achieve minimal overhead. We are considering porting this approach internally in Snowflake to derive further performance than what is achievable purely via the SQL interface. Through our work, we have identified features that could significantly improve runtime, such as better support for nested queries to avoid the use of **`LATERAL`** `FLATTEN` or the addition of array native functions like `ARRAY_FILTER`. We leave this for future work.

C. Generalizability to Other DSQLs

Our work has focused on translating JSONiq to Snowflake SQL via Snowpark. Bar recursive functions, it covers all major challenges in translating from JSONiq to SQL. The high-level approach used in this work of lowering a query to an iterator tree is well-known from database and compiler theory and can be applied to other DSQLs. Our implementation could potentially be used directly for other DSQLs via glue code, granted the language semantics are similar to JSONiq.

D. Generalizability to Other RDBMS

Our system is currently tailor-made for Snowflake and Snowflake SQL. The problem of translating between different SQL dialects is notoriously challenging. However, we believe it is possible to reuse our work and replace the logic of the iterator tree nodes to generate a Substrait execution plan [62] rather than building a Snowflake SQL query via Snowpark. In turn, the Substrait plan can be passed through various adapters to generate vendor-specific query plans [62].

VIII. Conclusion

We have presented our approach for translating JSONiq queries for nested and relational data to SQL using RumbleDB and Snowpark. Our approach takes a JSONiq query as input and generates a single native SQL query that fully encapsulates the original JSONiq logic and is completely executable in Snowflake. We exploit the Snowflake VARIANT type for storing and processing the nested data in a schema-oblivious way. We do not require any explicit shredding or schema from the source dataset. We have evaluated our work on the ADL (nested) and SSB (relational) benchmarks. We compare the performance of our approach against the benchmarks' handwritten reference query implementations and show that the automatic translations are on par with their handwritten counterparts across a wide range of data sizes. In the particular case of nested data, our system is an order of magnitude faster than state-of-the-art engines for semi-structured data.

References

[1] Oracle, "How to Store, Query, and Create JSON Documents in Oracle Database." https://blogs.oracle.com/sql/post/how-to-store-query-and-create-json-documents-in-oracle-database.

[2] Microsoft, "JSON data in SQL Server." https://learn.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver16.

[3] Postgres, "JSON Functions and Operators." https://www.postgresql.org/docs/current/functions-json.html.

[4] MySQL, "MySQL." https://dev.mysql.com/doc/refman/8.0/en/json-functions.html.

[5] AWS, "Querying JSON." https://docs.aws.amazon.com/athena/latest/ug/querying-JSON.html.

[6] BigQuery, "JSON Functions." https://cloud.google.com/bigquery/docs/reference/standard-sql/json_functions.

[7] D. Graur, I. Müller, M. Proffitt, G. Fourny, G. T. Watts, and G. Alonso, "Evaluating query languages and systems for high-energy physics data," *Proc. VLDB Endow.*, vol. 15, p. 154–168, oct 2021.

[8] J. Robie, M. Brantner, D. Florescu, G. Fourny, and T. Westmann, "JSONiq: XQuery for JSON," *JSON for XQuery*, pp. 63–72, 2012.

[9] D. Florescu and G. Fourny, "JSONiq: The History of a Query Language," *IEEE Internet Computing*, vol. 17, no. 5, pp. 86–90, 2013.

[10] "Jsoniq specification." https://www.jsoniq.org/docs/JSONiqExtensionToXQuery/html-single/index.html. Accessed: 2023-02-10.

[11] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, *et al.*, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 215–226, 2016.

[12] I. Müller, G. Fourny, S. Irimescu, C. B. Cikis, and G. Alonso, "Rumble: Data Independence for Large Messy Data Sets," *Proc. VLDB Endow.*, vol. 14, Dec. 2020.

[13] Snowflake, "Snowpark API documentation." https://docs.snowflake.com/en/developer-guide/snowpark/index.

[14] M. Proffitt, I. Müller, D. Graur, M. Adamec, P. David, E. Guiraud, and S. Binet, "iris-hep/adl-benchmarks-index: ADL Functionality Benchmarks Index v0.1," July 2021.

[15] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers 1*, pp. 237–252, Springer, 2009.

[16] D. DeHaan, D. Toman, M. P. Consens, and M. T. Özsu, "A comprehensive xquery to sql translation using dynamic interval encoding," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, (New York, NY, USA), p. 623–634, Association for Computing Machinery, 2003.

[17] R. Krishnamurthy, R. Kaushik, and J. F. Naughton, "Xml-to-sql query translation literature: The state of the art and open problems," in *Database and XML Technologies* (Z. Bellahsène, A. B. Chaudhri, E. Rahm, M. Rys, and R. Unland, eds.), (Berlin, Heidelberg), pp. 1–18, Springer Berlin Heidelberg, 2003.

[18] P. Buneman, "Semistructured data," in *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '97, (New York, NY, USA), p. 117–121, Association for Computing Machinery, 1997.

[19] D. Yu, "Efficient processing of almost-homogeneous semi-structured data," master thesis, ETH Zurich, Zurich, 2018.

[20] JSound, "JSound specification." http://www.jsound-spec.org/publish/en-US/JSound/2.0/html-single/JSound/index.html.

[21] Snowflake, "Snowflake Documentation." https://docs.snowflake.com/.

[22] Snowflake, "VARIANT Documentation." https://docs.snowflake.com/en/user-guide/semistructured-considerations.

[23] "BigQuery documentation." https://cloud.google.com/bigquery/docs. Accessed: 2023-03-30.

