SQL++: We Can Finally Relax!

Michael Carey
University of California, Irvine
Irvine, CA, USA
mjcarey@ics.uci.edu

Don Chamberlin
IBM Research (Retired)
San Jose, CA, USA
chamberlin.don@gmail.com

Almann Goo
Amazon Web Services
Seattle, WA, USA
almann@amazon.com

Kian Win Ong
Meta
Menlo Park, CA, USA
kianwin@meta.com

Yannis Papakonstantinou[§]
Google Cloud
San Diego, CA, USA
yannispap@google.com

Chris Suver
Amazon.com
Seattle, WA, USA
csuver@amazon.com

Sitaram Vemulapalli
Couchbase, Inc.
Santa Clara, CA, USA
sitaram.vemulapalli@couchbase.com

Till Westmann
Couchbase, Inc.
Santa Clara, CA, USA
till@couchbase.com

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00438

***Abstract*—SQL is five decades old and has outlasted many programming and query languages that have come and gone during its lifetime. It was born shortly after the introduction of the relational model, and was designed for querying a flat and typed tabular world. Support for modern, flexible data in the SQL standard and in relational database systems has largely been approached via the addition of new column types (e.g. XML or JSON) together with functions to operate on them. It is time for a cleaner solution that retains the benefits that have allowed SQL to be so successful for so long.**

We describe SQL++, a SQL extension that relaxes SQL's strictness in terms of both object structure (flat → nested) and schema (mandatory → optional), along with a multi-party effort to agree on a core definition and syntax supportable by multiple vendors. SQL++ sees relational data as a subset of a more flexible object model and it sees collections of document data (e.g., JSON) as a natural and supportable relaxation as opposed to a "bolt on" addition via a SQL column type. We describe the core features of SQL++ and explain how its definition can accommodate flexible data, while staying true to SQL in situations where the target data is tabular and strongly typed.

***Index Terms*—semistructured data, query, JSON, SQL, NoSQL**

I. Introduction

Since the start of the cloud era, the database world has seen the emergence of multiple database systems that are designed to serve specific purposes. These database systems, often referred to as "NoSQL" systems, generally support semi-structured data models, and have adopted a wide variety of languages and/or APIs. The variations among these languages are often due to the fact that they were designed to serve specific access patterns very efficiently. For example, key-value stores focus on one specific access pattern.

Specialized query languages are often lacking in expressive power, and they are sometimes tightly-coupled to a specific data storage format. As these multiple database systems have grown and evolved, they have not been converging in either syntax or semantics. This lack of syntactic/semantic coherence may be diluting our field's resources and impeding the creation and adoption of new database applications.

[§]Work was substantially done at UCSD and AWS.

If a widely-recognized query language like SQL could be extended to handle semi-structured and other modern data formats well, then the skills, experience, and tools of the SQL community could be brought to bear on developing more modern applications. Broadening the scope of SQL itself in this way is the goal of the SQL++ language. SQL++ is a backward-compatible extension of SQL that is designed to handle semi-structured, nested, and schema-optional data. We believe that the time is ripe for the appearance of such a language.

SQL++ builds on initial research that was done at UC San Diego [1], [2]. The initial UCSD work showed how the query capabilities of eleven different NoSQL and NewSQL languages could be captured under a minor expansion of the SQL syntax, utilizing a handful of configuration modifiers to capture their occasional semantic differences. A user-friendly combination of those configuration modifiers became the *SQL++* dialect used by the Apache AsterixDB system (which originated at UC Irvine and UC Riverside) [3] and later by Couchbase Server originally under the name *N1QL* (for non-1NF query language) [4]–[6]. Concurrently, AWS adopted and further elaborated on a SQL-compatible version of UCSD SQL++ under the name *PartiQL*, and released a PartiQL open source reference implementation [7]. Amazon Redshift [8], Amazon QLDB [9], Amazon DynamoDB [10], Amazon S3/Glacier Select [11], AWS IoT TwinMaker [12] and various Amazon internal systems consequently adopted PartiQL. Inspired by a recent SIGMOD keynote talk [13], an effort was formed to bring the aforementioned dialects together under a single SQL++ definition, and we present this effort's results herein.

The effort to define a unified SQL++ definition has been based on the following tenets:

- *SQL compatibility*: SQL++ should facilitate adoption by maintaining compatibility with SQL. Existing SQL queries should continue to work, with identical syntax and semantics, in SQL query processors that are extended to provide SQL++. This avoids any need to rewrite existing SQL queries, and it makes it easy for developers and business intelligence tools to leverage SQL++.

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00438

- *First-class nested data*: The data model for SQL++ treats nested data as a fundamental part of the data abstraction. Consequently, the SQL++ query language provides syntax and semantics that comprehensively and accurately access, query, and construct nested data, while naturally composing with the standard features of SQL.
- *Optional schema and query stability*: SQL++ does not require a predefined schema over a query's target input. However, SQL compatibilities that pertain to schemas should also be respected. Technically, the result of a working query should not change if a schema is imposed on existing data, so long as the underlying data itself remains the same.
- *Composability*: SQL++ should have a minimum number of extensions over SQL. The extensions should be easy to understand, lend themselves to efficient implementation, and compose well with one another and with SQL itself, much as functions in functional programming languages do. This enables intuitive filtering, joining, aggregation, and windowing on a combination of structured, semistructured, and nested data.
- *Format independence*: SQL++'s syntax and semantics should not be not tied to a particular data format. A query should be written identically across underlying data in any of today's many nested and/or semistructured formats: JSON [14], Parquet, Avro, ORC, CSV, CBOR [15], Ion [16], and others. Queries should operate on a comprehensive logical type system that maps to diverse underlying formats.

