The SQL++ Query Language: Configurable, Unifying and Semi-structured

Kian Win Ong, Yannis Papakonstantinou, Romain Vernoux
{kianwin,yannis,rvernoux}@cs.ucsd.edu

ABSTRACT

NoSQL databases support semi-structured data, typically modeled as JSON. They also provide limited (but expanding) query languages. Their idiomatic, non-SQL language constructs, the many variations, and the lack of formal semantics inhibit deep understanding of the query languages, and also impede progress towards clean, powerful, declarative query languages.

This paper specifies the syntax and semantics of SQL++, which is applicable to both JSON native stores and SQL databases. The SQL++ semi-structured data model is a superset of both JSON and the SQL data model. SQL++ offers powerful computational capabilities for processing semi-structured data akin to prior non-relational query languages, notably OQL and XQuery. Yet, SQL++ is SQL backwards compatible and is generalized towards JSON by introducing only a small number of query language extensions to SQL. Indeed, the SQL capabilities are most often extended by removing semantic restrictions of SQL, rather than inventing new features.

Recognizing that a query language standard is probably premature for the fast evolving area of NoSQL databases, SQL++ includes configuration options that formally itemize the semantics variations that language designers may choose from. The options often pertain to the treatment of semi-structuredness (missing attributes, heterogeneous types, etc), where more than one sensible approaches are possible.

SQL++ is unifying: By appropriate choices of configuration options, the SQL++ semantics can morph into the semantics of existing semi-structured database query languages. The extensive experimental validation shows how SQL and four semi-structured database query languages (MongoDB, Cassandra CQL, Couchbase N1QL and AsterixDB AQL) are formally described by appropriate settings of the configuration options.

Early adoption signs of SQL++ are positive: Version 4 of Couchbase’s N1QL is explained as syntactic sugar over SQL++. AsterixDB will soon support the full SQL++ and Apache Drill is in the process of aligning with SQL++.

1. INTRODUCTION

Numerous databases marketed as SQL-on-Hadoop, NewSQL and NoSQL support Big Data applications. These databases generally support the 3Vs[7]. (i) Volume: amount of data (ii) Velocity: speed of data in and out (iii) Variety: semi-structured and heterogeneous data. Due to the Variety requirement, they have adopted semi-structured data models, which are generally different subsets of enriched JSON[8].

Their evolving query languages fall short of full-fledged semi-structured query language capabilities[2] and have many variations. Some variations are due to superficial syntactic differences. However, other variations are genuine differences in query language capabilities and semantics. The lack of succinct, formal syntax and semantics inhibits a deep understanding of the various systems. It also impedes progress towards declarative languages for querying semi-structured data.

SQL++ is a semi-structured query language that is backwards compatible with SQL, in order to be easily understood and adopted by SQL programmers. The described semi-structured SQL++ data model is a superset of JSON and the SQL data model. The SQL++ model expands SQL with bags (as opposed to having JSON arrays only) and enriched values, i.e., atomic values that are not only numbers and strings (vendors have already adopted this extension[5]). Vice versa, one may think of SQL++ as expanding SQL with JSON features: arrays, heterogeneity, and the possibility that any value may be an arbitrary composition of the array, bag and tuple constructors, hence enabling arbitrary nested structures, such as arrays of arrays. The SQL++ query language inputs and outputs SQL++ data. It makes the following contributions towards the evolution of query languages for JSON databases.

Full-fledged semi-structured language Many commercial JSON databases started as key-value and document-oriented databases. Others started with SQL as their base. In either case, they grow towards full-fledged JSON databases. SQL++ provides a full-fledged target language whose semantics pick the salient features of past full-fledged declarative query languages for non-relational data models: OQL[2], the nested relational model and query languages[8, 15, 1] and XQuery (and other XML-based query languages)[4, 6]. Importantly, in the spirit of XQuery and OQL, SQL++ is a fully composable and semi-structured language, hence being able to input and output nested and heterogeneous

---

9¹As explained below, the SQL data model itself is a subset of enriched JSON.
9²They also fall short of full-fledged SQL capabilities also.
structures. The cornerstones of achieving this while also being SQL compatible, are described next.

Removing restrictions of SQL semantics Instead of inventing (or re-inventing) many new features, the extension from SQL to SQL++ is most often achieved simply by removing semantic restrictions of SQL:

1. Unlike SQL’s FROM clause variables, which bind to tuples only, the FROM clause variables of SQL++ bind to any JSON element.
2. SQL++ is fully composable in the sense that subqueries can appear anywhere, potentially creating nested results when they appear in the SELECT clause.
3. While correlation of the subqueries in a FROM clause is not allowed in SQL, the SQL++ subqueries of a FROM clause may be correlated with (earlier defined) variables of the same FROM clause.
4. The groups created by the GROUP BY 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. Interestingly, the SQL++ approach ends up explaining in a simpler way even plain SQL’s grouping and aggregation.
5. Unlike SQL, the SQL++ semantics do not require schema or any homogeneity on the input data.

Extensions: Variables binding to attribute/value pairs and array positions In addition to not requiring a schema, SQL++ can allow a pair of variables of the FROM clause to range over the attribute name and attribute value pairs of input tuples. Similarly, when a FROM clause ranges over an array, SQL++ allows a pair of variables to capture the data at an array position and the index of this position. In this way SQL++ seamlessly expands the logic of SQL to “schema” inspection and arrays.

As a result, the SQL++ syntax and semantics are short (as evidenced by this paper’s syntax and semantics definitions), despite adding functionality on SQL. A key methodology that led to short, succinct semantics is the staging: First we define an SQL++ core. Consequently, SQL compatibility is achieved as syntactic sugar over the core. Similarly, many JSON database query language constructs of various vendors, can also be reduced to syntactic sugar over the core. As a proof of concept, we coauthored with the Couchbase N1QL team a reduction of the N1QL v4 query language into the SQL++ core, by using simple rewriting rules that treat the N1QL v4 syntactic constructs as syntactic sugar over SQL++. Indeed the formal semantics of N1QL v4 are explained by this reduction [12]. (We had earlier collaborated with Couchbase to align N1QL v4 with SQL++.)

The simplicity and power of the language have led to positive early adoption signs: (a) the Couchbase N1QL - SQL++ alignment, (b) an SQL++ interface for AsterixDB[6] captures the SELECT-FROM-WHERE language part and a full release of SQL++ for AsterixDB will soon follow; and (c) collaboration with MapR’s Apache Drill towards aligning the query language with SQL++. An earlier version of SQL++ has been used by the federated query processor of the FORWARD application development platform http://forward.ucsd.edu/sqlpp.

Configurable At the same time, via communication with the vendors we recognize that a model and query language standard is probably premature for such a young and fast-evolving area. Yet, the language designers need to know now the options that are available, especially as it pertains to the handling of semi-structured aspects (semantics for missing attributes, heterogeneous types, etc), which are not captured by the SQL backwards compatibility.

To handle this requirement, SQL++ includes configuration options that describe

1. which features are supported and
2. (for the supported features) different options of language semantics that formally capture the variations of existing database query languages.

Unifying and Explanatory By appropriate choices of configuration options, the SQL++ semantics morph into the semantics of other SQL+JSON databases. The paper shows how the SQL standard and four well known (Cassandra CQL, MongoDB, Couchbase N1QL, AsterixDB AQL) semi-structured database query languages, are explained as particular settings of the configuration options. While SQL++ does not support the exact syntax of any of these four databases, it can be morphed by the configuration options to support equivalent queries. By understanding each database’s capabilities in terms of SQL++ configuration options, the reader can focus on the fundamental differences of the databases without being confused by syntactic idiosyncrasies of various query languages and superficial differences in the documented descriptions of their semantics. To further facilitate understanding of SQL++ and the effect of the various configuration options, we provide a web-accessible reference implementation of SQL++ at http://forward.ucsd.edu/sqlpp.

