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. (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.

Their evolving query languages fall short of full-fledged semi-structured query language capabilities 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 JSON 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). 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, XQuery,..
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 clause ranges 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.
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 is 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 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++.

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, 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 idiosyncracies 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 [19] 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
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 contain 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.

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,
Figure 4: Binding tuples in an SFW Query

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)Γ comprising a config env Γc and a variable binding env Γb (or binding env). (We may omit the subscript b in examples where the environment is clearly a variable binding environment.) A binding env is a binding tuple \((x_1 : v_1, \ldots, x_n : v_n)\), where each \(x_i\) is a variable name that is unique, and binds to the SQL++ value \(v_i\). Notice that a binding tuple is structurally a SQL++ tuple. The characterization “binding” pertains to its use in the semantics, rather than structural differences.

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

The notation Γb ⊨ q → v denotes that the SQL++ query q evaluates to the value v within Γb, i.e. when every free variable of q is instantiated by its binding in Γb. For example, \((x : 5, y : 3) ⊨ (x + y)/2 → 4\).

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 Γb = \((v_1 : v_1, \ldots, v_m : v_m)\) where the pairs \(v_i : v_i\) are the database’s named values \(v_i\) and their names \(v_i\). 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 Γb 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.)

**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 (Γ in the example). Thereafter the FROM clause outputs the bag of binding tuples \(B_{\text{FROM}}^{\text{out}}\) (which has 3 binding tuples in the example). In each binding tuple of \(B_{\text{FROM}}^{\text{out}}\), each variable of the FROM clause is bound to a value. There are no restrictions that a variable binds to homogenous 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 (\(B_{\text{WHERE}}^{\text{out}} = B_{\text{FROM}}^{\text{out}}\)), 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 \(b|\Gamma_b\), where Γb 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 \(B_{\text{WHERE}}^{\text{out}}\). The binding env of the first evaluation is \((r : \{co : 2.2\})\|\Gamma_b = (r : \{co : 2.2\}, \text{readings :} \ldots, \text{max :} 2)\), 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 (resp. array, if ORDER BY is present) of binding tuples, and outputs the SFW query’s result, which is a bag (resp. array) with exactly one element for each input binding tuple. In the example, the SELECT ELEMENT expression \(r.no2\) is evaluated once for each of the 2 input binding tuples of \(B_{\text{SELECT}}^{\text{out}}\). The first evaluation is within the binding env \((r : \{co : 1.8, no2 : 0.7\})\|\Gamma_b = (r : \{co : 1.8, no2 : 0.7\})\|\Gamma_b\), and outputs the element \(0.7\).

Finally, notice that the above discussion of SFW queries did not capture the set operators UNION, INTERSECT and EXCEPT. As is the case with SQL semantics too, the coordination of ORDER BY with the set operators is treated as a special case, which is described in the extended version.

**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.

---

9 SQL++ also supports a syntax improvement where SELECT is optionally written as the last clause.
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 a attribute and a 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 4.1) and a complex use case in equality (Section 4.3). The support/non-support of query features described in Section 4 are also specified by config options. Section 4.1 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 @p (Figure 3) configures (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 \( \Gamma_g \) is a config tuple \((p_1 : o_1, \ldots, p_m : o_m)\), where each parameter \( p_i \) maps to an option \( o_i \).

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 \@p is expanded to a set of coordinated config annotations \@p. 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 may be overridden by annotations that apply to its subqueries. Suppose \@p(q: o) is evaluated in env \( \Gamma = (\Gamma_b, \Gamma_c) \). (The query \( q \) may be a subquery of an enclosing query.) Then \( q \) is evaluated in \( \Gamma' = (\Gamma_b, (\@p.o) \mid \Gamma_c) \). That is, \( q \) is evaluated in a context where \( p \) is set to \( o \), regardless of whether it could have been configured differently in an enclosing context. Similar to binding tuples, given two config tuples \( c \) and \( c' \), their concatenation \( c \cdot c' \) retains the mapping of \( c \) if parameter \( p \) is mapped in both \( c \) and \( c' \). Therefore, two different parts of a query may correspond to configurations of different databases.