While these tenets are generally orthogonal to each other, SQL compatibility and composability are occasionally in conflict. Fundamentally, this is due to the fact that in ANSI SQL, we cannot fully model each of the `FROM`, `WHERE`, `GROUP BY`, `SELECT` and other clauses as being fully composable operators that simply feed inputs and outputs to each other. Aggregate functions and handling of nested query results are SQL's two most prominent violations of functional composability. For the purpose of reconciling compatibility and composability SQL++ takes two measures: First, we define a *SQL++ Core*, consisting of fully composable operators. Then SQL itself is defined as "syntactic sugar" rewritings over the SQL++ Core. Moving the needle between compatibility and composability turns into choosing whether to incorporate or not these rewritings. We include a *SQL compatibility* flag in SQL++ whose setting can be toggled between prioritizing composability or prioritizing SQL compatibility. Another advantage of relying on a functional SQL++ Core is that the semantics of SQL++ and of SQL become shorter and more concise, despite adding more functionality to SQL.

The composability of SQL++ can also be perceived as a *relaxation of SQL restrictions*, which leads to achieving more functionality (i.e., querying semistructured data) mostly by removing constraints rather than by adding features.

SQL++ relaxes a number of aspects of SQL:

1) While SQL collections (*a.k.a.* relations or tables) consist of homogeneous tuples [17], SQL++ allows collections to be anything composed by arrays, multisets, structs, and scalars, without requiring homogeneity. Schema is optional in SQL++.
2) Typing rules are dynamically checked in SQL++, with the possibility of static type checking when the optional schema is present. In the interest of processing flexibly semistructured data, SQL++ allows processing to continue even when dynamic type errors happen (see Section IV) so that the processing of "healthy" data can proceed, while a convenient signal, which most often leads to data exclusion, happens for the data that led to typing errors. To support applications that want to catch type errors early and stop processing when they happen, SQL++ also offers a stop-on-error mode.
3) Unlike SQL's `FROM` clause variables, which can only bind to tuples, the `FROM` clause variables in SQL++ can bind to any type of SQL++ data. For example, `FROM` variables can bind to tuples, or to arrays, or to scalars, or to any combination thereof.
4) SQL++ is fully composable in the sense that subqueries can appear *anywhere*, potentially creating nested results when they appear in the `SELECT` clause.
5) The groups created by the SQL++ `GROUP BY` clause are directly usable in nested queries – as opposed to SQL's approach where they may only participate in aggregate functions in very limited and particular ways. Indeed, the SQL++ approach ends up explaining SQL's grouping and aggregation in a simpler, direct way.

In the interest of inspiring further community collaboration on SQL++, in 2019 AWS released its PartiQL open source reference implementation [7], while Apache AsterixDB has offered open source SQL++ support since 2017 [18].

The remainder of this paper discusses the key aspects of SQL++, which can be mostly conceived as relaxations of SQL. Section II describes the data model underlying SQL++. Section III talks about SQL++'s support for querying nested data. Section IV describes how SQL++ deals with the world without schemas. Section V discusses how SQL++ enables the creation of nested data and how doing so relates to aggregation both in SQL and SQL++. Section VI discusses how SQL++ can turn attribute names into data and vice versa, by unpivoting and pivoting. Section VII briefly touches on prior related language work. Finally, Section VIII provides a recap of the paper and a call to arms for our community and industry.

II. Data Model

SQL++ generalizes, mostly by relaxation, the SQL data model. A SQL++ database contains one or more SQL++ *named values*. A *name* is an identifier, such as a collection name, that is associated with a SQL++ value. It may also be a dotted/namespaced identifier, such as `hr.emp_nest_tuples`, that could reflect the database/table hierarchy of a MySQL database or the schema/table hierarchy of a Postgres database.

For the examples in this paper, we will be using an object notation using SQL literals that is similar to a data

format such as JSON, CBOR, or Ion. This object notation is meant to represent data that is self-describing and thus to be interpreted without a schema. We have adapted examples from the PartiQL tutorial [7] here to aid in explaining SQL++.

```
{{
  {
    'id': 3,
    'name': 'Bob Smith',
    'title': null,
    'projects': [
      {'name': 'Serverless Query'},
      {'name': 'OLAP Security'},
      {'name': 'OLTP Security'}
    ]
  },
  {
    'id': 4,
    'name': 'Susan Smith',
    'title': 'Manager',
    'projects': []
  },
  {
    'id': 6,
    'name': 'Jane Smith',
    'title': 'Engineer',
    'projects': [
      {'name': 'OLTP Security'}
    ]
  }
}}
```

Listing 1. An example SQL++ collection named `hr.emp_nest_tuples`.

A value can be *absent*, *scalar*, *tuple*, *collection*, or any composition thereof. Further subtyping applies to these types. Collections may be arrays, denoted by `[ ... ]`, or bags (i.e., multisets), denoted by {{ ... }} (or << ... >>). The scalars are the SQL scalar types. The full extent of the SQL type system coverage is left up to SQL++ implementations. SQL relies on schema to dictate what the data types in its values are, but SQL++ relaxes this reliance to allow data to be self-describing.

A tuple is a set of attribute name/value pairs, where each name is a string (as in SQL). A tuple in the SQL++ data model is unordered. Notice the contrast with schemaful SQL: a conventional SQL tuple is an ordered tuple since the schema dictates the order of the attributes and certain SQL operations may use this order. Also, unlike SQL, the SQL++ data model allows for the possibility of duplicate attribute names. This is in the interest of compatibility with non-strict data in formats such as JSON, Ion, and CBOR. However, SQL++ does not *encourage* duplicate attribute names. In particular, navigation into tuples via the conventional dot notation (Section III) can lead to nonreproducable results in the presence of duplicate attribute names.