An earlier, extended version [13] shows how an additional six databases correspond to particular settings of the configuration options. We expect that some of the results listed in the feature matrices describing configuration options will change in the next years as the space evolves rapidly. Despite the forthcoming changes, we expect SQL++ to remain a standing contribution in

1. guiding towards formal, SQL backwards compatible, minimalistic syntax and semantics for a full-fledged semi-structured JSON language and
2. enable understanding of differences in this space and guide towards rational decisions via the configuration option mechanism.

2. DATA MODEL

The SQL++ data model is a superset of both SQL’s relational tables and JSON, based on the observation that they both use similar concepts: A SQL tuple corresponds to a JSON object literal and a SQL string/integer/boolean to the respective JSON scalar. A JSON array is similar to a SQL table (bag) with order.

Figure 1 shows the BNF grammar for SQL++ values, and Figure 2 shows an example SQL++ value. A SQL++ database generally contains one or more SQL++ top-level named values (BNF line 1). A name, such as the name sensors of Figure 2 is a string and is unique. A value is a scalar, complex, missing (explained below) or null. A complex value is either a tuple or a collection. A tuple is a set of attribute name/value pairs, where each name is a unique string within the tuple (as in SQL). Notice, SQL++
The elements of an array/bag can be any kind of value. Each attribute value can be a scalar, a tuple, a collection, or a complex value. A collection is either an array or a bag. An array is ordered (similar to a SQL table) and its elements cannot be accessed by position in Section 4.2. In contrast, a bag is unordered (similar to a SQL table) and its elements cannot be accessed by ordinal position. (See specifics of access behavior in Section 5.)

A SQL++ query is either an SFW query (i.e., SELECT-FROM-WHERE, lines 5-16 of the grammar of Figure 3) or an expression query (or expression, lines 17-27). Unlike SQL expressions, which are restricted to outputting scalar and null values, SQL++ expression queries output arbitrary values, and are fully composable with SFW queries. Furthermore, SQL++ supports the top-level query to also be an expression query, not just a SFW query as in SQL.

**3. QUERIES, ENVIRONMENTS AND BINDING TUPLES**

A SQL++ query is either an SFW query (i.e., SELECT-FROM-WHERE, lines 5-16 of the grammar of Figure 3) or an expression query (or expression, lines 17-27). Unlike SQL expressions, which are restricted to outputting scalar and null values, SQL++ expression queries output arbitrary values, and are fully composable with SFW queries. Furthermore, SQL++ supports the top-level query to also be an expression query, not just a SFW query as in SQL.

A SQL++ (sub)query is evaluated within an environment, which provides variable bindings (as defined next) and optionally configuration options, defined in Section 4.

**Environments and Variable Bindings:** Each SQL++ (sub)query $q$ is evaluated within an environment (for brevity, env) $\Gamma$ comprising a $\text{config}$ $\text{env} \Gamma_c$ and a variable $\text{named_value}$ $\rightarrow$ $\text{name} :: \text{value}$

Figure 1: BNF Grammar for SQL++ Values

Figure 2: Example of a SQL++ Value

syntax follows JSON. Therefore, tuples are denoted by { }. In Figure 2 sensors is a tuple with attributes location and readings. Each attribute value can be a scalar, a null, or a complex value (BNF line 15).

A collection is either an array or a bag (BNF lines 16-17). Both arrays and bags may contain duplicate elements. An array is ordered (similar to a JSON array) and each element is accessible by its ordinal position. (See specifics of access by position in Section 4.2.) In contrast, a bag is unordered (similar to a SQL table) and its elements cannot be accessed by ordinal position. Following JSON’s syntax, arrays are denoted with [...], whereas following AQL’s syntax, bags are denoted with {...}. The value of readings is an array (Example lines 3-13). The value of ozone is a bag (Example line 11).

A scalar value is either primitive or enriched (BNF lines 8-9). Primitive values are the scalar values of the JSON specification, i.e., strings, numbers or booleans (BNF lines 10-13). Enriched values (such as dates and timestamps) are extensions over JSON, and are specified using a type constructor over primitives (BNF line 14). The value timestamp(‘2014-03-12T20:00:00’) (Example line 5) is enriched.

The elements of an array/bag can be any kind of value and can be heterogeneous. That is, there are no restrictions between the elements of an array/bag. For example, the two tuples in the array of lines 3 to 13 are heterogeneous because: (i) each tuple has a different set of attributes (lines 5-7 vs 10-12), (ii) the ozone attribute of the first tuple maps to a number (line 6) while the ozone of the second maps to a bag (line 11). The elements of the ozone bag of line 11 are also heterogeneous (1st and 3rd are numbers, 2nd is a tuple). Furthermore, unlike SQL where the values are tables that have homogeneous tuples that have scalars, SQL++ allows arbitrary composition of complex values. E.g., the top level value is a tuple, the array of lines 3-13 has two heterogeneous tuples, and the bag of line 11 has two scalars and one tuple.

SQL++ and many semi-structured databases also contains maps. A map contains mappings, where each mapping maps a left value to a right value, and each left value is unique within the map. In effect, both a map and a tuple are sets of name/value pairs where names are unique. Their only difference is that a map allows the left value of each mapping to be any arbitrary value, whereas a tuple restricts attribute names to strings (as in SQL). Therefore we do not further discuss maps, as they are essentially tuples without the string restriction.
1. When a query is submitted to a database, it is evaluated in the environment comprising $\Gamma$, which may differ from the environment it is written in. For example, $\Gamma = \{ (\text{NAME} \rightarrow \text{value} : \text{DATA} \rightarrow \text{data}) \}$. The query is evaluated in the environment of the query (denoted by $\Gamma$). The FROM clause inputs a bag of binding tuples $B_{\text{FROM}}$ (which has 3 binding tuples in the example). Each binding tuple of $B_{\text{FROM}}$ maps a variable of the FROM clause to a value. There are no restrictions that a variable binds to a value in a FROM clause.

2. The pattern of "input bag, evaluate constituent expressions, output bag" has a few exceptions, each of them exemplified in the example: First, the ORDER BY clause inputs a bag of binding tuples and outputs an array of binding tuples. Second, a LIMIT/OFFSET clause need not evaluate its constituent expression for each input binding tuple. Finally, the SELECT ELEMENT clause inputs a bag and outputs an array of binding tuples.

Figure 4: Binding tuples in an SFW Query

\[
\begin{align*}
\text{FROM} & \quad \text{Readings AS } r \\
\text{WHERE} & \quad r.co < \text{max} \\
\text{ORDER BY} & \quad r.no2 \\
\text{LIMIT} & \quad 2 \\
\text{SELECT ELEMENT} & \quad r.no2
\end{align*}
\]

\[
\text{bindings env} \Gamma_b \text{ (or binding env). (We may omit the subscript } b \text{ in examples where the environment clearly is a variable binding env.) A binding env is a binding tuple } (x_1 : v_1, \ldots, x_n : v_n), \text{ where each } x_i \text{ is a variable that is unique, and binds to the SQL++ value } v_i. \text{ Notice that a binding tuple is structurally a SQL++ tuple. The characterization "binding" pertains to its use in the semantics, rather than structural differences.}

\text{Given two binding tuples } b \text{ and } b', \text{ their concatenation is a binding tuple, denoted as } b||b', \text{ that has the variable bindings of both } b \text{ and } b'. \text{ In order to define variable scoping later, we require that if a variable } x \text{ is bound in both } b \text{ and } b', \text{ } b||b' \text{ retains only the binding of } x \text{ in } b. \text{ Hence each variable remains unique within a binding tuple.}

\text{The notation } \Gamma_b \vdash q \rightarrow v \text{ denotes that the SQL++ query } q \text{ evaluates to the value } v \text{ within } \Gamma_b, \text{ i.e. when every free variable of } q \text{ is instantiated by its binding in } \Gamma_b. \text{ For example, } \langle x : 5, y : 3 \rangle \vdash (x + y)/2 \rightarrow 4.

