Noname manuscript No.
(will be inserted by the editor)

The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases

Kian Win Ong · Yannis Papakonstantinou · Romain Vernoux

Received: date / Accepted: date

Abstract SQL-on-Hadoop, NewSQL and NoSQL databases provide semi-structured data models (typically JSON based) and respective query languages. Lack of formal syntax and semantics, idiomatic (non-SQL) language constructs and large variations in syntax, semantics and actual capabilities pose problems even to database experts: It is hard to understand, compare and use these languages. It is especially tedious to write software that interoperates between two of them or an SQL database and one of them.

Towards solving these problems, first we formally specify the syntax and semantics of SQL++. It consists of a semi-structured data model (which extends both JSON and the relational data model) and a query language that is fully backwards compatible with SQL. SQL++ is "unifying" in the sense that it is explicitly designed to encompass the data model and query language capabilities of current SQL-on-Hadoop, NoSQL and NewSQL databases.

Then, we itemize fifteen SQL++ data model and query language features and benchmark eleven databases on their support of the multiple options associated with each feature, leading to feature matrices and commentary. Each feature matrix is the result of empirical validation through sample queries. Since SQL itself is a subset of SQL++, the SQL-aware reader will easily identify in which ways each of the surveyed databases provides more or less than SQL. The eleven databases are Hive, Jaql, Pig, Cassandra, JSONiq, MongoDB, Couchbase, SQL, AsterixDB, BigQuery and UnityJDBC. They were selected due to their market adoption or because they present cutting edge, advanced query language abilities.

Finally, we briefly discuss the use of SQL++ as the query language of the FORWARD virtual database query processor, which executes SQL++ queries over SQL and non-SQL databases and the use of SQL++ in the FORWARD application framework, which enables rapid development of live reports and interactive applications on SQL and non-SQL databases. FORWARD provides a proof-of-concept of SQL++'s applicability as a unifying data model and query language.

This work was supported by NSF DC 0910820, NSF III 1219263, NSF IIS 1237174 and Informatica grants. The grants' PI is Prof. Papakonstantinou who is a shareholder of an entity that commercializes some results mentioned in this research.

Kian Win Ong
E-mail: kianwin@cs.ucsd.edu

Yannis Papakonstantinou*
E-mail: yannis@cs.ucsd.edu

Romain Vernoux
E-mail: rvernoux@cs.ucsd.edu

1 Introduction

Numerous databases marketed as SQL-on-Hadoop, NewSQL [30] and NoSQL have emerged to catalyze Big Data applications. These databases generally support the 3Vs [11]. (i) Volume: amount of data (ii) Velocity: speed of data in and out (iii) Variety: semi-structured and heterogeneous data. As a result of differing use cases and design considerations around the Variety requirement, these new databases have adopted semi-structured data models that vary among each other. Their query languages have even more variations. Some variations are due to superficial syntactic differences. Some variations arise from the data model differences. Finally, other variations are genuine differences in query capabilities.

In this setting, even researchers and practitioners with many years of SQL database experience face problems in two areas:

1. *Comprehension*: Significant effort is needed to understand, compare and contrast the semi-structured data models and query languages of these novel databases. The informal (and often underspecified) syntax and semantics of the provided query languages make comprehension even harder or impossible, as it becomes apparent from the avalanche of syntax and semantics questions in online forums.
2. *Development*: It is difficult to write software that retrieves data from multiple such databases, given the different data models, different query syntaxes and the (often subtly) different query semantics. These interoperability issues occur frequently in practice, for example, whenever an organization adopts one of these new databases and then builds applications that need integrated access to data stored in the new database and in its existing SQL databases.

Towards solving the above problems, we formally specify the syntax and semantics of SQL++, which is a unifying semi-structured data model and query language that is designed to encompass the data model and query language capabilities of NoSQL, NewSQL and SQL-on-Hadoop databases. The SQL++ semantics stands on the shoulders of the extensive past work from the database R&D community in non-relational data models and query languages: OQL [2], the nested relational model and query languages [15, 28, 1] and XQuery (and other XML-based query languages) [27, 10, 5].

SQL++ is an extension to SQL and is backwards-compatible with SQL. This choice was made in order to facilitate the SQL-aware audience in two aspects: First, since many surveyed databases do not support the entirety of standard SQL capabilities, the provided comparisons explain the extent to which each surveyed database supports the SQL capabilities. Second and most importantly, the reader will understand in what ways semi-structured data models and query languages extend SQL's capabilities, understand in which ways these extensions may relate to each other, and obtain an overview on which surveyed databases support these extensions.

Then we itemize fifteen SQL++ data model and query language features and benchmark eleven databases on their support of the multiple options associated with each feature. For this benchmark, we cover the most popular SQL-on-Hadoop, NoSQL and NewSQL databases from DB-Engines [8] (a popularity tracker for database engines) and industry surveys [12, 17]. We have also selected research-oriented databases that push the agenda on query languages for JSON or JSON-like data, as these databases gravitate towards more sophisticated and complete query capabilities.

The benchmark's results are presented through fifteen feature matrices and additional analysis/commentary that classify each database's data model and query language capabilities as a subset of SQL++. The matrices further decompose each feature into as many as eleven constituent sub-features and options, in order to facilitate fine-grained comparisons across different data models and languages. Besides providing information on supported and unsupported features and options, the matrices also qualify capability differences that cut across individual features, such as the composability of various query language features with each other. For readability, we interleave the SQL++ specification sections with the respective benchmarking (capability classification) sections.

The approach of outlining the differences between the various databases using SQL++ achieves two benefits: First, SQL++ offers the reader a formal specification of the discussed features and capabilities. Second, by understanding each database's capabilities in terms of SQL++, 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.

The relatively immature state of query language documentation of the surveyed databases leaves many questions unanswered. We dealt with this problem using a hands-on approach: Each feature matrix has been empirically validated by executing sample queries on the surveyed databases. A benchmark comprising sample queries, empirical observations, as well as links to supporting documentation and bug reports is available at `http://forward.ucsd.edu/sqlpp`.

The feature matrices of this survey paper classify many capabilities of semi-structured data models and query languages. The most prominent capabilities are:

- What kinds of data values are supported by each database?
- What kind of schemas and constraints are supported?
- How does the query language access and construct nested data?
- How is missing information represented and handled?
- What are the options and semantics for equality on non-scalar and heterogeneous values?
- What are the options and semantics for ordering on non-scalar and heterogeneous values?
- Is aggregation supported?

– Is join supported?
– Are extensions (such as UDFs) provided to circumvent limitations?

We expect that some of the results listed in the feature matrices will change in the next years as the surveyed databases will release newer and better implementations. The arXiv/CoRR version of this paper [24] and the benchmark will be updated to reflect these changes.

Despite the forthcoming changes, we expect SQL++ and the comparison methodology followed by this survey to remain a standing contribution. Besides its value to developers, SQL++ can also assist the query language designers in the NoSQL, NewSQL and SQL-on-Hadoop space towards (1) producing formal versions of the syntax and semantics of their query languages (2) aligning with SQL syntax and semantics (whenever possible) and (3) expanding beyond SQL in semantically consistent ways.

Notice that this survey focuses on data model and query language capabilities exclusively: it does not discuss performance or scalability. Furthermore, we do not opine on the business or technical importance of the features that are present (or absent) from each surveyed database.

SQL++ as the Query Language of the FORWARD Virtual Database: Towards aiding the development of software that retrieves data from these new databases, we provide the FORWARD processor (Figure 1). (Additional information at `http://forward.ucsd.edu/sqlpp`) Conceptually, each database (be it SQL or non-SQL) appears to the client as a set of SQL++ virtual views. In the simplest case, the client issues a SQL++ query over the data of a single database, and FORWARD translates the query into the underlying database's native query language. Since SQL is a subset of SQL++, this use case captures the case where an SQL-based report writer issues an SQL query on a non-SQL database. FORWARD will translate the SQL query into the native language. In other more complex cases, given a SQL or SQL++ query that is not directly supported by the underlying database, FORWARD will decompose the SQL or SQL++ query into one or more native queries that are supported. Subsequently, FORWARD will combine the native query results and compensate in the middleware for any semantics or capabilities discrepancies between SQL++ and the underlying database. Finally, in yet more complex cases, the client issues a SQL or SQL++ query that integrates the data of two or more databases; the typical use case being a SQL database and a non-SQL database.