SQL++ offers two kinds of absent values: `NULL` and `MISSING` are both available for representing missing information. The motivation is as follows: Unlike SQL, where a query that refers to a non-existent attribute name is expected to fail during compilation, in semi-structured data one expects a query to be *permissive* and keep operating in many situations where SQL would fail. One such situation is when some of the tuples do not define an attribute that a query's path mentions. Another situation is when functions input wrongly-typed arguments. Addressing both cases, SQL++ contains the special value `MISSING`, which is the path result in cases where navigation fails to bind to any information or where a function fails due to missing or wrongly typed inputs. The distinction between `MISSING` and `NULL` enables retention of the original distinction between a missing attribute and a present but null-valued attribute. The utility of `MISSING` (as opposed to just having `NULL`) will become further apparent when navigation into semi-structured data and construction of semi-structured results is discussed below, where we will see that the value `MISSING` may not itself appear as an attribute's value.

III. Accessing Nested Data

The data model of SQL-92 only has tables with tuples that contain scalar values (the "normal form" of [17]). A key feature of many modern formats is nested data. That is, modern data can have attributes whose values may themselves be tables (i.e., collections of homogenous tuples of scalars), or may be arrays of scalars, or arrays of arrays and many other combinations.

In Listing 1, the value of the `projects` attribute is an array, which happens to be an array of tuples. The following SQL++ query finds the names of employees who work on projects that contain the string `'security'` and outputs them along with the name of the security project.

```
SELECT e.name AS emp_name,
       p.name AS proj_name
FROM hr.emp_nest_tuples AS e,
     e.projects AS p
WHERE p.name LIKE '%Security%'
```

Listing 2. A SQL++ query accessing nested tuples.

This query effectively joins each employee tuple with the project tuples that are nested inside it. This feature requires no syntactic extensions to SQL. It simply allows expressions in the `FROM` clause to refer to variables that are defined earlier in the `FROM` clause (in this case, the expression `e.projects` refers to the variable `e` defined earlier). This feature, which relaxes a constraint of SQL, is called "left-correlation."

Once we allow left-correlation, the query's semantics are similar to SQL. The alias `e` (also called a variable in SQL++) gets bound to each employee, in turn. For each employee, the variable `p` gets bound to each project of the employee, in turn. Thus the query's meaning, much as for cross products or joins in SQL, is illustrated by Pseudocode 1.

Notice that the query involves a variable that is ranging over a nested collection (`p` in the example), along with a variable (`e` in the example) that is ranging over a "table", as standard SQL aliases do. All variables, no matter what they range over, can be used as needed in the `FROM`, `WHERE`, and `SELECT` clauses, as we will see in the examples that follow.

Pseudocode 1 SQL++ query with nested tuples.

```
1: for each employee TUPLE e ∈ hr.emp_nest_tuples do
2:     for each project TUPLE p ∈ e.projects do
3:         if p.name LIKE '%Security%' then
4:             output TUPLE
5:             'emp_name' ↦ e.name
6:             'proj_name' ↦ p.name
```

The explicit denotation of variables is essential to SQL++ Core – unlike SQL, wherein one might simply write `name` to imply `e.name`. In SQL, the presence of schema allows this form of static disambiguation, but since schema is optional in SQL++ a query has to explicitly call out the variables. Nevertheless, if schema is available, then SQL++ also allows expressions that are disambiguated using the schema. Formally, disambiguation results in the rewriting of the user-provided SQL++ query into a SQL++ Core query that explicitly denotes the variables that were omitted.

A. Aliases may bind to any value – not just tuples

The previous example has illustrated nested attributes that were arrays of tuples and variables (aliases) that were ranging over the nested tuples. It need not be the case, however, that the nested attributes are collections of tuples. They may just as well be arrays of scalars, arrays of arrays, or any combination of data that one can create by composing scalars, tuples and arrays. The user need not learn a different set of query language features for each case. The unnesting feature, which we have already seen, is sufficient since variables may bind to any type of value. For example, we could modify the data from Listing 1 so that `projects` are arrays of scalars, as illustrated below in Listing 3.

```
{{
  {
    'id': 3,
    'name': 'Bob Smith',
    'title': null,
    'projects': [
      'Serverless Querying',
      'OLAP Security',
      'OLTP Security'
    ]
  },
  ...
}}
```

Listing 3. Collection `hr.emp_nest_scalars`: The arrays of tuples of `hr.emp_nest_tuples` in Listing 1 are replaced with arrays of scalars.

```
SELECT e.name AS emp_name,
       p AS proj_name
FROM hr.emp_nest_scalars AS e,
     e.projects AS p
WHERE p LIKE '%Security%'
```

Listing 4. A SQL++ query with nested scalars.

The query in Listing 4, again, finds the names of employees who work on projects that contain the string `'security'` and outputs them along with the name of the security project. Notice that the variable `p` ranges (again) over the content of `e.projects`. In this case, though, since `e.projects` contains strings (as opposed to tuples), the variable `p` binds each time to a project name string.

Pseudocode 2 demonstrates that this query is semantically very similar to the earlier query.

Pseudocode 2 SQL++ query with nested scalars.

```
1: for each employee TUPLE e ∈ hr.emp_nest_scalars do
2:     ▷ Notice that p is not a tuple. ◁
3:     for each project STRING p ∈ e.projects do
4:         if p LIKE '%Security%' then
5:             output TUPLE
6:             'emp_name' ↦ e.name
7:             'proj_name' ↦ p
```

To wrap up, the above example exhibits a key relaxation (and, thus, a generalization) of the SQL++ semantics as compared to SQL semantics. The tuple calculus-based theoretical underpinnings of SQL define the semantics of its `FROM` clause as delivering *bindings* of the aliases into tuples. In contrast, SQL++ treats the `FROM` clause as a function that delivers bindings of the variables to arbitrarily typed values. In our example, the `FROM` clause produced bindings of the form $\langle \texttt{e} : \ldots, \texttt{p} : \ldots \rangle$ and, as we saw, the `p` bindings were to strings.