The ability to configure subqueries with different annotations becomes handy when SQL++ is used as a federated query language. In such cases, if a subquery that is pushed to a certain database is configured according to the options of such database then it can indeed be pushed. In effect, the task of a federated query processor (such as FORWARD) is to rewrite the user query into one where all subqueries that are pushed to an underlying database have been configured according to the configuration options of such database.

4.2 Path Navigation

A tuple path navigation \( t.a \) from the tuple \( t \) to its attribute \( a \) (Figure 4) returns the value of \( a \). An array path navigation \( a [i] \) returns the \( i \)-th element of the array \( a \) (Figure 5). Notice that consecutive tuple/array navigations

As syntactic sugar, parameters of the same group only need to specify the group once (lines 28–32), i.e. \@p is sugar for \@p. Consecutive config annotations can also omit parentheses (line 4).

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

\[
\begin{align*}
\text{path_param} & \rightarrow \text{absent} : (\text{null}\text{missing}\text{error}) \\
\text{type_mismatch} & \rightarrow (\text{null}\text{missing}\text{error})
\end{align*}
\]

### Figure 6: Path Evaluation Functions for \( t.a \) and \( a[i] \)

### Figure 7: Configuring Tuple/Array Navigation
Table 1: @nav.failure Macro Expansion

<table>
<thead>
<tr>
<th>@nav.failure</th>
<th>error</th>
<th>null</th>
<th>missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>@tuple_nav.absent</td>
<td>error</td>
<td>null</td>
<td>missing</td>
</tr>
<tr>
<td>@tuple_nav.type_mismatch</td>
<td>error</td>
<td>null</td>
<td>missing</td>
</tr>
<tr>
<td>@array_nav.absent</td>
<td>error</td>
<td>null</td>
<td>missing</td>
</tr>
<tr>
<td>@array_nav.type_mismatch</td>
<td>error</td>
<td>null</td>
<td>missing</td>
</tr>
</tbody>
</table>

(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 when it is a tuple that does not have an a attribute. Multiple options are possible for these cases and, indeed, multiple options have been taken by the surveyed databases. Figure 3 specifies the evaluation function of t.a and the function of a[x], which utilize parameters from the @tuple_nav and array_nav parameter groups (Figure 3 lines 29-30). The absent parameter (Figure 1 line 2) specifies the returned value when an attribute/array element is absent: null, missing, or throw an error. The type_mismatch 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 @tuple_nav.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 @tuple_nav/array_nav 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 @tuple_nav (absent: error, type_mismatch: 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 1: (i) error specifies a strict form of path navigation (as in SQL), such that navigation failures result in errors thrown. Notice, it expands to @tuple_nav.absent: error, @tuple_nav.type_mismatch: 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 xy 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 \( x = \text{null} \rightarrow \text{null} \). 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 8 shows the parameters and options of the equality parameter group, which are utilized in the equality evaluation function of Figure 9. For deep equality comparisons between complex values in \( f_{\text{array}}, f_{\text{bag}} \) and \( f_{\text{tuple}} \), additional subtleties are introduced by null and missing elements/attributes. Consider \( f_{\text{array}} \) using two arrays \( x \) and \( y \) that each has \( n \) elements, and all elements are either scalar or complex values. Then, each pairwise comparison \( x[i] = y[i] \) evaluates to true/false, and array equality is simply the result of the operator \( \text{eq} \), which returns the conjunction of these true/false values. However, \( x[i] = y[i] \) can also evaluate to null/missing \( \text{eq} \) must be configured to also produce a result for the logical conjunction between null/missing true/false. This configuration is achieved by the five parameters in Figure 8 lines 9 to 13, which extend SQL’s 3-valued logic.