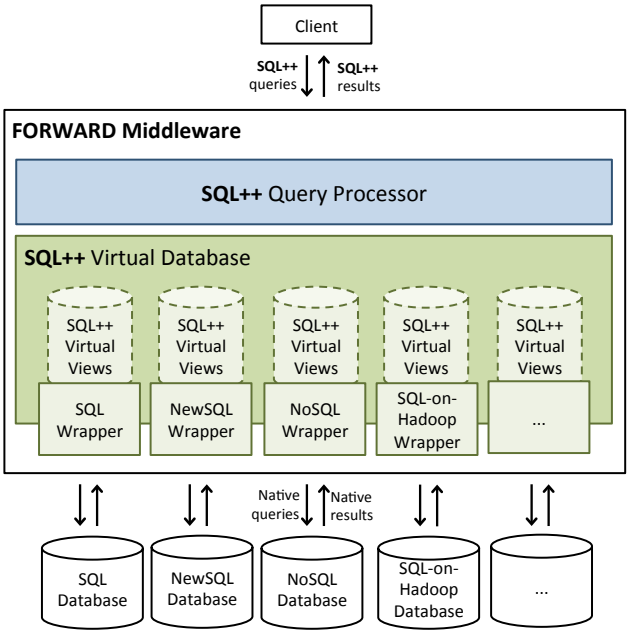

Fig. 1: The SQL++ based FORWARD Virtual Database Query Processor

Due to space limitations, this paper focuses on the language aspects of SQL++ and the comparison of NoSQL, NewSQL and SQL-on-Hadoop databases, rather than the implementation of SQL++ in FORWARD. Nevertheless, the applicability of SQL++ as a unifying data model and query language is practically tested in the FORWARD project.

SQL++ in FORWARD's Live and Interactive Visualizations and Applications: The FORWARD project (`http://forward.ucsd.edu`) also enables the rapid development of live, interactive visualizations and small database-driven applications. The developer simply provides (1) the SQL++ that compiles the dynamic data of the page from the underlying SQL databases, non-SQL databases and application main memory (e.g., session data) and (2) the visualization markup. FORWARD automatically takes care of the AJAX/incremental update of the page when changes in the underlying data occur. For example, Figure 30 (Appendix B) shows a page that accesses a cars collection in a database and displays corresponding markers on the map. Figure 31 shows the page's complete source code. As the data are updated (i.e., as the cars move) the markers correspondingly move on the page, without the need of any additional page update code by the developer.

Since visualization components (e.g. Google Maps) have been assigned enriched JSON models by FORWARD, SQL++ provides an excellent fit for reports

and visualizations, thanks to its richness, document orientation and JSON compatibility. 14 academic and commercial applications have been built using the FORWARD visualization and application development framework.[1]

Roadmap: Section 2 presents preliminaries, including the databases surveyed, and background considerations for the design of SQL++. Section 3 presents the SQL++ data model and compares databases on data modeling features. Section 4 presents the SQL++ query language and compares the databases' query language capabilities. Section 5 presents query language extensions that can be used to circumvent limitations in query capabilities. Section 6 discusses future work beyond features covered in this survey. Finally, Appendix A shows an example of how the FORWARD middleware query processor executes and translates SQL++ queries, and Appendix B shows an example of FORWARD live visualizations and corresponding source code.

2 Preliminaries

2.1 Benchmarked Databases and Query Languages

The databases and query languages surveyed typically appear across different product categories in the popular press:

1. Hadoop (or SQL-on-Hadoop): We survey Apache Hive [31], IBM Jaql [4] and Apache Pig [23]. Our discussion of Hive also applies to Cloudera Impala [6], since Impala is designed to be highly compatible with Hive.
2. NoSQL: Although NoSQL databases initially provided only APIs for accessing data, several have matured to also provide query languages. We survey Apache Cassandra (CQL) [18], MongoDB [21], Couchbase (N1QL) [7] and JSONiq [13].
   We have omitted key-value stores such as Redis [26] and Amazon Dynamo [9] from this survey, since they provide only low-level programmatic APIs instead of query languages. We also exclude graph databases such as Neo4j [22] and RDF stores, as their distinctly different data models are beyond the scope of this survey, i.e. databases at the intersection of relational and JSON-based data modeling.
3. Relational Databases: Since RDBMSes provide comprehensive coverage of the SQL specification, we consider standard SQL as a single database / query language.
4. NewSQL [30]: NewSQL databases, such as Google F1 [29], VoltDB [33] and MemSQL [20], also implement the SQL standard. We do not distinguish between the SQL standard compliance of respective databases, and refer to their query languages in aggregate as SQL.
5. We also survey AsterixDB's AQL [3], Google BigQuery (aka Dremel [19]) and Mongo JDBC [32]. The latter is a JDBC driver provided by the UnityJDBC middleware that allows Java programs to issue SQL statements over a MongoDB database (above). Mongo JDBC translates the SQL statements to MongoDB's API, thereby augmenting MongoDB's query capabilities (e.g. even though MongoDB cannot execute joins, the middleware can simulate some joins) but also altering query semantics (e.g. the middleware may transform the query result so that it conforms to a relational schema as required by JDBC).

	Database	Version
A	Hive	0.10.0
B	Jaql	0.5.1
C	Pig	0.11.0
D	CQL (Cassandra)	3.1.5
E	JSONiq	1.0.11
F	MongoDB	2.4.7
G	N1QL (Couchbase)	dev.preview 2
H	SQL	SQL:2011
I	AQL (AsterixDB)	0.8.3 beta
J	BigQuery	Released Mar 25 2014
K	Mongo JDBC	4.2.316

Table 1: Versions Used for Experimentation

Table 1 tabulates the specific versions of each database that are used to run the experimentation queries for all feature matrices.

2.2 Background of SQL++ Design

SQL++ has been based on SQL, so that its syntax and semantics will be familiar to the vast majority of researchers and developers. The SQL++ data model is an extension of both the relational model and of JSON, which is the dominant semi-structured data model among emerging databases. The SQL++ query language extends SQL with semi-structured capabilities paralleling those of surveyed languages.

We also note how SQL++ and the surveyed databases and languages relate to OQL [2] and XQuery [27]. Indeed, the reader will observe that OQL has greatly inspired and influenced the syntax of SQL++, which can be seen as OQL restricted to literal values (i.e. non-objects), but extended with semi-structured features such as optional schemas and data heterogeneity. Moreover, JSON is much closer to OQL's data

[1] Commercialization site: http://app2you.com

```
 1 'sensors' :: {
 2  location: 'Alpine',
 3  readings: [
 4   {
 5    time : timestamp('2014-03-12T20:00:00'),
 6    ozone: 0.035,
 7    no2  : 0.0050
 8   },
 9   135 :: {
10    time : '135t'::timestamp('2014-03-12T22:00:00'),
11    ozone: 'm',
12    co   : 0.4
13   } ]
14 }
```

Fig. 2: Example of a SQL++ Value

model than XML's. Since both OQL and JSON distinguish between collections versus tuples, we established a straightforward isomorphic mapping between JSON and OQL's data model, whereas an isomorphic mapping between JSON and XML's ordered labeled trees would be significantly more complicated.

3 Data Model

The data models of the surveyed databases reside on the spectrum between the rigidly structured SQL relational model and the semi-structured JSON.[2] On SQL's end, a database has a fixed schema comprising flat tables, where a table is a bag of homogeneous tuples and each tuple is a set of scalar attributes. On JSON's end: (i) there is no fixed schema, (ii) values include object literals (i.e. records) and arrays, which can be nested to arbitrary depth (iii) array elements can be heterogeneous with respect to each other.

The SQL++ data model is designed as a superset of both SQL's relational tables and JSON, based on the observation that the surveyed databases use concepts that are similar across both data models: A JSON array is similar to a SQL table with order, a SQL tuple to a JSON object literal, and a SQL string/integer/boolean to the respective JSON scalar.