IV. Absence of Schema and Semi-structured Data

Many data formats do not require a schema that describes the data - that is, they involve *schemaless data*. In such cases it is possible to have various "heterogeneities" in the data:

- One tuple may have an attribute x while another tuple may not have this attribute.
- In one tuple of a collection of tuples an attribute x may be of one type, e.g., string, while in another tuple of the same collection the same attribute x may be of a different type – e.g, array.
- The elements of a collection (be it a bag or an array) can be heterogeneous (not have the same type). For example, the first element may be a string, the second one may be an integer, and the third one may be an array. While we do not recommend such data modeling practices, they can arise as a result of requirements evolution or due to legacy. For example, it turns out that converters of XML into JSON sometimes create such heterogeneities.
- Generally, any composition is possible, as we can bundle heterogeneous elements together in arrays and bags.

Heterogeneity is not particular to schemaless data. NewSQL schemas may allow for heterogeneity in the types of the data. For example, one of the Hive data types is the union type [19], which allows a value to belong to any one of a list of types, as in the Hive schema of Listing 5 example where the `projects` attribute may be either a string or an array of strings.

```
CREATE TABLE emp_mixed (
  id       INT,
  name     STRING,
  title    STRING,
  projects UNIONTYPE<STRING, ARRAY<STRING>>
);
```

Listing 5. A Hive data definition for a SQL++ table with a mixed attribute.

Thus, we see that data may have heterogeneities, regardless of whether it is described by a schema or not. SQL++ tackles heterogeneous data in ways that we will see in the next few use cases and feature descriptions.

A. Missing Attributes

Consider the collection of tuples named `emp_null` of Listing 6. Bob Smith has no title and, as is typical in SQL, his lack of title is modeled with the null value.

```
{{
 {'id': 3,
  'name': 'Bob Smith',
  'title': null },
 {'id': 4,
  'name': 'Susan Smith'
  'title': 'Manager' },
 {'id': 6,
  'name': 'Jane Smith',
  'title': 'Engineer'}
}}
```

Listing 6. Example `hr.empl_null` "table" with `NULL` values.

Nowadays, many semi-structured formats allow their users to represent "missing" information in two ways: The first way is by use of a `NULL` datum. The second kind is the plain absence of the attribute from the tuple. That is, we could represent the fact that Bob Smith has no title by simply having no title attribute in the 'Bob Smith' tuple:

```
{{
 {'id': 3,
  'name': 'Bob Smith'}, -- no title
 {'id': 4,
  'name': 'Susan Smith'
  'title': 'Manager' },
 {'id': 6,
  'name': 'Jane Smith',
  'title': 'Engineer'}
}}
```

Listing 7. Example `hr.emp_missing` of a "table" omitting an attribute value.

SQL++ does not adopt a position about when to use `NULL` versus when to use "missing". Myriads of existing data already use one of the two or both. However, SQL++ enables queries to distinguish between `NULL` and missing values, and it also enables query results that have nulls and missing values. Indeed, SQL++ makes it very easy to propagate source data nulls through as query result nulls and source data missing attributes through as query result missing attributes using the special value `MISSING`.

B. MISSING as a Value

Consider the SQL++ query of Listing 8, which happens to also be a valid SQL query:

```
SELECT e.id,
       e.name AS emp_name,
       e.title AS title
FROM hr.emp_missing AS e
WHERE e.title = 'Manager'
```

Listing 8. A SQL++ query referring to a potentially missing attribute.

What will happen when this query processes the Bob Smith tuple, which has no title? The first step to answering this question is understanding the result of the path `e.title` when the alias (variable) `e` binds to the tuple `{'id': 3, 'name': 'Bob Smith'}`. Or in more basic terms, what is the result of the expression `{'id': 3, 'name': 'Bob Smith'}.title`? The SQL++ answer is that it is the special value `MISSING`.

Generally, `MISSING` values are produced in three cases:

1) Navigation into a missing attribute. For example, `{'id': 3, 'name': 'Bob Smith'}.title` returns `MISSING`.
2) A function or operator is evaluated over arguments of the wrong types. Essentially, the flexible mode of SQL++ prefers to return `MISSING` instead of a dynamic type error when evaluating expressions such as `2 * 'some string'`.
3) Whenever a function or operator has a `MISSING` input, it returns a `MISSING` result. This ensures that `MISSING` values created by Cases 1 and 2 can be easily propagated through a series of transformations. However, in SQL-compatibility mode, this rule has one exception: if an SQL expression, given a null input, would return a non-null result, the same expression in SQL++ returns the same result when given a `MISSING` input. For example, the expression `COALESCE(MISSING, 2)` will return `2` because this is what SQL's `COALESCE(NULL, 2)` will do.

The next question is how to utilize `MISSING` values in result construction. For starters, in SQL++'s SQL compatibility mode, the user can treat them as identical to `NULL` if he or she doesn't care about the distinction between null and missing values. More precisely, in SQL compatibility mode, SQL++ delivers the following guarantee: Given a working SQL query q over a collection d that has null values and a collection d' where some nulls have been replaced with missing attributes, the SQL++ query q will deliver the same result $q(d')$ as the SQL result $q(d)$, except that some attributes that would have null values in $q(d)$ will be simply missing in $q(d')$. From a SQL compatibility point of view, this difference is immaterial to the SQL user who doesn't care about the difference between the two. However, the null-missing distinction enables SQL++ to expand into (a) flexibly executing queries that would not work in SQL and (b) easily propagating `MISSING` from inputs to outputs. In the earlier example, when the query outputs the Bob Smith tuple, the expression `e.title` will evaluate to `MISSING` and the output tuple will not have a `title` attribute.

```
SELECT e.id,
       e.name AS emp_name,
       CASE WHEN e.title LIKE 'Chief %'
            THEN "Executive"
            ELSE "Worker"
       END AS category
FROM hr.emp_missing AS e
```

Listing 9. SQL++ query operating on MISSING.