Figure 10 shows two different configurations and the different results that the same query \( r = r \) has for each configuration. The left example follows languages (such as Javascript) which consider null a sentinel value that is equal to itself. null = null evaluates to true, thus the array \( r \) deep equals itself. The right example follows SQL’s 3-valued logic, which uses null to represent an unknown value. Since the array contains an unknown value, the result of deep equality is also unknown. Therefore, to support equality that behaves reasonably with respect to null, we recommend language designers utilize the macro config annotation @unknown.value. In the case of databases that support null and do not support missing, the two sensible options are sentinel and logic. If a language designer chooses the 3-valued, she has implicitly chosen that null = null is null but also that null and true is null.

In the presence of missing, the @unknown.value must specify its behavior in comparison. In the sentinel approach, the missing will behave as yet another sentinel value. In the logic-based approach, we recall that missing is introduced as “inapplicable”, while null stands for the currently unknown. In such case, the logic approach is expanded as shown in Table 2. Notice that Table 2 provides a pure sentinel approach and an unknown/inapplicable logic approach. One may still consider other possibilities and this is why macro operate as guidelines, rather than definitive options.

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 the attribute/value pairs of tuples (a feature reminiscent of SQL extensions for dynamic schema introspect [2]). Furthermore, FROM 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 [8,15]) 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 SQL++ FROM clause and corresponding config parameters. The syntax and semantics of FROM are defined inductively with three cases.

1. **from_clause**
   - from_item
   - | expr_query AS var (AT var)?
   - | expr_query AS (var | var)
   - | from_item (INNER/LEFT OUTER)? CORRELATE? from_item
   - | from_item FULL OUTER? CORRELATE? from_item ON expr_query
   - | from_item, from_item
   - | from_item (INNER/LEFT RIGHT) JOIN from_item ON expr_query
   - | (INNER/OUTER) FLATTEN(expr_query AS var, expr_query AS var)

2. **from_item**
   - | bag_order: (counter/null/missing/error)
   - | coerce_null_to_collection : (singleton/empty/error)
   - | coerce_missing_to_collection : (singleton/empty/error)
   - | coerce_value_to_collection : (singleton/empty/error)
   - | coerce_null_to_tuple : (empty/error)
   - | coerce_missing_to_tuple : (empty/error)
   - | no_match: (null/missing)
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 e AS x (AT y)? 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 com-

posability, as e can also be a variable, path, function call, value constructor or value literal (Figure 3 lines 20-27).

Let e → c, where c is a collection with n elements u₁, . . . , uₙ. For each uᵢ, j = 1 . . . n, the FROM clause outputs a binding tuple (x : uᵢ, y : vᵢ), where vᵢ is the ordinal position of uᵢ in c when c is an array, or a value specified by the config parameter @from.bag_order (Figure 11 line 14) when c is a bag. A language designer sets @from.bag_order to counter to produce a random order and assign sequential numbers accordingly. Alternately, uses null or missing as sentinel values indicating bag positions are meaningless. As in SQL, the SQL++ FROM clause produces an unordered bag of binding tuples. An output order can be later imposed by ORDER BY, which may utilize the position variable y.

For example, Figure 12 shows a SQL++ query that reverses an array. The FROM clause defines element variable r and position variable p, and outputs 3 binding tuples that respectively bind r to elements and p to positions. The ORDER BY clause uses p to sort the binding tuples in descending order of their original positions in the array, and the SELECT ELEMENT clause outputs the reversed array elements.

Since SQL++ is semi-structured, it is possible that e evaluates into a value c that is not a collection. In such cases, the language designer may choose to coerce c into a collection, according to one of the "coerce" configuration options (Figure 11 lines 15-17). For example, if @from.coerce_null_to_collection is set to singleton, then when c is null, c is coerced into \{null\}. If set to empty, then null is coerced into the empty collection \{\}. If c is a (scalar or tuple) value then the designer may choose to coerce it into the collection \{c\}, by setting @from.coerce_value_to_collection to singleton.