Section 3.1 presents the SQL++ data model and then surveys the data model feature support of the surveyed databases. Section 3.2 presents the SQL++ schema features and then surveys accordingly.

3.1 Databases and Values

The syntax of the SQL++ data model follows and extends JSON. Figure 2 shows an example of a database with a single SQL++ value named `sensors`. The value

[2] JSON-alike models are also called *document oriented*.

1	*value*	→	*defined_value*
2		|	**missing**
3	*defined_value*	→	[*id* **::**] *scalar_value*
4		|	[*id* **::**] *complex_value*
5		|	[*id* **::**] **null**
6	*complex_value*	|	*tuple_value*
7		|	*collection_value*
8		|	*map_value*
9	*scalar_value*	→	*primitive_value*
10		|	*enriched_value*
11	*primitive_value*	→	**'** *string* **'**
12		|	*number*
13		|	**true**
14		|	**false**
15	*enriched_value*	→	*type* **(** *primitive_value* **, ...)**
16	*tuple_value*	→	**{** *name* **:** *defined_value* **, ...}**
17	*collection_value*	→	*array_value*
18		|	*bag_value*
19	*array_value*	→	**[** *value* **, ...]**
20	*bag_value*	→	**{{** *value* **, ...}}**
21	*map_value*	→	**map(** *value* **:** *defined_value* **, ...)**

Fig. 3: BNF Grammar for SQL++ Values

provides semi-structured sensor readings that measure enviromental pollutants. It will be used as the paper's running example. The top-level value is a tuple with attributes `location` (line 2) and `readings` (lines 3-13). The latter is an array of two tuples, and the tuples are *heterogeneous* because: (i) each tuple has a different set of attributes (lines 5-7 vs 10-12), and (ii) the `ozone` attribute maps respectively to values of different types across different tuples (line 6 vs 11). Each `time` attribute maps to a timestamp value (lines 5,10), which is an extension over JSON, as described next.

3.1.1 Syntax and Semantics

A SQL++ database generally contains one or more SQL++ named top-level values. A *name*, such as the name `sensors` of Figure 2, is a string and is unique. Figure 3 shows the BNF grammar for SQL++ values. A value is a `missing` value (explained below) or a defined value, i.e. a *scalar*, *complex* or `null` (lines 3-5). A complex value is either a *tuple*, *collection* or *map* (lines 6-8). A scalar value is either *primitive* or *enriched* (lines 9-10). Primitive values are the scalar values of the JSON specification, i.e. strings, numbers or booleans (lines 11-14). Enriched values (such as dates and timestamps) are extensions over JSON, and are specified using a type constructor over primitives (line 15). For example, the value `timestamp('2014-03-12T20:00:00')` (Figure 2, line 5) is an enriched value.

A tuple is a set of attribute name/value pairs, where each attribute name is a unique string within the tuple (as in SQL). Each attribute value can be scalars and `null`, as well as complex values (line 16), thus extending beyond SQL's flat tables. Since SQL++ follows JSON's syntax, tuples are denoted by `{...}`. For ex-

ample, lines 4-8 of Figure 2 show a tuple with attributes `time`, `ozone` and `no2`.

A collection is an *array* or a *bag* (lines 17-18). Both 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 4.3.) 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 `{{...}}`. An array/bag contains tuples (as in SQL), as well as `missing`, `null`, scalars and complex values (lines 19-20), therefore extending beyond SQL's tables. Furthermore, unlike SQL which requires tuples within a table to be homogeneous with respect to each other, elements of a SQL++ array/bag can be heterogeneous. That is, there are no restrictions between the elements of an array/bag. For example, lines 4-8 and 9-13 of Figure 2 show respectively two tuples within an array that are heterogeneous with respect to each other. Another kind of collection is *lists*, i.e., collections whose elements are ordered but are not accessible by position. Due to their functionality overlap with arrays and their use by a single surveyed database, SQL++ does not include lists.

A map contains mappings, where each mapping maps a left value to a right value, and each left value is unique within the map (line 21). A subtle point to note is the similarities and differences between maps and tuples. In effect, both a map and a tuple are sets of name/value pairs where names are unique. However, 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). For the purpose of classification, the SQL++ data model includes both tuples and maps, since most databases support tuples but not maps.

The special value `missing` (line 2) is utilized (in addition to `null`) for representing the result of a path navigation that fails. For example, given tuple `t` with a single attribute `a`, the path `t.b` fails to navigate into a defined value. We formalize the connection between path navigation and `missing` in Section 4.3.

Finally, any SQL++ value may be associated with an optional *id* (see lines 3 to 5 of Figure 3), which is a string or a number and is unique across the whole database. Id's are commonly used in the surveyed databases for efficiently finding a value in the database. (They are often referred to as *keys*.) In particular, given an id k, the associated value is obtained by the call `find(`k`)`. The id k of a value v is syntactically denoted in the BNF by k`::`v. For example, Figure 2 shows the key `135` of the tuple value of lines 9 to 13 and the key `'135t'` of the scalar value of line 10. Notice that id's are optional. Indeed, in the case of elements of arrays, values in maps and attribute values of tuples an application can achieve reasonably similar efficient random access by synthesizing its own id's. For example, consider the tuple `1::{... a: 2:: v}`, which is identified by `1` and has an attribute `a` whose value is identified by `2`. It is easy to see that the attribute's value could also be uniquely identified by the sequence of the id `1` and the attribute name `a`, i.e., by knowing the id of the tuple and the attribute name associated with this attribute value. It is easy to see that the same technique also applies to the elements of arrays and values in maps. It can also apply recursively. For example, if the tuple `1` itself is an attribute value of an enclosing identified tuple `0`, then it can be indirectly identified using the id `0`. [3]

SQL++ extensions over JSON and SQL: In summary, the SQL++ data model extends JSON with `missing` (line 2), enriched values (line 15), bags (line 20) and maps (line 21). SQL++ extends SQL with `missing` (line 2), arrays (line 19) and maps (line 21). Furthermore, SQL++ extends SQL with *arbitrary composition of complex values* and *heterogeneity*. An example of the composition extension is that SQL requires attribute values to be only scalars, whereas SQL++ allows an attribute value to be any arbitrary value. We further elaborate on composability in Section 3.1.2. An example of the heterogeneity extension is that SQL requires all tuples of a bag to have the same attributes with the same types, whereas SQL++ has no such restriction. Indeed, a SQL++ array/bag may contain heterogeneous elements comprising a mix of tuples, scalars, and nested bags/arrays/maps.

3.1.2 Classifying Values

Next we list the feature dimensions according to which we classify and benchmark the data models of the surveyed databases. In Table 2, full support is denoted with ✓, lack of support with ×, and partial support with ❊. We consider both the query input (i.e. the database and also literal constants and subqueries that may appear in a query) and the query output when classifying the SQL++ values each database supports.

1. Composability: In the SQL data model, the top-level value is restricted to be a bag, which in turn

[3] Notice that id's and names could be merged into a single concept, since names are strings that uniquely identify an object. SQL++ keeps the two concepts distinct since names are expected to be semantically meaningful strings (e.g., `sensors`) while keys may be automatically generated strings or numbers.

	Database	1. Composability (Top-Level Value)	2. Heterogeneity	3. Arrays	4. Bags	5. Maps	6. Tuples	7. Primitives
A	Hive	Bag/tuples	×	✓	×	❋	✓	✓
B	Jaql	Any Value	✓	✓	×	×	✓	✓
C	Pig	Bag/tuples	❋	×	❋	❋	✓	✓
D	CQL	Bag/tuples	×	❋	×	❋	×	✓
E	JSONiq	Any Value	✓	✓	×	×	✓	✓
F	MongoDB	Bag/tuples	✓	✓	×	×	✓	✓
G	N1QL	Bag/tuples	✓	✓	×	×	✓	✓
H	SQL	Bag/tuples	×	×	×	×	×	✓
I	AQL	Any Value	✓	✓	✓	×	✓	✓
J	BigQuery	Bag/tuples	×	×	×	×	✓	✓
K	Mongo JDBC	Bag/tuples	✓	✓	×	×	✓	✓

Table 2: Feature Matrix for Values

contains tuples (H1). In that sense, the SQL data model is not composable. The majority of the surveyed databases follow SQL's restriction to composability (A1, C1, D1, F1, G1, J1, K1). The top-level bag of tuples is a special citizen in the sense that an array/map/tuple/scalar must reside as an attribute of a tuple, and not as a top-level value by itself. Furthermore, these databases only support a bag at the top-level, and an attribute's value cannot be a bag. Notably, Jaql, JSONiq and AQL support the top-level to be any arbitrary value (B1, E1, I1), thus providing a composable data model.