Thanks to the easy propagation of MISSING, the same treatment of MISSING would happen if, say, we had a query that sorts employees into categories based on their titles, as shown in Listing 9.

Again, for Bob Smith, the CASE WHEN e.title ... END will evaluate to CASE WHEN MISSING ... END, which will in turn evaluate to MISSING. (Note that in JDBC/ODBC communication, with a schemaful result, the MISSING will be communicated as NULL for communication compatibility purposes.)

V. Result Construction, Nesting, and Grouping

SQL++ allows for queries that create nested results as well as for queries that create any type of result collection – not just collections of tuples of scalars. At the SQL++ Core level, its power comes from relaxing the SQL SELECT clause into the SQL++ SELECT VALUE clause and exposing the groups created by SQL's GROUP BY clause.

A. Creating Collections of Any Value

The SQL++ Core grounds its ability to construct results in its SELECT VALUE clause, which provides the power to construct collections of any type of data. Consequently, the SELECT clause of SQL can be explained as being syntactic sugar over SELECT VALUE. For example, the query shown in Listing 10 outputs each tuple of hr.emp_nest_scalars (Listing 3), except that instead of keeping all of their projects, each tuple has only the security projects of each employee. Notice how SELECT VALUE p is being used in the query. The result of this query is shown in Listing 11.

A SELECT VALUE <expression> query (or subquery, as in this example) returns a collection of whatever the <expression> evaluates to. Thus SQL's SELECT can be rewritten into the SQL++ Core SELECT VALUE as follows:

SELECT e_1 AS a_1, ..., e_n AS a_n FROM ...
is equivalent to
SELECT VALUE { a_1:e_1, ..., a_n:e_n } FROM ...

```
SELECT e.id AS id,
       e.name AS emp_name,
       e.title AS emp_title,
       ( SELECT VALUE p
         FROM e.projects AS p
         WHERE p LIKE '%Security%'
       ) AS security_proj
FROM hr.emp_nest_scalars AS e
```

Listing 10. SQL++ query projecting a nested value.

```
{{
  {
    'id': 3,
    'name': 'Bob Smith',
    'title': null,
    'security_proj': {{
      'OLAP Security',
      'OLTP Security'
    }}
  },
  {
    'id': 4,
    'name': 'Susan Smith',
    'title': 'Manager',
    'security_proj': {{}}
  },
  {
    'id': 6,
    'name': 'Jane Smith',
    'title': 'Engineer',
    'security_proj': {{
      'OLAP Security'
    }}
  }
}}
```

Listing 11. Example result from nested SELECT VALUE.

However, when a SQL SELECT appears as a subquery, SQL compatibility requires that it not be treated simply as being a shorthand of SELECT VALUE. Rather, the context of the subquery designates whether the subquery's result should be coerced into a scalar value (e.g., when 5 = <subquery>), coerced into a collection of scalars (e.g., when 5 IN <subquery>), etc. None of this implicit "magic" applies to SELECT VALUE, which always produces a collection that will not be coerced. Indeed, SQL++, when not operating in SQL compatibility mode, always treats SELECT as a shorthand for SELECT VALUE, thus delivering the composability and simplicity of functional programming languages and enabling a proper treatment of nested data and nested results.

B. GROUP BY and GROUP AS

Another pattern for creating nested results in SQL++ is via the GROUP AS extension to SQL's GROUP BY clause. This pattern is more efficient and more intuitive than nested SELECT VALUE queries when the required nesting is not based on the nesting of the input. (To clarify, the example in Section V-A is one where the nesting in the output follows the nesting of the input, so the intuitive solution does not require GROUP BY.)

The SQL++ GROUP AS extension generalizes the SQL GROUP BY clause by making the formulated groups (and their content) available in their entirety to a SQL++ query's SELECT and HAVING clauses. This is in contrast to SQL's GROUP BY, where the SELECT and HAVING clauses can have aggregate functions over grouped columns but they cannot access the individual values contained in the grouped columns (due to the fact that nested data is not available in the data model that underlies SQL).

To better understand the workings of GROUP BY ... GROUP AS, it is best to think of a SQL++ query as being a pipeline of clauses, starting with the FROM, continuing with the optional

WHERE, proceeding to the optional GROUP BY, and then the optional HAVING, and finishing with the SELECT clause. Each clause is a function that inputs data and outputs data. In that sense, in the upcoming example, the GROUP BY ... GROUP AS clause is a function that inputs the result of the FROM and outputs its result to the SELECT. (SQL++ also supports the post-SELECT clauses of SQL, e.g., ORDER BY, LIMIT, and OFFSET.)

In keeping with this model of a query-block as a pipeline of functional clauses, SQL++ allows the SELECT clause to be written either at the beginning of the query-block (as in SQL) or at the end of the block (which is both more consistent with the execution model and much clearer regarding where variables mentioned in the SELECT clause are coming from). This flexibility can be considered as another example of where SQL++ relaxes a constraint of SQL.

The query in Listing 12 inverts the hierarchy of employees with nested projects. It produces a list of the security projects (after conversion to lower case) and it includes nested lists of the names of employees that work on each project. Notice that in this query, as an aside, the SELECT clauses appear at the ends of their respective query-blocks rather than at the start. (Either placement is fine in SQL++.) The nested result created by this query follows in Listing 13.

```
FROM hr.emp_nest_scalars AS e, e.projects AS p
WHERE p LIKE '%Security%'
GROUP BY LOWER(p) AS p GROUP AS g
SELECT p AS proj_name,
   (FROM g AS v
    SELECT VALUE v.e.name) AS employees
```

Listing 12. SQL++ query with grouping.

```
{{
  {
    'proj_name': 'OLTP Security',
    'employees': {{
      'Bob Smith'
    }}
  },
  {
    'proj_name': 'OLAP Security',
    'employees': {{
      'Bob Smith',
      'Jane Smith'
    }}
  }
}}
```

Listing 13. Example result from query using GROUP AS.