1b) SQL++ Core, Ranging over Tuple Attributes: FROM t AS \{a:v\} enables ranging over the attribute name-value pairs of a tuple (Figure 11 line 5). When expression t evaluates to a tuple \{u₁ : v₁, . . . , uₙ : vₙ\}, FROM outputs the bag of binding tuples \{(a : u₁, v : v₁) . . . (a : uₙ, v : vₙ)\}.

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 Γ. Let Bᵢ be the bag of binding tuples that would result from the clause FROM l. Then, FROM l INNER CORRELATE r outputs all binding tuples \{bᵢ∥bᵢᵣ\}, where \b₁ ∈ Bᵢ, \bᵢᵣ ∈ Bᵢᵣ, and Bᵢᵣ is the bag of binding tuples that would result from evaluating FROM r within the binding env \bᵢᵣΓ.

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:a\} would result in binding tuples \b₁, \b₂, \b₃ each with variables g and a. Evaluating the right hand side a AS v in the context of \b₁ = \{g : "co", a : \{0.7, \{0.5, 2\}\}\} results into \b₁ᵣ = \{(v : 0.7), (v : \{0.5, 2\})\}. Consequently the FROM clause outputs the first 2 binding tuples of Figure 14. Likewise \b₂ᵣ = \{g : no2, a : ["repair"]\}, leads to \b₂ᵣᵣ = \{(v: ["repair"])\} which leads to FROM outputting the 3rd binding tuple. Since \b₃ᵣᵣᵣ = \{g : s02, a : [1]\} results in an empty \b₃ᵣᵣᵣ, there is no binding tuple that corresponds to s02.
expression \( a \text{ AS } v \), in general \( r \) can be a complex SFW query. For example, we could replace the \( a \text{ AS } v \) in Figure 14 with the following subquery, which would correlate each \( q \) a pair only with the top-1 element of each collection.

\[
\text{(FROM } a \text{ AS } x \\
\text{ORDER BY } x \\
\text{LIMIT } 1 \\
\text{SELECT ELEMENT } x) \text{ AS } v
\]

(2b) Core, \textsc{left outer correlate}: In \textsc{from} / \textsc{left outer correlate} \( r \) (Figure 11 line 6), \( r \) can again utilize variables defined by \( l \). (For brevity one may simply write \textsc{left \( r \), omitting either or both of the \textsc{outer} and \textsc{correlate} keywords.) Similar to SQL's \textsc{left join}, the \textsc{left} outputs binding tuples derived from \( l \), even if there are no matching binding tuples from \( r \). Formally, let \( B^l \) be the bag of binding tuples that results from the clause \textsc{from} \( l \). Then, from \textsc{from} \( l \text{ left } r \) outputs all binding tuples that \textsc{from} \( l \text{ inner } r \) would output. In addition, for each \( b^l_i \in B^l \) such that \( b^l_i \) is an empty bag, \textsc{left} also outputs a binding tuple \( b^r_i| \{x^l_i : @\text{from.no.match} . . . , x^n_i : @\text{from.no.match}\} \), where \( x^l_1 . . . x^n_i \) are variables defined by \( r \), and each variable is bound to the value configured by \textsc{from.no.match} (line 20). Since \textsc{left} generalizes SQL's \textsc{left join} (as discussed below), achieving SQL compatibility requires configuring \textsc{from.no.match} to \textsc{null}. Alternatively, a language designer may also choose configuring to \textsc{missing}. The extended version argues that the latter is preferable because it distinguishes the kinds of information absence (when the language designer wants \textsc{null} to mean "unknown" and \textsc{missing} to mean "inapplicable") and (2) enables simple rewritings that correlate \( r \) and open the gates to set-at-a-time processing, of the kind accomplished by SQL processors.