\text{The binding environments of a query and of its subqueries are produced as follows:}

1. When a query is submitted to a database, it is evaluated in the environment comprising $\Gamma = \{ (n_1 : v_1, \ldots, n_m : v_m) \}$ where the pairs $n_i : v_i$ are the database’s named values $v_i$ and their names $n_i$. For example, Figure 4 shows a SFW query that filters for co readings less than max, and outputs the top-2 no2 readings. The query is evaluated on a database that has two named values: the bag named readings and the number named max. Thus, the query’s environment $\Gamma_b$ has variable bindings readings and max.

2. The FROM clause of a SFW query produces new environments by concatenating bindings of its variables to the environment of its SFW query, as explained below. Subqueries are evaluated in these new environments. (The GROUP BY clause also produce additional variable bindings, as explained in Section 5.3.)

\textbf{SFW Query Clauses:} Similar to SQL semantics, the clauses of a SQL++ SFW query are evaluated in the following order: FROM, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT / OFFSET and SELECT. The FROM expression (readings in Figure 4) is evaluated in the environment of the query ($\Gamma$ in the example). Thereafter the FROM clause outputs the bag of binding tuples $B_{\text{FROM}}$ (which has 3 binding tuples in the example). In each binding tuple of $B_{\text{FROM}}$, each variable of the FROM clause is bound to a value. There are no restrictions that a variable binds to homogeneous values across binding tuples. In the example, r respectively binds to 3 values that are heterogeneous.

Each subsequent clause inputs a bag of binding tuples, evaluates the constituent expressions of the clause (which may themselves contain nested SFW queries), and outputs a bag of binding tuples that is in turn input by the next clause. For instance, the WHERE clause inputs the bag of binding tuples that have been output by FROM ($B_{\text{FROM}}^\text{out} = B_{\text{FROM}}^\text{in}$), and outputs the subset thereof that satisfies the condition expression $e$ of the WHERE clause. Condition $e$ is evaluated once for each input binding tuple $b$ in the binding env $B_{\text{env}}$, where $b$ is the binding env of the SFW query. In the example, the condition $r.co < \text{max}$ is evaluated once for each of the 3 input binding tuples of $B_{\text{env}}$. The binding env of the first evaluation is $\langle r : \{ (\text{co} : 2.2) \} \rangle |\Gamma_b = (r : \{ (\text{co} : 2.2) \}), \text{readings : } \ldots, \text{max : } 2$, thus the first binding tuple is not output.

The pattern of “input bag, evaluate constituent expressions, output bag” has a few exceptions, each of them exemplified in the example: First, the ORDER BY clause inputs a bag of binding tuples and outputs an array of binding tuples. Second, a LIMIT/OFFSET clause need not evaluate its constituent expression for each input binding tuple. Finally, the SELECT ELEMENT clause inputs a bag and outputs an array of binding tuples.

\textbf{Generalizing Variable Bindings:} Assume that readings in Figure 4 were a SQL table (i.e. a bag of homogeneous tuples) with number attributes co and no2. SQL semantics dictate that FROM readings AS r outputs homogeneous tuples with attributes co and no2. In contrast, the SQL++ FROM clause outputs binding tuples, each of which binds the single variable $r$ to a potentially heterogeneous tuple. In the example, $r$ binds to tuples containing co and no2 attributes. This additional level of nesting generalizes SQL++ variable bindings beyond SQL’s homogeneous tuples, to encompass arbitrary readings elements that are heterogeneous (as shown in the example), including a mixture of scalars, tuples and/or collections.

\textbf{Composability:} Notice that query evaluation treats each value named $n$ identically to a variable named $n$. For example, the query of Figure 4 could be a nested query where readings and max are variables defined in the FROM clause.

\textbf{7} SQL++ also supports a syntax improvement where SELECT is optionally written as the last clause.
of the parent query. This property serves the composability of the language.

**Heterogeneity:** SQL++ supports named values, query results and variables to be arbitrary values. This is in contrast to SQL which restricts named values to flat homogeneous tables (created by `CREATE TABLE`, `CREATE VIEW` and `WITH`), query results to flat homogeneous tables, and tuple variable bindings to homogeneous flat tuples.

The combination of composability and heterogeneity are needed to input and output arbitrarily structured SQL++ values.

## 4. Configuration Options

Config options enable SQL++ to be instantiated to different capabilities and semantics. In effect, they represent the choices that language designers may consider. Some options specify whether a syntactic feature is supported or not. For example, `GROUP BY` is supported by MongoDB, but not Cassandra. Other options specify variations in the semantics of a syntactic feature. For example, given a tuple `t` with a single attribute `a`, AsterixDB evaluates the path step `t.a` to return `null` (effectively losing the difference between a missing `b` attribute and a `b` attribute with `null` value). Couchbase returns `missing` (which behaves differently from `null`), and SQL throws an error.

Config options are extensively utilized throughout SQL++. This section presents the formalism, a simple use case in path navigation (Section 4.2) and a complex use case in equality (Section 4.3). The support/non-support of query features described in Section 2 are also specified by config options. Section 2 experimentally validates these config options by using them to systematically (a) describe the features of the surveyed databases as a subset of SQL++ and (b) describe their semantics as a specific instantiation of SQL++ semantics.

### 4.1 Config Envs, Parameters and Options

A config annotation `@g(p:o)` (Figure 5, lines 28-34) configures a (sub)query `q` with parameter group `g`, parameter name `p` and option `o`.

The config annotation is statically specified: it cannot be the result of another query. A query is evaluated in the configuration env(ironment) created by its config annotation. A config env `Γ` is a config tuple `(p1: o1, ..., pm: om)`, where each parameter `pi` maps to an option `oi`.

Using config options, language designers have the flexibility for very fine-grained customization of language semantics. To assist designers in choosing sets of options that are compatible with each other a *macro config annotation* `@g{p1:o1} ... @g{pm:om}`. For example, a macro config annotation may dictate that the semantics of `WHERE` in the presence of `null` and `missing` are identical to SQL's 3-valued logic, whereas `missing` is treated as `null`. Alternatively, the annotation may dictate another form of semantics where the distinction between `null` (i.e., unknown) and `missing` (i.e. inapplicable) is retained.

The utility of configuration override in federated query processing: A query’s configuration environment

9 As syntactic sugar, parameters of the same group only need to specify the group once (lines 28-32), i.e. `
`.

### 4.2 Path Navigation

A tuple path navigation `t.a` from the tuple `t` to its attribute `a` (Figure 3, line 21) returns the value of `a`. An array path navigation `a[i]` returns the `i`-th element of the array `a` (line 22). Notice that consecutive tuple/array navigations (e.g. `r.no[1]`) navigate deeply into complex values, and each navigation evaluates to a unique value. This is identical to SQL’s behavior, but different from XQuery/XPath’s behavior where each path step may return a set of one or more nodes.

Since SQL++ does not assume a schema, the semantics must also specify the return value when `t` is not a tuple or

**Figure 5:** BNF Grammar for Path Parameters/Options

```
1 path_param
2   →   absent : (null|missing|error)  
3 | type_mismatch: (null|missing|error)
```

**Figure 6:** Path Evaluation Functions for `t.a` and `a[i]`

| `t.a` | `t.a` | `a[i]` |
|-------|-------|-------|
| `@tuple_nav.absent` | `@tuple_nav.type_mismatch` | `@array_nav.absent` |
| `@array_nav.type_mismatch` |
| if `t` is a tuple that maps `a` to `v` | otherwise | if `a` is an array with `i`-th element as `v` |
| if `t` is a tuple that does not map `a` | otherwise | if `a` is an array with `n` elements ∧ `(i < 1 ∨ i > n)` |

**Figure 7:** Configuring Tuple/Array Navigation

```
\[ \begin{array}{l}
Γ = \{ r: \text{co: 1.2, no: [0.5, 2]} \} \\
Q = \{ \text{@tuple_nav (absent: missing, type_mismatch: null)} \} \text{ } \{ \text{@array_nav (absent: missing, type_mismatch: null)} \} \\
\{ \text{r.co, r.so, 7.co, r.no[1], r.no[3], r.co[1]} \} \\
\} \\
\{ \text{[1.2, missing, null, 0.5, missing, null]} \}
\end{array} \]
```
Table 1: @nav.failure Macro Expansion

| @nav.failure                      | error | null | missing |
|-----------------------------------|-------|------|---------|
| @tupleNav.absent                  | error | null | missing |
| @tupleNav.typeMismatch            | error | null | missing |
| @arrayNav.absent                  | error | null | missing |
| @arrayNav.typeMismatch            | error | null | missing |