2. Heterogeneity: Jaql, JSONiq, MongoDB, N1QL, AQL and Mongo JDBC fully support heterogeneous collections/maps (B2, E2, F2, G2, I2, K2). Conversely, Hive, CQL and BigQuery follow SQL in restricting collections/maps to be homogeneous (A2, D2, H2, J2).

Pig partially supports heterogeneity (C2): the input of a query can contain collections/maps with heterogeneous values. However, if the query's projection requires any computation on these values, such as function calls (e.g. `SELECT x+1`), Pig implicitly coerces the heterogeneous values into values of the same type (e.g. all `x` values will be coerced into numbers), and the query outputs homogeneous collections/maps.

Columns 3-7 classify which values are supported. For databases that treat the top-level as a special citizen (Column 1), these columns ignore the special citizen and indicate which are valid values within attributes of the bag of tuples. For example, SQL supports neither bags nor tuples as attributes of the top-level tuples (H4, H6). The following distinction of collections into arrays and bags in the following classification is based on the availability (or lack thereof) of access by ordinal position. Unlike SQL++ where arrays are ordered collections and bags are unordered, the surveyed databases also utilize bags that are *implied lists*. Given a bag, a database generally supports a method `next` that allows a program to obtain the elements of the bag, one at a time - as is the case in SQL. A bag is an implied list when a sequence of `next` calls returns elements according to a *determinitic order*. The list is implied in the sense that there is no data model feature (or API) that distinguishes between a list and a bag. Rather, it is the programmer's responsibility to know whether she accesses a bag that is an implied list versus a "plain" bag with no deterministic order. Conversely databases that use arrays may output arrays whose order is nondeterministic, i.e., random. Section 4.10.2 will classify cases of implied lists and arrays with nondeterministic order, which pertain to the use (or non-use) of `ORDER BY` in queries.

3. Arrays: Hive, Jaql, JSONiq, MongoDB, N1QL, AQL and Mongo JDBC fully support arrays (A3 B3, E3, F3, G3, I3, K3). CQL has partial support as it restricts an array to only contain scalar values, and not to exceed 65535 elements (D3).

Pig, SQL and BigQuery do not support arrays (C3, H3, J3). In particular, BigQuery uses Protocol Buffers[4] as its data model, which does not support arrays but provides *repeated attributes* as a workaround. Within Protocol Buffers, a tuple does not have unique attribute names, thus multiple attributes with the same name can respectively map to different values. This is reminiscent of XML, where each element can have one or more child elements with the same tag name.

4. Bags: Only AQL fully supports bags (I4). Notably, AQL is the only database that fully supports both arrays and bags. Pig supports bags, but restricts the bag elements to be tuples (C4).

5. Maps: Maps are supported only in Hive, Pig and CQL, but all with restrictions (A5, C5, D5): a Hive map contains only scalar keys, a Pig map contains only string keys, and a CQL map contains only 65535 mappings that have scalar keys and scalar values.

6. Tuples: All databases fully support tuples, except for CQL and SQL both of which do not support a tuple being nested within the top-level bag of tuples (D4, H4). In CQL however, one workaround is to use maps to simulate tuples, but since maps are homogeneous (above), all the attribute values must have the same type.

7. Primitives: Each database fully supports all 3 primitives. Moreover (not shown in table), a few support additional enriched values such as `binary` (Jaql, JSONiq, MongoDB, SQL etc.) and `date` (JSONiq, MongoDB, N1QL, SQL etc.). Since `number` is a double-

[4] A data interchange format widely used in Google for RPC protocols and storage formats [25]

precision floating-point, databases also support values with more efficient representations, such as `integer`, `long` and `float`.

Using the matrix, we also note that three databases support the entire JSON-subset of the SQL++ data model: Jaql, JSONiq and AQL. I.e., they support composability, heterogeneity, arrays, tuples and primitives (B1-3,6-7, E1-3,6-7, I1-3,6-7).

3.2 Schemas

In databases, *schemas* provide a *type system* that can restrict the type and structure of the values and declares acceptable names. For example, a SQL schema dictates table names, attribute names and attribute types. Similarly a XML Schema or a XML DTD declares element names and also restricts the sub-element structure of valid XML documents. Schemas also enforce *semantic constraints*, such as uniqueness and referential integrity/foreign keys. Given the focus of this paper on semi-structured data, it focuses on the type system of SQL++ (and of the surveyed databases) without discussing the details of semantic constraints.

Schemas are used to validate data before they are stored. They are also used to *statically* check that a query is meaningful. For example, a SQL query processor checks that a given query refers to table and attribute names declared in a schema, before the query is actually evaluated on the data. Furthermore, schemas are useful when optimizing storage and queries for performance. Finally, schemas explain the *intention* of a database, even before any data are added to the database.

At one extreme, SQL databases require schemas that impose a rigid structure. At the other extreme, schemaless databases do not support any aspect of schemas. Between the two extremes, the *SQL++ schema language* (and to various degrees some surveyed databases) enables the specification of rigid structures when the structure is known and regular, while it can gracefully degrade towards schemaless (1) when the developer sees no value in declaring and restricting the structure of the data or (2) when the data are not characterized by a regular structure.

Formally, a schema S *allows* a set $allow(S)$ of possible states of the database, while other states are not allowed. In contrast, any state is allowed in a schemaless database, as long as it conforms to the data model. Schema languages can be formally compared by their ability to describe allowed sets of states: A schema language $\mathbf{S}$ is *less expressive* than a language $\mathbf{S'}$ if for every schema S that can be expressed with $\mathbf{S}$ there is an equivalent schema S' that can be expressed with $\mathbf{S'}$, where equivalence means that $allow(S) = allow(S')$. $\mathbf{S}$ is strictly less expressive than $\mathbf{S'}$ if there is at least one schema S' expressible in $\mathbf{S'}$ such that there is no equivalent schema S expressible in $\mathbf{S}$. Practically, the schema language comparisons are explained in Section 3.2.2 in terms of supported features, where each feature is essentially a way to constraint the set of allowed states.

3.2.1 SQL++ Type System

1	*type*	|	*plain_type*
2			[*allow_null*]
3			[*allow_missing*]
4	*allow_null*	→	**null**|**not null**
5	*allow_missing*	→	**missing**|**not missing**
6	*plain_type*	→	*any_type*
7		|	*union_type*
8		|	*scalar_type*
9		|	*complex_type*
10	*complex_type*	→	*tuple_type*
11		|	*collection_type*
12		|	*map_type*
13	*any_type*	→	**any**
14	*union_type*	→	*type* **|** ...
15	*scalar_type*	→	*primitive_type*
16		|	*enriched_type*
17	*primitive_type*	→	**string**
18		|	**number**
19		|	**boolean**
20	*enriched_type*	→	**date**
21		|	...
22	*tuple_type*	|	**{** *open_type* **}**
23		|	**{** *closed_type* **}**
24	*open_type*	→	*attr* **,** ... **, ***
25	*closed_type*	→	*attr* **,** ...
26	*attr*	→	*required_attr*
27		|	*optional_attr*
28	*required_attr*	→	*name* **:** *type*
29	*optional_attr*	|	*name* **?** **:** *type*
30	*collection_type*	→	*array_type*
31		|	*bag_type*
32	*array_type*	→	**[** *type* **]**
33	*bag_type*	→	**{{** *type* **}}**
34	*map_type*	→	**map(** *type* **:** *type* **)**