Let's examine what's going on with GROUP AS here. In this query, the FROM clause delivers a collection of bindings for e and p. The GROUP BY ... GROUP AS ... then produces a multiset of objects that has one field for each value of the group-by expression (i.e., for each security project LOWER(p) AS p) and a second field g whose value (in each row) is the collection of employee/project e/p tuples that belong to that group. Thus the GROUP BY ... GROUP AS ... output is the collection of p/g bindings illustrated in Listing 14.

```
-- first binding
p: 'olap security'
g: {{
    { e: { 'id': 3, 'name': 'Bob Smith', ... },
      p: 'OLAP Security'
    },
    { e: { 'id': 6, 'name': 'Jane Smith', ... },
      p: 'OLAP Security'
    }
}}
-- second binding
p: 'oltp security'
g: {{
    { e: { 'id': 3, 'name': 'Bob Smith', ...},
      p: 'OLTP Security'
    }
}}
```

Listing 14. Output of the GROUP BY ... GROUP AS ... clause.

Finally the SELECT clause takes the output produced by the GROUP BY ... GROUP AS ... clause as its input and outputs the final query result.

Before leaving this section it is important to note that SQL has additional analytical features such as CUBE, ROLLUP, and GROUPING SETS for grouped aggregation as well as window functions (i.e., OVER) for more advanced analytics. These features are wholly compatible with SQL++ and then become able to operate on and produce nested and heterogeneous data (e.g., see [20]). Their compatibility stems from SQL++'s defining equality identically to SQL in the exclusive presence of scalars and NULL. (And for each SQL feature that does not error on NULL, the feature will also work with MISSING.)

C. Aggregate Functions

As noted earlier, aggregate functions like AVG are among the SQL features that lack composability. For each of the traditional aggregate functions of SQL, SQL++ Core provides a fully composable function that takes a collection as input and returns the aggregated value of that collection. The composable version of AVG is named COLL_AVG. This naming convention applies to the other SQL aggregate functions as well: MAX has a composable version named COLL_MAX, etc.

In this section we will illustrate, by example, how SQL queries containing aggregate functions are transformed into SQL++ Core queries using composable functions. The examples are based on a flat collection named hr.emp with (at least) four columns: name, deptno, title, and salary. The theme of the transformation process is that the data or group of data that is being aggregated is first (conceptually) materialized and then passed (conceptually again) to the composable function which aggregates it. (It is important to point out that this materialization is conceptual; under the hood a SQL++ engine is free to optimize, e.g., by using pipelineable aggregation operations when evaluating a query.)

The first SQL query, shown in Listing 15, finds the average salary of engineers in the hr.emp collection. Its SQL++ Core equivalent is shown in Listing 16.

```
SELECT AVG(e.salary) AS avgsal
FROM hr.emp AS e
WHERE e.title = 'Engineer'
```

Listing 15. First aggregation query, SQL version

```
{{
  {'avgsal':
    COLL_AVG(
       SELECT VALUE e.salary
       FROM hr.emp AS e
       WHERE e.title = 'Engineer'
    )
  }
}}
```

Listing 16. First aggregation query, SQL++ core version

The second SQL query, shown in Listing 17, lists all of the departments in the `hr.emp` collection and the average salaries of their engineer employees. Its SQL++ core equivalent in shown in Listing 18. This SQL++ Core query is written using the permitted `SELECT`-clause-last style.

```
SELECT e.deptno, AVG(e.salary) AS avgsal
FROM hr.emp AS e
WHERE e.title = 'Engineer'
GROUP BY e.deptno
```

Listing 17. Second aggregation query, SQL version

```
FROM hr.emp AS e
WHERE e.title = 'Engineer'
GROUP BY e.deptno AS d GROUP AS g
SELECT VALUE
  {deptno: d,
   avgsal: COLL_AVG(
     FROM g AS gi
     SELECT gi.e.salary
   )
  }
```

Listing 18. Second aggregation query, SQL++ Core version

VI. Pivoting and Unpivoting

In use cases that require the structuring and/or reorganizing of semistructured data it is important to have the ability to flexibly turn data into attributes and vice versa. This is what the pivoting and unpivoting features described next accomplish.

A. Unpivoting Tuples

Let us begin by examining the use case for unpivoting. The collection of Listing 19 is an interesting example, as it uses as attribute *names* data that would typically be attribute *values* in the SQL world. The query given in Listing 20 unpivots the stock ticker/price pairs in this data set and returns the result shown in Listing 21.

```
{{
  {'date': '4/1/2019',
   'amzn': 1900, 'goog': 1120, 'fb': 180},
  {'date': '4/2/2019',
   'amzn': 1902, 'goog': 1119, 'fb': 183}
}}
```

Listing 19. The `closing_prices` collection

```
SELECT c."date" AS "date",
       sym AS symbol,
       price AS price
FROM closing_prices AS c,
     UNPIVOT c AS price AT sym
WHERE NOT sym = 'date'
```

Listing 20. A query that unpivots ticker/price pairs

```
{{
  {
    'date': '4/1/2019',
    'symbol': 'amzn',
    'price': 1900
  },
  {
    'date': '4/1/2019',
    'symbol': 'goog',
    'price': 1120
  },
  {
    'date': '4/1/2019',
    'symbol': 'fb',
    'price': 180
  },
  {
    'date': '4/2/2019',
    'symbol': 'amzn',
    'price': 1902
  },
  {
    'date': '4/2/2019',
    'symbol': 'goog',
    'price': 1119
  },
  {
    'date': '4/2/2019',
    'symbol': 'fb',
    'price': 183
  }
}}
```