Figure 8: BNF Grammar for Equality Parameters/Options

when it is a tuple that does not have an attribute. Multiple options are possible for these cases and, indeed, multiple options have been taken by the surveyed databases. Figure 4 specifies the evaluation function of $t.a$ and the function of $a[i]$, which utilize parameters from the @tupleNav and arrayNav parameter groups (Figure 3) lines 29-30. The absent parameter (Figure 5 line 2) specifies the returned value when an attribute/element is absent: null, missing, or throw an error. The typeMismatch parameter (line 3) specifies whether to return null, missing, or throw an error when a tuple/array navigation is invoked on a non-tuple/array. For brevity, we omit trivial conversions from a parameter option to a SQL++ value. For example, $t.a$ @tupleNav.absent denotes returning a SQL++ null value for the parameter option null, a missing value for the option missing, and throwing an error for the option error.

In the query of Figure 7, the @tupleNav/@arrayNav config annotations specify that navigating into an absent attribute/element returns missing, and navigating from a non-tuple/array returns null. As another example, SQL’s semantics for tuple navigation is specified by @tupleNav (absent: error, typeMismatch: error). Since a SQL database has fixed schemas, these errors are caught by static type checking.

Although the four path parameters are independent of each other and can (in principle) be configured with different options, we recommend language designers to utilize the macro config annotation @nav.failure so that path navigation behaves consistently for all navigation failures. The macro supports three options, which expand to parameters and respective options as specified in Table 4: (i) error specifies a strict form of path navigation (as in SQL), such that navigation failures result in errors thrown. Notice, it expands to @tupleNav.absent.error, @tupleNav.typeMismatch: error, etc. (ii) null specifies a lenient form of path navigation, such that failures result in null values (iii) likewise, missing specifies lenient path navigation where failures result in missing values.

4.3 Equality

Selection and join queries prominently use the = equality function. Since SQL++ variables bind to arbitrary heterogeneous values and missing information is prevalent, the evaluation function for the equality comparison $x=y$ must specify the semantics of comparing specified (i.e., non-null, non-missing), null and missing values. Furthermore, the lack of schema may lead to comparing mismatched types. For instance, a type mismatch can evaluate to false/null/missing or an error. A language designer can also choose whether to follow SQL’s 3-valued logic in which $x=null\rightarrow null$. Analogous decisions apply for missing. Moreover, if deep equality is supported between complex values, there are further options for combining the results of shallow equality, which go beyond true/false to include null and missing.

Figure 9 shows the parameters and options of the equality parameter group, which are utilized in the equality evalu-
In the presence of `null` values, SQL++ core constructs support `null` values. Table 2 demonstrates how the `null` value is handled in SQL++. By defining a `null` value, all operations involving `null` yield `null` results. Moreover, `null` values can be compared using `null` logic, which uses five parameters: `null`, `true`, `false`, `missing`, and `unknown`. The `null` logic is defined as `null = null` and `null AND null = null`. Additionally, `null` elements can be compared using a `null` logic, which uses three parameters: `null`, `true`, and `false`. In Table 2, the `null` logic is evaluated using three cases: `null`, `true`, and `false`. The `null` logic is used to compare elements in SQL++ core constructs, allowing for the distinction between `null` and `true` values. This enables SQL++ core constructs to directly express queries over `null` values, providing a more expressive and flexible query language.

5. SFW QUERY CLAUSES

5.1 FROM clause

The SQL++ FROM clause allows variables to range over any data, unlike SQL FROM clause tuple variables that range over tuples only. The SQL++ variables can range over heterogeneous elements, over nested collections, and over type values pairs of tuples (a feature reminiscent of SQL extensions for dynamic schema introspection). Furthermore, SQL++ variables can register the order of input elements in arrays, which is useful for order-aware queries.

Some semi-structured query languages have introduced unnesting constructs (following the nested relational algebra [6, 15]) that essentially specify how the variables of the FROM clause range over nested data. The SQL++ core semantics show that these constructs can be fused with SQL and be expressed succinctly by the combination of (a) allowing correlated queries in SQL’s FROM clause and (b) a distinction of how variable bindings of the correlated queries combine, which is akin to the distinction between JOIN versus LEFT OUTER JOIN in SQL. The combination of these two simple features, allows the SQL++ core to directly express queries that could only be expressed in cumbersome ways (involving intermediate results) even in advanced prior languages, such as XQuery and the nested relational algebra.

Figure 11 shows the BNF grammar for the FROM clause and corresponding config parameters. The syntax and semantics of FROM are defined inductively with three cases. (1) Lines 4-5: the SQL++ core base case where the FROM item ranges over a single collection or tuple (2) Lines 6-7: the SQL++ core inductive case where the FROM item comprises correlation between two other FROM items (3) Lines 8-12: the “syntactic sugar” cases, where the grammar introduces well known SQL constructs (e.g., joins, outer joins) as well as the unnesting constructs of NoSQL databases. The semantics of the “syntactic sugar” constructs are explained by reduction to the SQL++ core constructs of #1 and #2.

(1a) SQL++ Core, Ranging over Collection Elements: FROM ? AS ? (AT ?) supports ranging over elements of a collection (line 4). x is an element variable, and y is an optional position variable. As in SQL expression e can be any SFW subquery or named value (Figure 3 lines 18-19). SQL++ extends beyond SQL towards full compositability, as e can also be a variable, path, function call,
may choose to coerce it into the collection \(\{\{\}\}\) to bag of binding tuples evaluates to a tuple \((1b)\) SQL++ Core, Ranging over Tuple Attributes:

- SQL++ enables ranging over the attribute name-value pairs of a tuple (Figure 11 line 5). When expression \(t\) evaluates to a tuple \(\langle a_1 : v_1, a_2 : v_2, \ldots, a_n : v_n \rangle\), FROM outputs the bag of binding tuples \(\langle \langle a : a_1, v : v_1 \rangle, \ldots, \langle a : a_n, v : v_n \rangle \rangle\).

- Figure 13 shows an example SQL++ query that inputs a tuple with 2 attributes \(no2\) and \(co\). The FROM clause defines variables \(g\) and \(v\), and outputs 2 binding tuples where \(g\) is bound to the attribute name and \(v\) is bound to the attribute value. The SELECT ELEMENT clause outputs a bag with 2 elements, one for each input tuple attribute. In SQL’s parlance, the SQL++ query has transposed/pivoted the tuple’s attributes (columns) into the collection’s elements (rows).

- If \(t\) evaluates into a value that is not a tuple, coercion may be activated by the configuration options of Figure 11 lines 18-19: When \(t\) evaluates to null or missing, the language designer may elect to coerce \(t\) into the empty tuple \(\{\}\), instead of producing an error.

- (2a) Core, INNER CORRELATE: The first inductive step case is FROM l INNER CORRELATE \(r\) (Figure 11 line 6). (For brevity, also written as FROM l INNER \(r\).) The FROM item \(r\) can utilize variables defined by FROM item \(l\). Suppose the SFW query is evaluated in binding env \(\Gamma\). Let \(B^l\) be the bag of binding tuples that would result from the clause FROM \(l\). Then, FROM \(l\) INNER CORRELATE \(r\) outputs all binding tuples \(B^l_i \times B^r_j\), where \(B^l_i \in B^l\), \(B^r_j \in B^r\), and \(B^r\) is the bag of binding tuples that would result from evaluating FROM \(r\) within the binding env \(\overline{\Gamma}\).