[24] D. Piparo, P. Canal, E. Guiraud, X. V. Pla, G. Ganis, G. Amadio, A. Naumann, and E. Tejedor, "Rdataframe: Easy parallel root analysis at 100 threads," in *EPJ Web of Conferences*, vol. 214, p. 06029, EDP Sciences, 2019.

[25] C. J. Date, *A Guide to the SQL Standard.* Addison-Wesley Longman Publishing Co., Inc., 1989.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pp. 15–28, 2012.

[27] S. Specification and T. P. P. C. TPC, "Tpc benchmark tm h," 1993.

[28] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, "Xquery 1.0: An xml query language," 2002.

[29] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[30] Apache, "Apache Parquet Documentation." https://parquet.apache.org/docs/.

[31] "AWS EMR documentation." http://docs.aws.amazon.com/emr/. Accessed: 2023-03-30.

[32] ArangoDB, "AQL documentation." https://www.arangodb.com/docs/stable/aql/.

[33] K. W. Ong, Y. Papakonstantinou, and R. Vernoux, "The sql++ query language: Configurable, unifying and semi-structured," *arXiv preprint arXiv:1405.3631*, 2014.

[34] M. A. Hubail, A. Alsuliman, M. Blow, M. Carey, D. Lychagin, I. Maxon, and T. Westmann, "Couchbase analytics: Noetl for scalable nosql data analysis," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2275–2286, 2019.

[35] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann, "AsterixDB: A Scalable, Open Source BDMS," *Proc. VLDB Endow.*, vol. 7, Oct. 2014.

[36] M. A. Hubail, A. Alsuliman, M. Blow, M. Carey, D. Lychagin, I. Maxon, and T. Westmann, "Couchbase analytics: Noetl for scalable nosql data analysis," *Proc. VLDB Endow.*, vol. 12, p. 2275–2286, aug 2019.

[37] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita, "Jaql: A scripting language for large scale semistructured data analysis," *Proceedings of the VLDB Endowment*, vol. 4, no. 12, pp. 1272–1283, 2011.

[38] B. van der Zander, "Xidel repository." https://github.com/benibela/xidel.

[39] Zorba, "Zorba documentation." http://www.zorba.io/documentation/latest.

[40] IBM, "IBM websphere documentation." https://www.ibm.com/docs/en/datapower-gateway/2018.4.

[41] M. N. Mami, D. Graux, H. Thakkar, S. Scerri, S. Auer, and J. Lehmann, "The query translation landscape: a survey," 2019.

[42] R. Krishnamurthy, R. Kaushik, and J. F. Naughton, "Efficient xml-to-sql query translation: Where to add the intelligence?," in *VLDB*, pp. 144–155, 2004.

[43] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk, "Querying xml views of relational data," in *VLDB*, vol. 1, pp. 261–270, 2001.

[44] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "Xml-ql: a query language for xml," 1998.

[45] M. Fernandez, A. Morishima, and D. Suciu, "Efficient evaluation of xml middle-ware queries," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pp. 103–114, 2001.

[46] M. Fernandez, W.-C. Tan, and D. Suciu, "Silkroute: Trading between relations and xml," *Computer Networks*, vol. 33, no. 1-6, pp. 723–745, 2000.

[47] I. Manolescu, D. Florescu, and D. Kossmann, "Answering xml queries on heterogeneous data sources.," in *Vldb*, vol. 1, pp. 241–250, 2001.

[48] A. Deutsch and V. Tannen, "Mars: A system for publishing xml from mixed and redundant storage," in *Proceedings 2003 VLDB Conference*, pp. 201–212, Elsevier, 2003.

[49] A. Deutsch, L. Popa, and V. Tannen, "Chase & backchase: A method for query optimization with materialized views and integrity constraints," *Technical Reports (CIS)*, p. 11, 2001.

[50] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen, "A chase too far?," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 273–284, 2000.

[51] R. Murthy and S. Banerjee, "Xml schemas in oracle xml db," in *Proceedings 2003 VLDB Conference*, pp. 1009–1018, Elsevier, 2003.

[52] G. Malcolm, *Programming Microsoft SQL Server 2000 with XML.* Microsoft Press, 2002.

[53] A. Deutsch, M. Fernandez, and D. Suciu, “Storing semistructured data with stored,” in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pp. 431–442, 1999.

[54] D. Florescu and D. Kossmann, “Storing and querying xml data using an rdmbs,” *IEEE data engineering bulletin*, vol. 22, p. 3, 1999.

[55] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas, “Efficient relational storage and retrieval of xml documents,” in *The World Wide Web and Databases: Third International Workshop WebDB 2000 Dallas, TX, USA, May 18–19, 2000 Selected Papers 3*, pp. 137–150, Springer, 2001.

[56] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura, “Xrel: a path-based approach to storage and retrieval of xml documents using relational databases,” *ACM Transactions on Internet Technology (TOIT)*, vol. 1, no. 1, pp. 110–141, 2001.

[57] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, “Storing and querying ordered xml using a relational database system,” in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 204–215, 2002.

[58] S. Cluet, “Designing oql: Allowing objects to be queried,” *Information systems*, vol. 23, no. 5, pp. 279–305, 1998.

[59] T. Grust, S. Sakr, and J. Teubner, “Xquery on sql hosts,” in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB ’04, p. 252–263, VLDB Endowment, 2004.

[60] J. S. Rellermeyer, S. Omranian Khorasani, D. Graur, and A. Parthasarathy, “The coming age of pervasive data processing,” in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 58–65, 2019.

[61] D. Graur, R. Bruno, and G. Alonso, “Specializing generic java data structures,” MPLR 2021, (New York, NY, USA), p. 45–53, Association for Computing Machinery, 2021.

[62] substrait io, “Substrait: Cross-language serialization for relational algebra.” https://github.com/substrait-io/substrait, 8 2021.