Listing 21. The result of unpivoting ticker/price pairs

```
SELECT sym AS symbol,
       AVG(price) AS avg_price
FROM closing_prices c,
     UNPIVOT c AS price AT sym
WHERE NOT sym = 'date'
GROUP BY sym
```

Listing 22. A SQL++ query that computes average stock prices

Unpivoting tuples enables the use of attribute names as if they were data. For example, it becomes easy to compute the average price for each symbol as in the query of Listing 22.

```
{{
  {'symbol': 'amzn', 'price': 1900},
  {'symbol': 'goog', 'price': 1120},
  {'symbol': 'fb',   'price': 180}
}}
```

Listing 23. The `today_stock_prices` collection

```
PIVOT sp.price AT sp.symbol
FROM today_stock_prices sp
```

Listing 24. A `PIVOT` query

```
{
  'amzn': 1900,
  'goog': 1120,
  'fb': 180
}
```

Listing 25. The result of pivoting

B. Pivoting Tuples

Pivoting, conversely, serves to turn a collection into a tuple. For example, consider the collection `today_stock_prices` shown in Listing 23. Using this collection as input, the `PIVOT` query of Listing 24 produces the tuple shown in Listing 25.

The query of Listing 26 uses the grouping features of SQL++ together with pivoting in order to create a single tuple for all of the stock prices of each date when it inputs the collection of Listing 27. Listing 28 shows this query's result.

```
SELECT sp."date" AS "date",
       (PIVOT dp.sp.price AT dp.sp.symbol
        FROM dates_prices AS dp) AS prices
FROM stock_prices AS sp
GROUP BY sp."date" GROUP AS dates_prices
```

Listing 26. Combination of grouping and pivoting

```
{{
  {'date': '4/1/2019',
   'symbol': 'amzn', 'price': 1900},
  {'date': '4/1/2019',
   'symbol': 'goog', 'price': 1120},
  {'date': '4/1/2019',
   'symbol': 'fb', 'price': 180 },
  {'date': '4/2/2019',
   'symbol': 'amzn', 'price': 1902},
  {'date': '4/2/2019', 'symbol':
   'goog', 'price': 1119},
  {'date': '4/2/2019',
   'symbol': 'fb', 'price': 183 }
}}
```

Listing 27. The `stock_prices` input

```
{{
  {
    'date': '4/1/2019',
    'prices':
      {'amzn': 1900, 'goog': 1120, 'fb': 180}
  },
  {
    'date': '4/2/2019',
    'prices':
      {'amzn': 1902, 'goog': 1119, 'fb': 183}
  }
}}
```

Listing 28. The output of pivoting stock prices by date

VII. Prior Work

The history of the SQL language goes back five decades [21]. As commercial relational database offerings began to take hold in the business data world, the database community began expanding its reach to incorporate newer and richer forms of data. One target was engineering data, which led to a decade or so of work on object-oriented database systems with data models inspired by concepts from object-oriented programming languages [22], [23]. Languages like OQL [24] borrowed ideas from SQL but extended their reach to typed nested data. More or less in parallel, traditional relational database enthusiasts sought to extend the reach of relational systems to incorporate object-oriented ideas, leading to so-called object-relational systems such as Postgres, Starburst and others [23]. These systems also had SQL-like languages that could operate over richer but typed table structures with extensible column types.

Fast forwarding to 25 years ago, the database community became interested in exploring support for semistructured data. The Lore project at Stanford with its OQL-inspired query language Lorel [25] was arguably the seminal work in this direction. The XML data format was also becoming prevalent in roughly the same time frame, and was a practical, commercial example of self-describing, schema-less data. Query languages capable of querying XML data garnered much interest in both research and industry, and the W3C XQuery language standard [26] was the result of a community effort to meet that need. XQuery faced many of the challenges that SQL++ faced, but it focused on XML and departed significantly from SQL in many of its details.

Fast forwarding to the recent past, to the so-called "big data" and/or "NoSQL" eras, leads to the problem of querying and manipulating massive quantities of semistructured data (as well as applying parallelism to scale). On the NoSQL side, Cassandra and its CQL query language [27] emerged; CQL can be characterized as a SQL-like language, minus joins, for operating on nested but schema-ful tables. MongoDB also appeared; its approach to querying involves a mix of a basic "CRUD" API [28] and an "aggregation pipeline" API [29]. Microsoft's document database offering, Cosmos DB [30], has a SQL-based query language for querying single collections of documents that is roughly like a single-collection subset of SQL++; only intra-document joins are possible, for example.

Big data analytics engines, such as Hive [19] and later Spark [31], also appeared in this time frame. Both initially targeted very large volumes of mostly flat, schema-ful data, intending to supplant traditional data warehouse systems, so they each based their query languages (HiveQL and SparkSQL, respectively) on SQL. As time has progressed, their query languages have moved closer to SQL compliance; they also include features such as generic "explode" or "unnest" table functions to address the need to query non-flat data (e.g., Parquet files or schema-ful JSON). Most recently, Rockset has arrived on the scene as a data integration and indexing offering fronted with a heavily SQL-inspired query language [32] that relaxes SQL's restrictions in ways not unlike SQL++.

VIII. Conclusion

In this paper we have described the SQL++ query language. SQL++ is a SQL extension that relaxes SQL's strictness in terms of both object structure (moving from flat $\rightarrow$ nested) and schema (moving from mandatory $\rightarrow$ optional). SQL++ sees relational data as a subset of a more flexible object model and it sees collections of document data as a natural and supportable relaxation as opposed to a "bolt on" addition such as a new SQL column type [33]. We toured the core features of SQL++ and explained how its definition can accommodate flexible data while also remaining true to SQL in situations where a query's target data is tabular and strongly typed.