- For example, Figure 14 shows a SQL++ query that ranges over attribute/value pairs, thereafter ranges over the array of each attribute value. FROM reading AS \(g:\{\}\) would result in binding tuples \(b_1, b_2, b_3\), each with variables \(g\) and \(a\). Evaluating the right hand side \(a\) AS \(v\) in the context of \(b_1 = \langle g : "co" \rangle\) results into \(B^l = \langle \langle v : 0.7 \rangle \rangle\). Consequently the FROM clause outputs the first 2 binding tuples of Figure 13. Likewise \(b_2 = \langle g : no2, a : "repair" \rangle\), leads to \(B^r = \langle \langle v : "repair" \rangle \rangle\) which leads to FROM outputting the 3rd binding tuple. Since \(b_3 = \langle g : so2, a : [] \rangle\) results in an empty \(B^r\), there is no binding tuple that corresponds to \(so2\).

- Notice that correlated queries are akin to nested for-loops of imperative languages, where the left subquery \(l\) and the right subquery \(r\) are respectively the outer and inner loops. While this example has a simple expression \(r\) (i.e. the expression \(a\) AS \(v\)), in general \(r\) can be a complex SFW query. For example, we could replace the \(a\) AS \(V\) in Figure 14 with the following subquery, which would correlate each \(g, a\) pair only with the top-1 element of each collection.

- Figure 14 shows an example SQL++ query that inputs a tuple with 2 attributes \(no2\) and \(co\). The FROM clause defines variables \(g\) and \(v\), and outputs 2 binding tuples where \(g\) is bound to the attribute name and \(v\) is bound to the attribute value. The SELECT ELEMENT clause outputs a bag with 2 elements, one for each input tuple attribute. In SQL’s parlance, the SQL++ query has transposed/pivoted the tuple’s attributes (columns) into the collection’s elements (rows).

- If \(t\) evaluates into a value that is not a tuple, coercion may be activated by the configuration options of Figure 11 lines 18-19: When \(t\) evaluates to null or missing, the language designer may elect to coerce \(t\) into the empty tuple \(\{\}\), instead of producing an error.

- (2a) Core, INNER CORRELATE: The first inductive step case is FROM l INNER CORRELATE \(r\) (Figure 11 line 6). (For brevity, also written as FROM l INNER \(r\).) The FROM item \(r\) can utilize variables defined by FROM item \(l\). Suppose the SFW query is evaluated in binding env \(\Gamma\). Let \(B^l\) be the bag of binding tuples that would result from the clause FROM \(l\). Then, FROM \(l\) INNER CORRELATE \(r\) outputs all binding tuples \(B^l_i \times B^r_j\), where \(B^l_i \in B^l\), \(B^r_j \in B^r\), and \(B^r\) is the bag of binding tuples that would result from evaluating FROM \(r\) within the binding env \(\overline{\Gamma}\).

- For example, Figure 14 shows a SQL++ query that ranges over attribute/value pairs, thereafter ranges over the array of each attribute value. FROM reading AS \(g:\{\}\) would result in binding tuples \(b_1, b_2, b_3\), each with variables \(g\) and \(a\). Evaluating the right hand side \(a\) AS \(v\) in the context of \(b_1 = \langle g : "co" \rangle\) results into \(B^l = \langle \langle v : 0.7 \rangle \rangle\). Consequently the FROM clause outputs the first 2 binding tuples of Figure 13. Likewise \(b_2 = \langle g : no2, a : "repair" \rangle\), leads to \(B^r = \langle \langle v : "repair" \rangle \rangle\) which leads to FROM outputting the 3rd binding tuple. Since \(b_3 = \langle g : so2, a : [] \rangle\) results in an empty \(B^r\), there is no binding tuple that corresponds to \(so2\).

- Notice that correlated queries are akin to nested for-loops of imperative languages, where the left subquery \(l\) and the right subquery \(r\) are respectively the outer and inner loops. While this example has a simple expression \(r\) (i.e. the expression \(a\) AS \(v\)), in general \(r\) can be a complex SFW query. For example, we could replace the \(a\) AS \(V\) in Figure 14 with the following subquery, which would correlate each \(g, a\) pair only with the top-1 element of each collection.
ables defined by $l$. (For brevity one may simply write $l$
LEFT $r$, omitting either or both of the OUTER and CORRELATE
keywords.) Similar to SQL's LEFT JOIN, the LEFT out-
puts binding tuples derived from $l$, even if there are no
matching binding tuples from $r$. Formally, let $B'$ be
the bag of binding tuples that from $l$ INNER $r$ would output. In addition, for each $b_i \in B'$ such that $B'_i$ is an empty bag, LEFT also out-
puts a binding tuple $b_i \parallel \langle x_1', . . . , x_m' \rangle_{\text{from no match}},$ where $x_1', . . . , x_m'$ are variables defined by
$r$, and each variable is bound to the value configured by
$\text{from no match}$ (line 20). Since LEFT generalizes SQL's
LEFT JOIN (as discussed below), achieving SQL compatibility
requires configuring $\text{from no match}$ to null. Alterna-
tively, a language designer may also choose configuring to

For example, suppose Figure 11 uses LEFT instead of
INNER. Assuming $\text{from no match}$ is configured to null, the query will output the additional tuple: $(\text{gas} = \text{"so2"}, \text{val} = \text{null})$.

(2c) Core, FULL CORRELATE: from $l$ FULL CORRELATE $r$ ON $c$ (Figure 11 line 7) resembles SQL's FULL JOIN: $l$ and $r$
cannot utilize variables of each other, and their correlation
is specified by the condition $c$. Due to its similarity to SQL's
FULL JOIN, we omit a formal specification.

Formally, let $B'$ (resp. $B''$) be the bag of binding tuples
resulting from the clause FROM $l$ (resp. FROM $r$). Also, let
$l$ (resp. $r$) define variables $x_1, . . . , x_m$ (resp. $y_1, . . . , y_n$).
Then, FROM $l$ FULL CORRELATE $r$ on $c$ outputs all binding
tuples $b_i \parallel \langle b_j' \parallel \langle y_1', . . . , y_n' \rangle_{\text{from no match}},$ where $b_i' \in B'$, $b_j' \in B''$ and $c \rightarrow true$ within
binding env $b_i' \parallel \langle b_j' \parallel \Gamma_b \rangle$. In addition, for each $b_i'$ where there is
no $b_j'$ that matches (i.e. $c \rightarrow true$), FULL CORRELATE also
outputs a binding tuple $b_i' \parallel \langle y_1', . . . , y_n' \rangle_{\text{from no match}}$.
Conversely, for each $b_i'$ where there is no
matching $b_j'$, FULL CORRELATE also outputs a binding tuple
$\langle x_1' : @\text{from no match}, . . . , x_m' : @\text{from no match}\rangle$. (Figure 11 line 8) as a special case of
FROM $l$ INNER $r$, in which $r$ does not utilize variables of $l$.

(3a) Syntactic Sugar, SQL's Cartesian Product: For SQL compatibility, SQL++ supports FROM $l$, $r$ (Figure 11 line 8)
as a special case of FROM $l$ INNER $r$, in which $r$ does not utilize variables of $l$.

(3b) Syntactic Sugar, SQL's Join: Similarly, SQL++
supports SQL joins (lines 9-10) as a special case of
CORRELATE. LEFT JOIN is syntactic sugar for the core
LEFT CORRELATE as follows, and analogously, INNER JOIN is syntactic sugar for INNER CORRELATE. Finally, RIGHT
JOIN is syntactic sugar for LEFT JOIN, and FULL JOIN is
identical to FULL CORRELATE. (S) denotes syntactic sugar
and (C) denotes core SQL++.