Fig. 4: BNF Grammar for SQL++ Types

Figure 4 shows the BNF for declaring SQL++ types. Intuitively, a primitive / enriched / tuple / array / bag / map type (lines 17, 20, 22, 32-34) constraints correspondingly the primitive / enriched / tuple / array / bag / map values presented in Section 3.1. A *type* has a *plain type*, and optional *allow-null*/*allow-missing* clauses (lines 1-3). An allow-null clause indicates whether the type allows null values: `null` allows nulls, `not null` indicates otherwise (line 4). The allow-missing clause is analogous (line 5). The `any` type is the $\top$ (i.e. universal supertype) of the type system [35], i.e., it allows all values (line 13). If each type t_i allows set of values V_i, then the union type $t_1|\ldots|t_n$ allows the set of values $V_1 \cup \ldots \cup V_n$ (line 14). A tuple type is either *open* or *closed*: whereas an open type allows a tuple to

```
any

{ location: string, * }

{
  location: string, readings: [
    {
      time: timestamp, ozone: number|string,
      no2?: number, co?: number
    }
  ]
}
```

Fig. 5: Examples of SQL++ Types

contain additional attributes beyond those enumerated, a closed type does not (lines 22-25). Each attribute in a tuple type is either *required* or *optional* (lines 26-29). For example, the tuple type `{ x? : string }` indicates that attribute `x` is optional in the allowed tuples, but if present `x` must be a string value. A collection type is the *array type* or the *bag type* (lines 30-33). In particular, a collection type containing the `any` type or the union type allows heterogeneous elements in the collection value. Lastly, the map type specifies a type for all keys, and a type for all values (line 34). Figure 5 shows 3 examples of SQL++ types that allow the SQL++ value in Figure 2.

	Database	1. Type Check	2. Any Type	3. Union Type	4. Open Type	5. Optional	6. Not Null	7. Not Missing
A	Hive	✓	×	❊	×	×	×	-
B	Jaql	✓	✓	✓	✓	✓	❊	-
C	Pig	✓	×	×	×	×	×	-
D	CQL	✓	×	×	×	×	×	-
E	JSONiq	×	-					
F	MongoDB	×	-					
G	N1QL	×	-					
H	SQL	✓	×	×	×	×	✓	-
I	AQL	✓	×	×	✓	✓	×	-
J	BigQuery	✓	×	×	×	×	✓	-
K	Mongo JDBC	×	-					

Table 3: Feature Matrix for Types

3.2.2 Classifying Type Support

The SQL++ schema language is strictly more expressive than the schema languages of the surveyed databases. It supports a superset of the schema language features of the surveyed databases, which allows us to use SQL++ to classify the schemas of the different databases.

Table 3 classifies the databases by the typing features each supports. Column 1 classifies whether the surveyed databases support static type checking of a query, by retrieving the types of a query's input through either of two mechanisms: (1) In a database that uses schemas to constraint named values (such as SQL, which stores data in typed tables), the static type checker uses the types of the named values. (2) In a database where named values are not constrained by schemas (such as Hive and Pig, both of which physically store data as untyped byte arrays in HDFS), the developer is required to provide functions in the `FROM` clause (or equivalent) to parse these named values and assign types to them, before they can be used as query input. When either mechanism is supported to provide types for the query input, Columns 2-7 further classify whether a database supports the `any` type, union type, open tuple type, optional attribute, `not null` clause and `not missing` clause. Partial support is denoted as ❊, whereas - denote that typing features are not applicable. Based on these columns, the surveyed databases can be grouped into three categories.

1. Schemaless databases do not assign types to query input, as denoted by × in Column 1, and - in Columns 2-7. These databases are JSONiq, MongoDB, N1QL and Mongo JDBC (E1, F1, G1, K1).
2. Fixed schema databases assign only primitive/enriched/tuple/array/bag/map types to query input, and do not have typing features that further relax these types. In particular, these databases constraint a collection's elements to be (deeply) homogeneous with respect to each other. These databases are denoted with ✓ in Column 1 and × in Columns 2-5, namely Hive, Pig, CQL, SQL and BigQuery (A2-5, C2-5, D2-5, H2-5, J2-5). Hive is in this classification because the current version only partially supports the union type (A3), which has been implemented as an enriched type that many of Hive's underlying algebraic operators do not yet handle.
3. Flexible schema databases are those where values can either be constrained by schemas or unconstrained[5]. These databases are denoted by ✓ in Column 1, and one or more ✓ in Columns 2-5, namely Jaql and AQL (B2-5, I2-5).

Both SQL and BigQuery support the `not null` feature (H6, J6). Jaql partially supports the `not null` feature, as non-null clauses and plain types are mutually exclusive (B6). For example, one can specify a non-null type or a string type, but a non-null string type is not supported. If Jaql had fully supported the `not null`

[5] Unconstrained values are specified by a trivial schema: `any`

```
1  query          →  sfw_query
2                 |  expr_query
3  sfw_query      →  config...( sfw_query )
4                 |  SELECT [DISTINCT] select_clause
5                    [FROM from_item ...]
6                    [WHERE expr_query]
7                    [GROUP BY group_item ...]
8                    [HAVING expr_query ...]
9                    [(UNION|INTERSECT|EXCEPT)
10                     [ALL] sfw_query]
11                   [ORDER BY order_item ...]
12                   [LIMIT expr_query]
13                   [OFFSET expr_query]
14 select_clause  →  [TUPLE] select_item ...
15                |  ELEMENT expr_query
16 select_item    →  expr_query [AS attribute]
17 from_item      →  from_single
18                |  from_join
19                |  from_flatten
20 from_single    →  expr_query AS element_var
21                     [AT position_var]
22 from_join      →  from_item
23                     (LEFT|RIGHT|FULL) OUTER|INNER
24                     JOIN from_item ON expr_query
25 from_flatten   →  (OUTER|INNER) FLATTEN(
26                     expr_query AS element_var,
27                     expr_query AS element_var)
28 group_item     →  expr_query [AS grouping_var]
29 order_item     →  expr_query [ASC|DESC]
30 expr_query     →  config...( expr_query )
31                |  ( sfw_query )
32                |  named_value
33                |  variable
34                |  function_name (expr_query...)
35                |  EXISTS( sfw_query )
36                |  path_step
37                |  value
38 variable       →  element_var
39                |  position_var
40                |  grouping_var
41 path_step      →  expr_query . name
42                |  expr_query [ integer ]
43                |  expr_query -> value
44 config         →  @tuple_nav path_params
45                |  @array_nav path_params
46                |  @map_nav path_params
47                |  @group_by group_by_params
48                |  @order_by order_by_params
49                |  @eq equal_params
50                |  @lt less_than_params
51                |  @bag_op bag_op_params
```

Fig. 6: BNF Grammar for SQL++ Queries

feature, its schema language would be strictly more expressive than the other surveyed schema languages.

The `not missing` feature is inapplicable in all surveyed databases: they are either schemaless databases (E7, F7, G7, K7), or their data models do not support the `missing` value (A7, B7, C7, D7, H7, I7, J7).

4 Query Language

Figure 6 shows the BNF grammar for SQL++ queries. SQL++ extends SQL in three ways:

1. **Semi-structured**: Lines 8, 14-29, 32-34, 37, 39-43 correspond to extensions over SQL's syntax to support new semi-structured query capabilities.
2. **Composability**: Any type of value can be constructed by an SQL query or subquery. Furthermore, SQL++ queries are closed under composition, which implies that any query input can equally well be stored data or be a subquery's result. In particular, a SQL++ query (lines 1-2) is either an *SFW query* (i.e. `SELECT-FROM-WHERE`) or *expression query* (for brevity, *expression*). Unlike SQL expressions, which are restricted to outputting tuples of scalars, SQL++ expression queries output arbitrary values (lines 30-43), and are fully composable with SFW queries (lines 16, 20, 24, 26-29). Furthermore, SQL++ supports the top-level query to also be an expression query, not just a SFW query as in SQL.
3. **Configuration parameters** Lines 3, 30, 44-51 indicate SQL++'s novel scheme of *config parameters*, which formally and systematically capture the semantics differences of path navigation, group-by, ordering, bag/set operator, equality and inequality comparison semantics among the surveyed databases. The configuration parameters allow SQL++ to morph into any of the surveyed query languages. Furthermore, database query language designers can treat the values of configuration parameters as their space of options when they formalize the semantics of various operations.