As we have described, a multi-party effort is well underway to converge on a Core SQL++ definition and syntax that is supportable by multiple vendors. Work is also underway to incrementally close the syntactic and semantic gaps between their existing SQL++ based offerings and the core. Future joint work is expected to include developing a shared "compatibility kit" for use in checking for compliance with Core SQL++ in both its composability mode and its SQL compatibility mode. In closing, we would like to invite other systems' developers and tool providers to "relax" with us and join the effort.

References

[1] K. W. Ong, Y. Papakonstantinou, and R. Vernoux, "The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases," *CoRR*, vol. abs/1405.3631, 2014. [Online]. Available: http://arxiv.org/abs/1405.3631

[2] ——, "The SQL++ unifying semi-structured query language and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases," *CoRR*, vol. abs/1405.3631, 2014. [Online]. Available: https://arxiv.org/pdf/1405.3631v6.pdf

[3] (2024) Apache AsterixDB. Apache Software Foundation. [Online]. Available: https://asterixdb.apache.org/

[4] (2024) SQL++: Couchbase database query language. Couchbase, Inc. [Online]. Available: https://www.couchbase.com/products/n1ql/

[5] (2024) What's SQL++ for Analytics? Couchbase, Inc. [Online]. Available: https://docs.couchbase.com/server/current/analytics/1_intro.html

[6] D. Chamberlin, *SQL++ For SQL Users: A Tutorial*, 2018. [Online]. Available: https://a.co/d/eWFxyBC

[7] (2024) PartiQL tutorial. PartiQL. [Online]. Available: https://partiql.org/tutorial.html

[8] (2024) Querying semistructured data. Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/redshift/latest/dg/query-super.html

[9] (2024) Amazon QLDB PartiQL reference. Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/qldb/latest/developerguide/ql-reference.html

[10] (2024) PartiQL - a SQL-compatible query language for Amazon DynamoDB. Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-reference.html

[11] (2024) Amazon S3 SELECT command. Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-select-sql-reference-select.html

[12] (2024) AWS IoT TwinMaker knowledge graph. Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/iot-twinmaker/latest/guide/tm-knowledge-graph-resources.html

[13] D. Chamberlin, "49 years of queries (keynote video)," in *Companion of the 2023 International Conference on Management of Data*, ser. SIGMOD '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3555041.3589336

[14] (2024) Introducing JSON. json.org. [Online]. Available: https://www.json.org/json-en.html

[15] (2024) CBOR: RFC 8949 Concise binary object representation. IETF. [Online]. Available: https://cbor.io/

[16] (2024) Amazon Ion. Amazon Web Services. [Online]. Available: https://amazon-ion.github.io/ion-docs/

[17] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970. [Online]. Available: https://doi.org/10.1145/362384.362685

[18] M. Carey, "AsterixDB mid-flight: A case study in building systems in academia," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2019, pp. 1–12. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICDE.2019.00008

[19] (2024) Apache Hive language manual. Apache Software Foundation. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/LanguageManual

[20] M. Carey, I. Maxon, T. Westmann, D. Lychagin, P. Sinthong, and G. Galviso, "JSON analytics with Apache AsterixDB," *Big Data Open Source Systems Workshop (BOSS)*, 2020, Tutorial. [Online]. Available: https://boss-workshop.github.io/boss-2020/

[21] D. Chamberlin, "50 years of queries," *Commun. ACM*, 2024 (to appear).

[22] S. B. Zdonik and D. Maier, Eds., *Readings in object-oriented database systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989. [Online]. Available: https://www.amazon.com/Readings-Object-Oriented-Database-Kaufmann-Management/dp/1558600000

[23] *Commun. ACM*, vol. 34, no. 10, 1991, (Special issue on next-generation database systems.). [Online]. Available: https://dl.acm.org/toc/cacm/1991/34/10

[24] S. Cluet, "Designing OQL: Allowing objects to be queried," *Information Systems*, vol. 23, no. 5, pp. 279–305, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437998000131

[25] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, "The Lorel query language for semistructured data," *Int. J. Digit. Libr.*, vol. 1, no. 1, pp. 68–88, 1997. [Online]. Available: https://doi.org/10.1007/s007990050005

[26] H. Katz, D. Chamberlin, M. Kay, P. Wadler, and D. Draper, *XQuery from the Experts: A Guide to the W3C XML Query Language*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. [Online]. Available: https://www.amazon.com/exec/obidos/ASIN/0321180607/acmorg-20

[27] (2024) The Cassandra query language (CQL). Apache Software Foundation. [Online]. Available: https://cassandra.apache.org/doc/stable/cassandra/cql/

[28] (2024) MongoDB CRUD operations. MongoDB, Inc. [Online]. Available: https://www.mongodb.com/docs/manual/crud/

[29] (2024) Aggregation operations. MongoDB, Inc. [Online]. Available: https://www.mongodb.com/docs/manual/aggregation/

[30] (2024) Queries in Azure Cosmos DB for NoSQL. Microsoft, Inc. [Online]. Available: https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/query/

[31] (2024) Spark SQL reference. Apache Software Foundation. [Online]. Available: https://spark.apache.org/docs/latest/sql-ref.html

[32] (2024) Rockset SQL Guide. Rockset.com. [Online]. Available: https://docs.rockset.com/documentation/docs/sql-guide

[33] D. Chamberlin, "Comparing two SQL-based approaches for querying JSON: SQL++ and SQL:2016," 2019, Whitepaper, Couchbase, Inc. [Online]. Available: https://info.couchbase.com/rs/302-GJY-034/images/Comparing_Two_SQL_Based_Approaches_WP.pdf