For example, suppose Figure 14 uses \textsc{inner} instead of \textsc{left}. Assuming \textsc{from.no.match} is configured to \textsc{null}, the query will output the additional tuple: \((\text{gas}::\text{so2}, \text{val}::\text{null})\).

(2c) Core, \textsc{full correlate}: From \textsc{from} \textsc{full correlate} \( r \) on \( c \) (Figure 11 line 7) resembles SQL's \textsc{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 \textsc{full join}, we omit a formal specification.

Formally, let \( B^l \) (resp. \( B^r \)) be the bag of binding tuples resulting from the clause \textsc{from} \( l \) (resp. \textsc{from} \( r \)). Also, let \( l \) (resp. \( r \)) define variables \( x^l_1 . . . , x^l_n \) (resp. \( y^r_1 . . . , y^r_n \)). Then, from \textsc{from} \textsc{full correlate} \( r \) on \( c \) outputs all binding tuples \( b^l_i|b^r_j \), where \( b^l_i \in B^l \), \( b^r_j \in B^r \) and \( c \rightarrow \text{true} \) within binding env \( b^l_i|b^r_j|\textsc{env}^r \). In addition, for each \( b^l_i \) where \( b^l_i \) is an empty bag, \textsc{full} also outputs a binding tuple \( b^r_i|\{y^r_1 : @\text{from.no.match} . . . , y^r_n : @\text{from.no.match}\} \). Conversely, for each \( b^r_j \) where there is no matching \( b^l_i \), \textsc{full} also outputs a binding tuple \( \{x^l_1 : @\text{from.no.match} . . . , x^n_i : @\text{from.no.match}\}|b^r_j \).

(3a) Syntactic Sugar, SQL's Cartesian Product: For SQL compatibility, SQL++ supports \textsc{from} \( l \text{ } r \) (Figure 11 line 8) as a special case of \textsc{from} \( l \text{ inner } r \), 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 \textsc{correlate}. \textsc{left join} is syntactic sugar for the core \textsc{left correlate} as follows, and analogously, \textsc{inner join} is syntactic sugar for \textsc{inner correlate}. Finally, \textsc{right join} is syntactic sugar for \textsc{left join}, and \textsc{full join} is identical to \textsc{full correlate}. (S) denotes syntactic sugar and (C) denotes core SQL++.
by the input bindings. For each input binding tuple where the attribute name-value pairs are determined.

Figure 16: Using SELECT ATTRIBUTE to Output a Variable-Width Tuple

Figure 17: SELECT ATTRIBUTE nested in SELECT ELEMENT

(2) SQL++ Core, Outputting a Variable-Width Tuple: SELECT ATTRIBUTE $e_n : c_n$ (Figure 15, line 3) outputs a tuple where the attribute name-value pairs are determined by the input bindings. For each input binding tuple $b$ of $B_{SELECT}$, the output tuple contains an attribute $n : v$, where $b|\Gamma \vdash e_n \rightarrow n$ and $b|\Gamma \vdash e_v \rightarrow v$. An error occurs if $n$ is not a string. The attribute/value pair is omitted if $v$ is missing.

Figure 18 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 (no2) and the second binding tuple (co). Notice that Figure 15’s query is the inverse of Figure 18’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 comprise 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 $e_1 \ AS a_1, \ldots , e_n \ AS a_n$ (Figure 15, lines 4-5) as syntactic sugar for SELECT ELEMENT \{$a_1 : e_1, \ldots , a_n : e_n$\}. i.e., a query that outputs a collection of tuples constructed with attributes $a_1, \ldots , a_n$. Furthermore, when expression $e_i$ is of the form $e' . n$ (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 $e' . n$ is equivalent to SELECT $e' . n$ 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 $e_1 \ AS x_1, \ldots , e_m \ AS x_m$ (Figure 18, 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 $\Gamma_i$ be the binding env of the enclosing SFW query. The bag of input binding tuples $B_{GROUP}$ are partitioned into the minimal number of equivalence groups $B_1 \ldots B_n$, such that any two binding tuples $b, b' \in B_{GROUP}$ are in the same equivalence group if and only if every grouping expression $e_i$ evaluates to the identical value $v_i$, i.e., $b|\Gamma_i \vdash e_i \rightarrow v_i$ and $b'|\Gamma_i \vdash e_i \rightarrow v_i$. For each group $B_j$ (1 ≤ $j$ ≤ $n$), the GROUP BY clause outputs a binding tuple $b_j = (x_1 : v_1, \ldots , x_m : v_m, \ group : B_j)$.