Section 4.1 formalizes query evaluation environments, i.e. how the free variables of a query are instantiated/bound during evaluation. Section 4.2 connects the generation of environments with the evaluation order of query clauses in SFW queries.

Each of the next subsections first defines the syntax and semantics of a SQL++ query feature. Then it utilizes the SQL++ feature definition to systematically classify each database's query capabilities in feature matrices. Section 4.3 formalizes path navigation in tuples, arrays and maps (lines 36, 41-43) and classifies the support for such navigation in each database. Section 4.6 defines the `FROM` clause (lines 5, 17-27) and classifies according to a database query language's ability to iterate over subqueries, joins, flattened collections (i.e. unnesting) and ordinal positions of iterations. The section shows that not all databases have achieved parity with SQL: iterating over subqueries and joins are not fully supported by some of them. Section 4.7 describes the `WHERE` clause (line 6), focusing on ternary logic and subqueries. Section 4.8 describes the `SELECT` clause (lines 4, 14-16) and captures related SQL extensions such as outputting tuples that contain non-scalars (including nested collections) and outputting results that are not collections of tuples. Section 4.9 describes the `GROUP BY` (line 7)

and classifies databases on their capabilities to group on non-scalar/heterogeneous values and to construct nested collections via grouping. Section 4.10 describes the `ORDER BY` (lines 11, 29) and classifies databases on ordering on non-scalar/heterogeneous values, automatic order preservation abilities and the relationship between the output type and the presence (or absence) of `ORDER BY`. Section 4.11 describes bag/set operators (such as `UNION ALL`, lines 9-10) and classifies bag/set operations on non-scalar and heterogeneous values. Section 4.12 describes the `=` equality function and provides config parameters that classify the behavior of equality on non-scalar and heterogeneous values. Similarly, Section 4.13 describes the `<` less-than function and classifies accordingly.

The feature matrices focus on language expressiveness and omit discussions on performance and present system limitations. For example, MongoDB supports two different query APIs: a basic API restricted to filtering values, versus an advanced API that supports aggregation but has the system limitation of restricting output data to 16 MB. When classifying MongoDB's query capabilities, we focus on the latter API, which provides a more expressive language.

4.1 Named Values and Variable Bindings as Query Evaluation Environments

Each SQL++ query q is evaluated within an *environment* Γ that is a *variable binding tuple* (for brevity, *binding tuple*) $\langle x_1 : v_1, \ldots, x_n : v_n \rangle$ where each x_i is a variable and each SQL++ value v_i is the binding of x_i.[6] The notation $\Gamma \vdash q \rightarrow v$ denotes that the SQL++ query q evaluates to the value v within Γ, i.e. when every free variable of q is instantiated by its binding in Γ. For example, $\langle x : 5, y : 3 \rangle \vdash (x + y)/2 \rightarrow 4$.

Before a SQL++ (sub)query is evaluated, the SQL++ semantics produce the variable-binding pairs of the query's environment Γ in two steps. First, for each named value of the database, where the name is n and the value is v, the environment includes the pair $n : v$. Second, similar to SQL's tuple variables, the evaluation of a SQL++ `FROM` clause produces additional variable-binding pairs.[7]

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

For an example of environments, Figure 7 shows the SQL++ query `SELECT s.error * scale AS x FROM sensors AS s`, which multiplies the error margin of each sensor by a scaling factor. The query is evaluated on a database that has two named values: the collection named `sensors` and the number named `scale`. The evaluation of the query and its constituent queries proceeds as follows. First, the query `sensors` is evaluated in the context of Γ_0 (which includes the `sensors` and `scale` variables), and the result is the sensors collection value. The `FROM` clause outputs one binding tuple $b_i = \langle \texttt{s} : v_i \rangle$ for each element v_i where $i = 1, \ldots, n$ of the sensors collection. For each binding tuple b_i, the `SELECT` clause evaluates the constituent query `s.error * scale` within the environment $\Gamma_i = b_i||\Gamma_0 = \langle \texttt{s} : v_i, \texttt{sensors} : \ldots, \texttt{scale: 1.4} \rangle$, outputting a result value r_i. The query result is the bag $\{\{r_1, \ldots, r_n\}\}$. Figure 8 shows the first evaluation of `s.error * scale` within the environment $\Gamma_1 = b_1||\Gamma_0$.

Notice, query evaluation treats each value named n identically to a `FROM` variable named n. For example, the query of Figure 7 could be a nested query where `scale` and `sensors` are variables defined in the `FROM` clause of the parent query. The origin of `scale` and `sensors` does not matter to the query's evaluation.

Notice also that SQL++ supports both names and variables to be arbitrary values. This is in contrast to SQL which restricts named values to flat tables (e.g. `CREATE TABLE`, `WITH`), and tuple variables to flat tuples. SQL++'s blend of names and variables enables full composability, and opens the door to the query processing of arbitrarily structured SQL++ values.

In the following sections, we may denote $\Gamma \vdash q \rightarrow v$ simply as $q \rightarrow v$, when the environment is implied. When v is a collection (resp. scalar/boolean), q is simply referred to as a *collection query* (resp. *scalar/boolean query*).

4.2 SFW queries

Similar to SQL semantics, the clauses of an SFW query (lines 3-13 of Figure 6) are evaluated in the following order: `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT / OFFSET` and `SELECT`. For example, Figure 9 shows an SFW query that finds the top 2 readings below 1.0, and illustrates how each clause inputs/outputs binding tuples. For the purpose of illustration, the `SELECT` clause is shown at the bottom since it is the last clause to be evaluated.

[6] Notice that a binding tuple is an SQL++ tuple. The characterization "binding" pertains to its use in the semantics, rather than structural differences.

[7] Indeed, `GROUP BY` clauses also produce additional variable bindings during query evaluation, as explained in the respective sections.