(S) FROM $l$ AS $x$
(C) FROM $l$ AS $x$

LEFT JOIN $r$ AS $y$ ON $c$

LEfT CORRELATE ( FROM $r$ AS $y$
WHERE $c$
SELECT $y$ ) AS $y$

(3c) Syntactic Sugar, Flattening: Flattening (aka
unnesting) is commonly used within SQL-on-Hadoop and
NoSQL databases to range over nested collections. SQL++

| 1 | select_clause |
|---|----------------|
| 2 | SELECT ELEMENT $expr_query$ |
| 3 | SELECT ATTRIBUTE $expr_query$ : $expr_query$ |
| 4 | SELECT $expr_query$ (AS $attr\_name$)? |
| 5 | ( $expr_query$ (AS $attr\_name$)? ) |

Figure 15: BNF Grammar for SELECT Clause

| FROM readings: | (r: (gas: "no2", val: 0.7)). |
|---------------|-----------------------------|
| SELECT readings $r$ AS $y$ | \{(r: (gas: "no2", val: 0.7)).\} |
| FROM $l$ AS $x$ | \{ | $l$ AS $x$, $x.p$ AS $y$ | \} |

5.2 SELECT clause

The SELECT clause of SQL++ can output more than
SQL's collections of flat tuples. First, an output tuple can
contain other nested collections. Second, an output col-
lection can directly contain (i.e. without intervening tuples)
any arbitrary value, including scalars, collections, null
and missing. Finally, output tuples can be heterogeneous:
Missing attributes in the input can reflect to missing attri-
butes in the output. Indeed, an output tuple can have a
data-dependent number of attributes, where both the attri-
bute names and their values are provided by variables.

(1) SQL++ Core, Outputting a Collection: SELECT ELEMENT $e$ (Figure 15 line 2) is core SQL++ that outputs
a collection of arbitrary elements (both tuples and non-
tuples). Notice that the keywords are different from SQL's
plain SELECT, which is subsequently explained as syntac-
tic sugar for outputting specifically a collection of tuples.
The SELECT ELEMENT clause inputs a bag (resp. array, if
ORDER BY is present) of binding tuples $B_{\text{in}}$, and outputs a bag (resp. array) of values. Let $\Gamma$ be the binding env
of the enclosing SFW query. For each input binding
tuple $b \in B_{\text{in}}$, SELECT ELEMENT outputs a value $v$, where
$\Gamma \vdash e \rightarrow v$. Typically $e$ is a tuple/array/bag constructor (Figure 3 lines 24-26), but in general $e$ can be any expres-
sion. Figures 4 12 and 13 show respective examples of a
path expression, a plain variable and a tuple constructor
within SELECT ELEMENT.

When a missing is provided as an attribute value in
a constructor then the particular attribute/value pair is
omitted. Therefore, particular attribute omissions and
heterogeneities in the input can easily propagate to cor-
corresponding attribute omissions and heterogeneities in
the output. For example, consider the query FROM readings
AS $r$ SELECT ELEMENT ("co": $r$.co, "no2": $r$.no2) evaluated
on the readings heterogeneous collection of tuples of
Figure 4. The query output is identical to readings,
(2) SQL++ Core, Outputting a Variable-Width Tuple: SELECT ATTRIBUTE $e_0$,$e_v$ (Figure 13) line 3) outputs a tuple where the attribute name-value pairs are determined by the input bindings. For each input binding tuple $b$ of $\mathcal{B}_{\text{Input}}$, the output tuple contains an attribute $n \mapsto v$, where $b|\Gamma + e_b \mapsto n$ and $b|\Gamma + e_v \mapsto v$. An error occurs if $n$ is not a string. The attribute/value pair is omitted if $v$ is missing.

Figure 17 shows a collection-to-tuple pivoting example. The query output tuple contains two attributes whose names correspond to the evaluation of $r$.gas in the first binding tuple $(n02)$ and the second binding tuple $(co)$. Notice that Figure 17's query is the inverse of Figure 16's, and supports the complementary transposing/pivoting of the input collection’s elements (rows) into the output tuple’s attributes (columns).

(3) Core, Outputting Nested Values: Both SELECT ELEMENT and SELECT ATTRIBUTE can produce nested SFW queries (Figure 15, lines 2-3). For example, nesting a SFW query with SELECT ELEMENT within a parent SELECT ELEMENT clause will output nested collections. Figure 17 shows another example, where SELECT ATTRIBUTE is nested within a parent SELECT ELEMENT clause and outputs a collection of variable-width tuples. For each tuple in the input collection, the query outputs a tuple that contains only attributes that start with $co$.

(4) Syntactic Sugar, SQL's SELECT: For SQL compatibility, SQL++ supports SELECT $e_0$ AS $a_1$, ..., $e_v$ AS $a_n$ (Figure 15 lines 4-5) as syntactic sugar for SELECT ELEMENT ($a_1,e_1$, ..., $a_n,e_n$), i.e., a query that outputs a collection of tuples constructed with attributes $a_1$, ..., $a_n$. Furthermore, when expression $e_i$ is of the form $e' \cdot n$ (i.e., a path that navigates into tuple attribute $n$), SQL++ follows SQL in allowing AS $a_i$ (i.e., attribute names) to be optional. In this case, SELECT $e' \cdot n$ is equivalent to SELECT $e' \cdot n$ AS $n$.

5.3 GROUP BY clause

The SQL++ GROUP BY clause generalizes grouping to unify SQL’s semantics for 3 separate classes of functions: (i) ordinary functions (e.g., LIKE) that input scalars and output scalars (ii) aggregation functions (e.g., SUM) used with GROUP BY to input bags and output scalars (iii) window functions (e.g., ROW_NUMBER) used with OVER / PARTITION BY to input bags and output bags.

(1) Core, Group Variable: GROUP BY $e_1$ AS $x_1$, ..., $e_m$ AS $x_m$ (Figure 15 lines 9-10) is core SQL++ that creates groups. Each $e_i$ is a grouping expression, and each $x_i$ is a grouping variable. Let $\Gamma_1$ be the binding env of the enclosing SFW query. The bag of input binding tuples $\mathcal{B}_{\text{Input}}$ are partitioned into the minimal number of equivalence groups $B_1 \cdots B_n$, such that any two binding tuples $b, b' \in \mathcal{B}_{\text{Input}}$ are in the same equivalence group if and only if every grouping expression $e_i$ evaluates to the identical value $v_i$, i.e., $b|\Gamma_1 + e_i \mapsto v_i$ and $b'|\Gamma_1 + e_i \mapsto v_i$. For each group $B_j$ $(1 \leq j \leq n)$, the GROUP BY clause outputs a binding tuple $b_j = (x_1 : v_1, \ldots, x_m : v_m, \text{group} : B_j)$.

Notice that unlike SQL, the output binding tuple provides the partitioned input binding tuples in the special variable group, which can be explicitly utilized in subsequent HAVING, ORDER BY and SELECT clauses. Thus, a SQL++ query can perform complex computations on the groups, leading to results of any type (e.g., collections nested within collections). The explicit presence of groups in SQL++, while more general than SQL, also leads to simpler semantics than those of SQL, since the GROUP BY clause semantics are independent of the presence of subsequent functions in HAVING, ORDER BY and SELECT.

For example, Figure 15 shows a SQL++ query for counting and averaging the readings of each gas. The GROUP BY clause inputs 3 binding tuples, and uses $r$.gas to partition them into 2 groups. Within SELECT ELEMENT, the count function counts the elements in each group, whereas COUNT outputs the average $r$.num in each group. The GROUP BY clause is independent of COUNT and AVG, since both are general-purpose functions that can input arbitrary collections (i.e. not just groups).

(2) Syntactic Sugar, SQL's GROUP BY: For SQL compatibility, SQL++ allows grouping expressions $e_i$ in HAVING, ORDER BY and SELECT clauses, and omitting its corresponding grouping variable (i.e. AS $x_i$) in the GROUP BY clause. Thus, $e_i$ becomes syntactic sugar for $x_i$ within HAVING, ORDER BY and SELECT. For example, Figure 15 can equivalently use: SELECT ELEMENT ( gas : r.gas ... ).