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 18 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 $avg$ 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 group).

(2) Syntactic Sugar, SQL’s GROUP BY: For SQL compatibility, SQL++ supports using a grouping expression $e_i$ in HAVING, ORDER BY and SELECT clauses, and omitting its corresponding grouping variable (i.e. AS $x_i$) in the GROUP
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 exemplar 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.

Table 3: Data Model Feature Matrices

<table>
<thead>
<tr>
<th>@stored_named</th>
<th>SQLb</th>
<th>CQL</th>
<th>Mongo</th>
<th>N1QL</th>
<th>AQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. array</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2. bag</td>
<td>x</td>
<td>√</td>
<td>x</td>
<td>√</td>
<td>x</td>
</tr>
<tr>
<td>3. tuple</td>
<td>x</td>
<td>√</td>
<td>x</td>
<td>√</td>
<td>x</td>
</tr>
<tr>
<td>4. scalar</td>
<td>x</td>
<td>√</td>
<td>x</td>
<td>√</td>
<td>x</td>
</tr>
<tr>
<td>5. null</td>
<td>x</td>
<td>√</td>
<td>x</td>
<td>√</td>
<td>x</td>
</tr>
<tr>
<td>6. missing</td>
<td>x</td>
<td>√</td>
<td>x</td>
<td>√</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 4: Query Language Feature Matrices

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 @stored_named (lines 1-6) classify whether a stored named value can be an array, bag, tuple, scalar, null or missing. For all stored bags, independently of whether they are named or nested, @stored_bag (lines 7-13) classifies whether each type of element, as well as heterogeneous elements, are supported. @stored_array and @stored_tuple are analogously defined. Collectively, the parameters classify the values that a stored named value can recursively contain, wherein unsupported features indicate a lack of composability.

@stored_named: Following SQL, all databases restrict stored named values to bags (i.e. a SQL table).

@stored_bag: CQL, MongoDB and N1QL follow SQL in
restricting bags to only comprise tuples. AQL is classified with partial support, as the restriction applies to named bags but not nested bags. Unlike SQL and CQL, bags can be heterogeneous in MongoDB, N1QL and AQL.

@stored_array: SQL does not support arrays. A CQL array supports up to 65k scalar elements of the same type (e.g. string). MongoDB/N1QL arrays are heterogeneous and contain arrays, tuples, scalars and nulls. An AQL array further supports bag elements.

@stored_tuple: Tuple attributes can be scalars and nulls in SQL. 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 Figure 4 and Table 4. For brevity, the options of Figure 5 are abbreviated: null (n), missing (m) and error (×). 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 3 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, on 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 ranged over within the FROM clause, instead of being usable in arbitrary expressions (such as Table 1 = Table 2).

@eq: The parameters are defined in Figures 4 and 2. In SQL, (i) complex equality is irrelevant due to the named value restriction above (ii) l=null->null (iii) null=null->null (iv) l='a'->false, a peculiarity of MySQL unlike other SQL databases. All other features are irrelevant as complex equality and missing are unsupported. In CQL, l=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 x=my ⇔ [x]=[y]. For example, null_eq_scalar is inconsistent because 1=null->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 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, JSON 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 13 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 algebras) 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

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