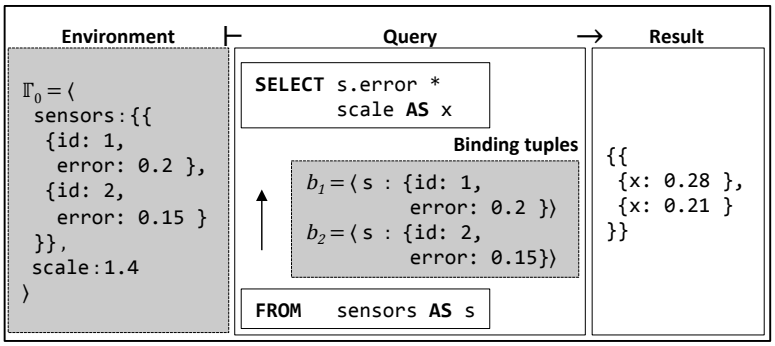

Fig. 7: Environment Γ_0 has variables corresponding to named values

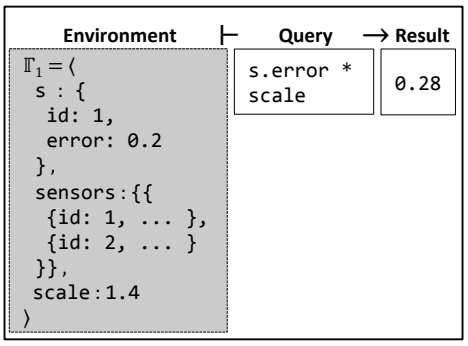

Fig. 8: Environment Γ_1 has both kinds of variables

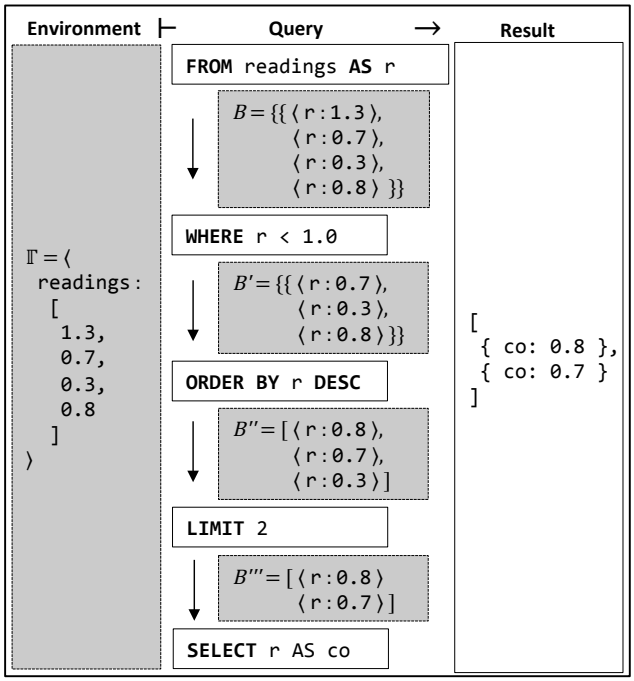

Fig. 9: Evaluating Clauses of an SFW Query

When evaluating a SFW query, the `FROM` clause first outputs a bag of binding tuples B_{out}. In each binding tuple of B_{out}, each variable of the `FROM` clause is bound to a value. For example in Figure 9, each binding tuple binds `r` to a scalar value. Most subsequent clauses input a bag of binding tuples B_{in} and output a bag of binding tuples B_{out}. For example, the `WHERE` clause (which typically follows the `FROM` clause) inputs the bag of binding tuples that have been output by `FROM`, and outputs the subset thereof that satisfies the condition expression of the `WHERE` clause.

In an evaluation pattern that is followed by most clauses, the constituent expression e of the `WHERE` clause (i.e., the condition expression e, which may itself contain nested SFW queries) is evaluated once for each binding tuple b_j of `WHERE`'s input bag B_{in}. The environment of each evaluation of e is $b_j \| \Gamma$, where Γ is the environment of the SFW query. For example in Figure 9, the `WHERE` condition `r < 1.0` is evaluated once for each of the four input binding tuples in B.

The pattern of "input bag, evaluate constituent expressions of the clause, output bag" has a few exceptions, each of them exemplified in Figure 9: 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` 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.

Finally, notice that the above discussion of SFW queries did not capture set operators (`UNION`, `INTERSECT` and `EXCEPT`). As is the case with SQL semantics as well, the coordination of `ORDER BY` with the set operators is treated as a special case, which is described in Section 4.10.

SQL++ variables versus SQL attributes: Consider the query `SELECT x.a, x.b FROM c AS x`, where `c` is a SQL table (i.e. a bag of tuples) with attributes `a` and `b`. SQL semantics dictate that the `FROM` clause outputs (binding) tuples with attributes `a` and `b`. In contrast, the SQL++ `FROM` clause outputs binding tuples, each of which binds a single variable `x` to a tuple with attributes `a` and `b`. SQL++ has introduced an additional level of tuple nesting, which is necessary for two reasons: First, unlike SQL, in SQL++ we cannot assume knowledge of schema, which is required in order to produce the SQL binding tuples. Second, in SQL++ a collection can contain elements beyond tuples, thus SQL++ requires a general approach to bind variables to non-tuple elements.

4.3 Path Steps

Since SQL's data model is flat, there are only paths `t.a`, which navigate from a tuple variable `t` (i.e., a variable bound to a tuple) to a scalar attribute `a`. In contrast, SQL++ provides paths where the navigation initiates from a variable `v` that may be bound not only to a tuple but possibly to a map or an array. Furthermore the navigation may return complex values, such as tuples, arrays and maps.

The semantics of the SQL++ paths also have to account for the semistructured nature of SQL++. Unlike SQL where static type checking of the query guarantees that every path navigation `t.a` is meaningful when the query runs (i.e., static checking guarantees that every tuple bound to the tuple variable `t` indeed has an attribute `a`), the runtime evaluation of an SQL++ path expression has to produce a result even in the case where the path fails to find the target of the navigation. There are more than one legitimate options that a language's designer may take (and, indeed, designers have taken) on what such a path should evaluate to. For example, given tuple `t` with a single attribute `c`, some surveyed databases evaluate the path step `t.b` into `null` and some return `missing` (which is distinct from `null`). To encompass such options and to enable systematic classification, SQL++ uses config parameters to systematically represent all possible options of navigation semantics. In the classification, one of the configurations options is the "error", which captures the inability of some databases to deal with failure to find a path's target.

4.3.1 SQL++ Syntax and Semantics

The basic semantics of SQL++ path describe the cases where the path step navigates from the appropriate source and finds a target:

1. A *tuple path step* `t.a` navigates from tuple `t` into its attribute `a` (Figure 6 line 41).
2. An *array path step* `a[i]` navigates from array `a` into its `i`-th element (line 42).
3. A *map path step* `m->k` navigates from map `m` into the right value of the mapping `g` that has left value `k` (line 43).

Figure 10 shows examples of tuple path steps (line 1), array path steps (line 2) and map path steps (line 3). A path expression with multiple path steps can navigate deeply into complex values (line 4). Note that each path step navigates into 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[8].

```
1 s.readings
2 readings[1]
3 measurements->timestamp('2014-03-12')
4 measurements->timestamp('2014-03-12').readings[1]
```

Fig. 10: Using Path Steps to Navigate into Complex Values

```
1  path_params     →
2    {
3      [absent        : (null|missing|error) ,]
4      [type_mismatch : (null|missing|error)  ]
5    }
```

Fig. 11: BNF Grammar for Path Config Parameters

$$
\begin{array}{lll}
t.a \rightarrow & v & \text{if } t \text{ is a tuple that maps } a \text{ to } v \\
& \texttt{@tuple\_nav.absent} & \text{if } t \text{ is a tuple that does not map } a \\
& \texttt{@tuple\_nav.type\_mismatch} & \text{otherwise} \\
\\
a[i] \rightarrow & v & \text{if } a \text{ is an array with } i\text{-th element as } v \\
& \texttt{@array\_nav.absent} & \text{if } a \text{ is an array with } n \text{ elements } \wedge\ (i < 1 \vee i > n) \\
& \texttt{@array\_nav.type\_mismatch} & \text{otherwise} \\
\\
m.k \rightarrow & v & \text{if } m \text{ is a map that maps } k \text{ to } v \\
& \texttt{@tuple\_nav.absent} & \text{if } m \text{ is a map that does not map } k \\
& \texttt{@tuple\_nav.type\_mismatch} & \text{otherwise}
\end{array}
$$

Fig. 12: Semantics for Path Navigation and Its Config Parameters

Query	→	Result

`@tuple_nav {`
`  absent       : missing,`
`  type_mismatch: null`
`} (`
`  [`
`  { low: 0.033, high: 0.035 }.high,`
`  { avg: 0.034 }.high,`
`  'N/A'.high`
`  ]`
`)`
		`[`
		` 0.035,`
		` missing,`
		` null`
		`]`

Fig. 13: Using Path Config Parameters to Specify Path Navigation Semantics

[8] Since each XPath path step returns a node-set, XPath further defines equality between two node-sets as true if and only if there is a node in the first node-set that equals a node in the second node-set. Consequently, equality becomes a non-transitive relation within XPath. Such is **not** the case in SQL++ or any of the surveyed databases.

The different options for a path step that fails to navigate are specified through SQL++ config parameters, which are presented in Section 4.5, after we have first the general aspects of configuration parameters in Section 4.4.

4.4 SQL++ Configurations

Generally *config parameters* are used to specify the different options of SQL++ query language features. The configured query `@c` t (q) is a *config* c with *config tuple* t on a query q (Figure 6, lines 3, 30). Each config tuple $\{p_1 : o_1, \ldots, p_n : o_n\}$ maps each *parameter* p_i to a *parameter option* o_i, where each option is either an SQL++ value or a *special value*, such as the special value `error`. In either case, a parameter option is statically specified: it cannot be the result of another query. Suppose `@c` t (q) is evaluated in environment Γ. Then q is evaluated in environment $< \texttt{@}c : t > \|\Gamma$. That is, q is evaluated in the context of config `@c`. Generally, a parameter option that is an SQL++ value v is used to dictate that v will be the result of an SQL++ query under conditions specified in the semantics. The special value `error` dictates that query processing fails in these conditions. A concrete use case of config parameters appears next, in the semantics of SQL++ paths.