(3) Syntactic Sugar, SQL's Aggregation Functions: SQL++ also supports syntactic sugar for aggregation functions. Suppose any of the SELECT, HAVING and ORDER BY clauses contains a function call $f(e)$, where $f$ is a SQL aggregation function such as SUM and AVG, and expression $e$ utilizes variables $v_1, \ldots, v_n$ that are defined in the FROM clause. Then, $\mathcal{F}(e)$ is syntactic sugar for $f(\text{SELECT } e' \text{ FROM group AS } p)$, where $e'$ results from sub-
stituting each $v_i$ in $e$ with $p.v_i$. For the special case where an aggregation function $f$ (such as `count`) needs the entire group as-is, $f(*)$ is syntactic sugar for $f(group)$. For example, Figure 18 can equivalently use: `count(*)` and `avg(r.num)`.

6. **SQL++ AS AN EXPRESSIVENESS BENCHMARK OF DATABASES**

To evaluate SQL++’s effectiveness as a configurable and unifying language, we utilize SQL++ config options as a formal expressiveness benchmark of five databases:

A. SQL as implemented in MySQL 5.6
B. CQL 3.2.0 of Apache Cassandra 2.1.2
C. the Aggregation Pipeline API of MongoDB 2.6.5
D. N1QL DP4 of Couchbase Server 3.0.1
E. AQL of AsterixDB 0.8.6.

All feature matrices have been empirically validated as follows. Classifying a feature as supported (✓) requires exemplary queries to return results consistent with both SQL++ semantics and the database’s documentation. Non-support of a feature (×) is substantiated by (i) explicit documentation of the restriction (ii) documentation that illustrates the lack of a feature (e.g. the absence of arrays among supported data types), and/or (iii) a query that throws an error. A feature that is partially supported (€) requires respective substantiation for both supported and unsupported parts. The full benchmark comprising sample data, queries and results, as well as links to supporting documentation and bug reports, is available at [http://forward.ucsd.edu/sqlpp](http://forward.ucsd.edu/sqlpp).

Table 3 presents feature matrices that classify the data model of the surveyed databases. We focus on named values that are stored within databases for two main reasons: (i) Since stored values are input by queries, they prominently affect whether query features interact with complex values, heterogeneity etc. (ii) Stored values are independent of query features (subsequently classified in Table 4), unlike the values of named views and query results. Parameters in

Table 4: Query Language Feature Matrices

| @stored_named | SQL | CQL | Mongo | N1QL | AQL |
|----------------|-----|-----|-------|------|-----|
| array          | ✓   | ✓   | ×     | ×    | ✓   |
| .bag           | ✓   | ✓   | ✓     | ✓    | ✓   |
| .tuple         | ✓   | ✓   | ✓     | ✓    | ✓   |
| .scalar        | ✓   | ×   | ×     | ✓    | ✓   |
| .null          | ✓   | ×   | ×     | ✓    | ✓   |
| .missing       | ✓   | ×   | ×     | ✓    | ✓   |

| @tuple_array   | SQL | CQL | Mongo | N1QL | AQL |
|----------------|-----|-----|-------|------|-----|
| support        | ✓   | ✓   | ✓     | ✓    | ✓   |
| absent         | ✓   | ✓   | ✓     | ✓    | ✓   |
| type_mismatch  | ✓   | ×   | m     | m    | n   |

| Array nav      | SQL | CQL | Mongo | N1QL | AQL |
|----------------|-----|-----|-------|------|-----|
| support        | ✓   | ✓   | ✓     | ✓    | ✓   |
| absent         | ✓   | ✓   | ✓     | ✓    | ✓   |
| type_mismatch  | ✓   | ×   | m     | m    | n   |

| @named_value   | SQL | CQL | Mongo | N1QL | AQL |
|----------------|-----|-----|-------|------|-----|
| restrict_to_from| ✓   | ✓   | ✓     | ✓    | ✓   |

| @tuple       | SQL | CQL | Mongo | N1QL | AQL |
|---------------|-----|-----|-------|------|-----|
| support       | ✓   | ×   | m     | m    | n   |
| complex       | ✓   | ✓   | ✓     | ✓    | ✓   |
| type_mismatch | ✓   | ×   | m     | m    | n   |
| null          | ✓   | ×   | ×     | ✓    | ✓   |
| null_eq_value | ✓   | ×   | ×     | ×    | ×   |
| null_eq_null  | ✓   | ×   | ×     | ×    | ×   |
| null_eq_missing| ✓  | ×   | ×     | ×    | ×   |
| missing_eq_value| ✓ | ×   | ×     | ×    | ×   |
| missing_eq_null| ✓ | ×   | ×     | ×    | ×   |
| missing_eq_missing| ✓ | ×   | ×     | ×    | ×   |

| @from         | SQL | CQL | Mongo | N1QL | AQL |
|----------------|-----|-----|-------|------|-----|
| support        | ✓   | ×   | m     | m    | n   |
| collection     | ✓   | ✓   | ✓     | ✓    | ✓   |
| at             | ✓   | ✓   | ✓     | ✓    | ✓   |
| bag_order      | ✓   | ×   | m     | m    | n   |
| coerce_null_to_collection| ✓ | ✓ | ✔ | ✔ | ✔ |
| coerce_missing_to_collection| ✓ | ✓ | ✔ | ✔ | ✔ |
| coerce_value_to_collection| ✓ | ✓ | ✔ | ✔ | ✔ |
| tuple          | ✓   | ✓   | ✓     | ✓    | ✓   |
| coeerce_null_to_tuple| ✓ | ✓ | ✔ | ✔ | ✔ |
| coeerce_missing_to_tuple| ✓ | ✓ | ✔ | ✔ | ✔ |
| no_match       | ✓   | ×   | ×     | ×    | ×   |
| inner_correlate| ✓   | ✓   | ✓     | ✓    | ✓   |
| left_correlate | ✓   | ✓   | ✓     | ✓    | ✓   |
| full_correlate | ✓   | ✓   | ✓     | ✓    | ✓   |
| cartesian_product| ✓ | ✓ | ✔ | ✔ | ✔ |
| inner_join     | ✓   | ×   | m     | m    | n   |
| left_join      | ✓   | ×   | m     | m    | n   |
| full_join      | ✓   | ×   | m     | m    | n   |
| inner_flatten  | ✓   | ×   | m     | m    | n   |
| outer_flatten  | ✓   | ×   | m     | m    | n   |

| @select        | SQL | CQL | Mongo | N1QL | AQL |
|----------------|-----|-----|-------|------|-----|
| support        | ✓   | ×   | m     | m    | n   |
| element        | ✓   | ×   | m     | m    | n   |
| attribute      | ✓   | ×   | m     | m    | n   |
| path           | ✓   | ×   | m     | m    | n   |
| function_call  | ✓   | ×   | m     | m    | n   |
| subquery       | ✓   | ×   | m     | m    | n   |

| @group_by      | SQL | CQL | Mongo | N1QL | AQL |
|----------------|-----|-----|-------|------|-----|
| support        | ✓   | ×   | m     | m    | n   |
| scalar         | ✓   | ×   | m     | m    | n   |
| complex        | ✓   | ×   | m     | m    | n   |
| null           | ✓   | ×   | m     | m    | n   |
| missing        | ✓   | ×   | m     | m    | n   |
| group_var      | ✓   | ×   | m     | m    | n   |

In addition, CQL supports arrays, MongoDB and
N1QL also support tuples, and AQL further supports bags. Table 4 classify query language features, and show how databases are more restrictive than SQL in some ways, but extend beyond SQL in others. We illustrate the classification convention using @tuple nav and @array nav (lines 1-6). Parameter support classifies whether tuple/array navigation is supported (√), unsupported (×) or partially supported (○). Parameters absent and type_mismatch are as defined in Figures 5 and 6. For brevity, the options of Figure 5 are abbreviated: null (n), missing (m) and error (x). In cases where a feature’s semantics vary under different circumstances, it is classified as inconsistent (∗). Finally, a feature is irrelevant (-) if it depends on other unsupported features. For example, since arrays are unsupported in SQL, array navigation is irrelevant.

@tuple nav: All databases support tuple navigation, but behave differently for absent attributes or navigation from non-tuples. MongoDB has inconsistent semantics: navigating from scalars return missing, yet navigating from arrays return other arrays. Notably, even though missing cannot be stored in MongoDB and N1QL (Table 4 lines 6, 12, 19, 26), it can be an intermediate result during query processing. @array nav: CQL supports arrays, but not array navigation. MongoDB only supports array navigation in the WHERE clause, in which an expression language different from that of other clauses has been implemented. In both N1QL and AQL, the contrary, array navigation is fully supported and behaves symmetrically with tuple navigation.

@named_value: All databases follow SQL such that a named value (which is always a bag) is restricted to being usable in arbitrary expressions (such as table1 = table2).

@eq: The parameters are defined in Figures 8 and 9. In SQL, (i) complex equality is irrelevant due to the named value restriction above (ii) 1=null→null (iii) null=null→null (iv) 1=‘a’→false, a peculiarity of MySQL unlike other SQL databases. All other features are irrelevant as complex equality and missing are unsupported. In CQL, 1=null→false (unlike SQL), and null=null is precluded by other limitations (details in the extended version). In MongoDB, equality is cleanly supported through identity comparisons of null/missing. In N1QL, equality on null/missing is broadly inconsistent, as N1QL violates the equivalence xmy ⇔ [x]=y. For example, null_eq_scalar is inconsistent because 1=null (like SQL), whereas [1]=null→false. Finally, AQL follows SQL equality. Notice that four-valued logic parameters are irrelevant in the 5 databases and have thus been omitted: either complex equality is unsupported, or it is supported but shallow equality does not return null/missing (see N1QL inconsistency above).

@from: The parameters classify FROM clause features (Section 5.1), and options for bag_order, no_match, coerce_null_to_collection etc. are as specified in Figure 11. SQL supports ranging over a named collection, over a subquery, Cartesian product, INNER JOIN, and LEFT JOIN. MySQL also supports a counter for the iteration order over a bag (lines 20-21). CQL, MongoDB and N1QL are less composable. CQL only supports ranging over a named collection. Likewise for MongoDB, but with the addition of INNER FLATTEN. N1QL further supports joins, but, unlike SQL, restricts them to between primary/foreign keys, and uses missing for no-matches. Notably, AQL exceeds SQL by supporting the more general INNER CORRELATE. However, AQL does not support LEFT/FULL JOIN. No database supports yet coercions, ranging over tuple attributes, LEFT CORRELATE, or OUTER FLATTEN.

@select: SQL and CQL can neither output nor store heterogeneous values. Conversely, MongoDB, N1QL and AQL can both output and store heterogeneous values. All databases follow SQL in supporting paths and functions within SELECT. However, many CQL functions (e.g., =, AND) are supported only in WHERE but not SELECT. Only N1QL and AQL support SELECT ELEMENT to output a collection of non-tuples, and only AQL fully supports subqueries to output nested collections. A SQL subquery must output scalars, and a N1QL subquery must be correlated with the outer query by primary/foreign keys. No database supports yet SELECT ATTRIBUTE to output a tuple of variable-width.

@group by: CQL does not support GROUP BY. All other databases support grouping all values in their data models. MongoDB, N1QL and AQL also extend beyond SQL to support the group variable. Counterintuitively, MongoDB groups null and missing together as if they were identical, which is contrary to @eq, null_eq_missing=false (line 13).

7. CONCLUSIONS AND FUTURE WORK

We described the FROM, WHERE, GROUP BY and SELECT of the configurable, unifying and semi-structured SQL++ query language. SQL++ has picked salient features (composability, semantics without need for schema) from prior query languages - most notably XQuery and OQL. Unlike XQuery’s data model, SQL is simple and easily adjusted to be backwards-compatible with SQL, as shown. This enables the SQL++ semantics to be significantly shorter than XQuery, while also offering additional functionalities such as a generalization of SQL’s (left) outerjoin and a generalization of SQL’s grouping. Furthermore, SQL++ avoids unnecessary extensions over SQL. Rather, it achieves many of its additional capabilities simply by removing semantic restrictions of SQL.

We also showed how appropriate settings of the configuration options morph SQL++ into SQL or any of four NoSQL query languages. The extended version [11] provides the syntax and semantics of ORDER BY, LIMIT, set operators (UNION etc) and schemas. Furthermore, it provides a SQL++ algebra that (in the spirit of SQL++) is highly compatible with the set-at-a-time processing that SQL engines (based on SQL algebra) use. An earlier extended version [13] provides the configuration options for an additional six semi-structured databases.

While the NoSQL space is bound to change, the configuration options technique (including the proposed macro options) provides a standing contribution in (a) itemizing and rationalizing the options of a language designer and (b) describing the capabilities and semantics of NoSQL databases, while abstracting away superficial syntactic differences.

The extended version [11] will be expanded to describe (using the configuration options technique) the capabilities of selected SQL databases that have recently introduced JSON columns (e.g., [10]) and respective UDFs. Instead of forcing users to learn multiple new primitives (via UDF’s) for ranging over various patterns of JSON data and/or constructing various patterns of JSON data, SQL++ lets FROM variables to freely range over any type of data: attribute/value pairs, array elements and index positions in either correlated or uncorrelated fashion, and based on this

95FULL JOIN is anomalously unsupported in MySQL.
freedom delivers high expressive power. Similarly, SQL++ allows the SELECT clause to seamlessly create heterogenous and semi-structured JSON by the mere reuse of nested queries.

Acknowledgements

We thank: Michalis Petropoulos (Pivotal), Yupeng Fu (Palantir) and Romain Vernoux (INRIA) for early contributions to the design and implementation of FORWARD’s SQL++ middleware processor; Ashna Jain (UCSD) and Romain Vernoux for their contributions on the experiments. Aditya Avinash for his contributions on the SQL++ reference implementation. Gerald Sangudi (Couchbase), Mike Carey (UCI) and Yannis Katsis (UCSD) for numerous comments and suggestions; Yingyi Bu (UCI) and Jules Testard (UCSD) for the ASTERIX SQL++ implementation. Jacques Nadeau (MapR) for pointing us to use cases requiring iterating over attribute/value pairs; (again) Gerald Sangudi for his work on formally reducing N1QL to SQL++; Arun Patnaik and Pritesh Maker (Informatica) for discussions on SQL++ as a unifying middleware query language. Finally, Informatica and NSF for their financial support of this work.

8. REFERENCES

[1] S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors. Nested Relations and Complex Objects, volume 361 of Lecture Notes in Computer Science. Springer, 1989.

[2] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O2 object-oriented database system. In DBPL, pages 122–138, 1989.

[3] A. Behm et al. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. Distributed and Parallel Databases, 29(3):185–216, 2011.

[4] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. SIGMOD Record, 29(1):68–79, 2000.

[5] BSON. http://bsonspec.org/

[6] A. Deutsch, M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. A query language for XML. Computer Networks, 31(11-16):1155–1169, 1999.

[7] E. Dumbill. What is Big Data?, 2012. http://strata.oreilly.com/2012/01/what-is-big-data.html

[8] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non first normal form relations. In PODS, pages 124–138. ACM, 1982.

[9] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. Schemasql - A language for interoperability in relational multi-database systems. In VLDB’96, pages 239–250, 1996.

[10] Z. H. Liu and D. Gawlick. Management of flexible schema data in RDBMSs: Opportunities and limitations for NoSQL. CIDR, 2015.

[11] K. W. Ong and Y. Papakonstantinou. The SQL++ query language: Configurable, unifying and semi-structured. CoRR, abs/1405.3631, 2015. http://arxiv.org/abs/1405.3631

[12] K. W. Ong, Y. Papakonstantinou, and G. Sangudi. A reduction of N1QL into SQL++ core, 2015. http://db.ucsd.edu/pubsFileFolder/382.pdf

[13] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. SQL++: An expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases, 2014. http://db.ucsd.edu/pubsFileFolder/375.pdf

[14] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML query language. W3C candidate recommendation, 2013. http://www.w3.org/TR/2013/PR-xquery-30-20131022/

[15] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. ACM Trans. Database Syst., 13(4):389–417, 1988.