4.5 SQL++ Path Configurations

The semantics of path navigation are specified by a *path config* (Figure 6, line 44-51). Figure 11 lists the path config parameters and their possible options. Figure 12 shows the semantics of path navigation using the path config parameters. The `absent` parameter specifies the returned value when a path step fails because an attribute/element/mapping is absent: `null`, `missing`, or throw an error (line 3). The `type_mismatch` parameter specifies whether to return `null`, `missing`, or throw an error when a tuple/array/map path step is invoked on a non-tuple/array/map (line 4).

For example in Figure 13, the `@tuple_nav` config specifies that navigating into an absent attribute returns `missing`, and navigating from a non-tuple returns `null`. As another example, SQL's semantics for path navigation is specified with the following config: `@tuple_nav {absent: error, type_mismatch: error}`[9].

	Database	1. Tuple Nav absent	2. Tuple Nav type_mismatch	3. Array Nav absent	4. Array Nav type_mismatch	5. Map Nav absent	6. Map Nav type_mismatch
A	Hive	e	e	n	e	n	e
B	Jaql	n	e	n	e	-	
C	Pig	e	e	-		n	e
D	CQL	e	e	-		-	
E	JSONiq	m	m	m	m	-	
F	MongoDB	m	m	-		-	
G	N1QL	m	m	m	m	-	
H	SQL	e	e	-		-	
I	AQL	n	e	n	e	-	
J	BigQuery	e	e	-		-	
K	Mongo JDBC	m*	m*	-		-	

Table 4: Feature Matrix for Path Navigation

4.5.1 Classifying Tuple, Array and Map Navigation

Table 4 classifies each database's semantics for path navigation in terms of a SQL++ path config. For conciseness, each cell shows only the first letter of a parameter option. Partial support is denoted with *, whereas inapplicability is denoted with -. For example, SQL is denoted with - for array/map config parameters (H3-6), since it lacks support for array/map navigation and its semantics never utilize these config parameters.

1-2. Tuple Navigation: JSONiq, MongoDB and N1QL return `missing` for both absent attributes and type mismatches during tuple navigation (E1-2, F1-2, G1-2). Hive, Pig, CQL, SQL and BigQuery throw errors for both absent attributes and type mismatches (A1-2, C1-2, D1-2, H1-2, J1-2). Jaql and AQL return `null` for absent attributes but throw errors for type mismatches (B1-2, I1-2).

Finally, since Mongo JDBC utilizes MongoDB for query processing, it uses `missing` for intermediate results, but replaces `missing` with `null` in the final output in order to conform to a relational schema as required by JDBC (K1, K2). Furthermore, Mongo JDBC behaves differently for type mismatches based on where tuple navigation is executed. If tuple navigation is delegated to MongoDB, `null` is returned as previously described. However, if tuple navigation is executed in the UnityJDBC middleware, an error will be thrown.

3-4. Array Navigation: In Jaql, JSONiq, N1QL and AQL, array navigation behaves consistently with tuple navigation (B1-4, E1-4, G1-4, I1-4). For Hive, array navigation returns `null` for absent elements, which is more lenient than its tuple navigation which throws an error for absent attributes (A1,3).

[9] Since a SQL database has fixed schemas, these errors are caight by static type checking.

5-6. Map Navigation: In Hive and Pig, map navigation returns `null` for absent mappings, and throws an error for type mismatches (A5-6, C5-6). For Pig, map navigation on absent mappings is more lenient than its tuple navigation, which throws an error for absent attributes (C1,5).

Three subtle quirks warrant further discussion:

1. Recall from Section 3.1 that CQL (partially) supports arrays/maps. Despite so, CQL does not support array/map navigations, and restricts that arrays/maps be retrieved in their entirety (D3-6).
2. When Jaql encounters a type mismatch (B2, B4), a soft error is thrown at runtime: the value is discarded from the current set being processed, but query processing continues for other values. Thus, it is not possible to tell from the query output whether type mismatches have occurred during query evaluation.
3. JSONiq does not always preserve `missing` values. For example, the equivalent JSONiq query of Figure 13 returns a result with the two `missing` values omitted. Since JSONiq has its roots in XQuery processing, it has inherited certain XQuery's semantics. In particular, each JSONiq query inputs and outputs a *sequence* of values, which is denoted with parentheses, i.e. `(...)`. A SQL++ `missing` value is encoded as a JSONiq empty sequence `()`. Moreover, similar to XQuery processing, nested JSONiq sequences are recursively concatenated into a single flat sequence. For example, `((1, 2), (), 3)` is concatenated to become `(1, 2, 3)`. Thus, `missing` values are lost in these cases.

4.6 `FROM` clause

The SQL++ `FROM` clause provides features that allow the variables to range over semi-structured data, unlike the SQL `FROM` clause variables that range over tuples only. The SQL++ variables can range over heterogeneous elements and over nested collections (a feature also known as *unnesting* in nested relational algebras [15,28]). Furthermore, `FROM` clause variables can register the order of input elements, which is useful for order-aware queries.

The surveyed databases vary in their level of support of such SQL++ semi-structured abilities. They also differ in their support of classical SQL capabilities, such as iterating over the results of subqueries and joining multiple collections.

4.6.1 SQL++ Formalism

The `FROM` clause specifies `FROM` items (Figure 6, line 5), where each `FROM` item is an expression (e.g. a named value, a SFW subquery, etc), `JOIN`/`OUTERJOIN` clause or a `FLATTEN` clause (lines 17-19), and outputs a bag of binding tuples B_{out}.[10] For ease of exposition, we first present the semantics for `FROM` clauses with a single `FROM` item, thereafter extend the semantics to clauses with multiple `FROM` items.

1. A `FROM` item with a single expression/sub-query is: `FROM` e `AS` x [`AT` y] (Figure 6, lines 20-21). x is an *element variable*, and y is a *position variable*. As in SQL, e can be a subquery or a named value (lines 31-32). SQL++ extends beyond SQL towards full composability, as e can also be a variable, function call, path or value literal (lines 33-37). Since an SQL++ collection comprises elements that may be tuples (as in SQL) but also arrays, bags and scalars, SQL's tuple variable in the `FROM` clause is generalized to SQL++'s element variable. Furthermore, the position variable is also an extension over SQL.
   Let $e \rightarrow c$, where c is a collection (i.e., bag or array) with n elements $v_1, \ldots, v_n$.[11] For each v_j, $j = 1 \ldots, n$, the `FROM` clause outputs a binding tuple b_j,where $b_j = \langle x : v_j,\ y : u_j \rangle$ and u_j is the ordinal position of v_j in c when c is an array, or the sentinel value -1 when c is a bag, where the -1 signifies that c was not an array and hence positions are meaningless. As in SQL, the `FROM` clause iterates over collections in non-deterministic order. An output order may be later imposed by `ORDER BY`.
   For example, Figure 14 shows a SQL++ query that uses a position variable to obtain ordinal positions of an array, and outputs the array in reverse order. The `FROM` clause defines the element variable `r` and the position variable `p`, and outputs binding tuples b_1, b_2 and b_3 that bind `r` and `p` to scalar values. The `ORDER BY` clause uses the variable `p` to sort the binding tuples in descending order of their original ordinal positions in the array.
2. Next consider a `FROM` clause involving a single join or a single outerjoin. (The following semantics generalize in a straightforward way to a join/outerjoin expression with multiple joins and/or outerjoins.) In the single join/outerjoin the `FROM` clause syntax is: `FROM` l `AS` x_l [`AT` y_l] J `JOIN` r `AS` x_r [`AT` y_r] `ON` e (Figure 6, lines 23-25), where l and r are expressions/subqueries, x_l and x_r are element variables,

[10] We will see that the `FLATTEN` and `JOIN` clauses are not strictly necessary, since they can be emulated by sequences of expressions.
[11] If c is not a collection, then $e \rightarrow \{\{c\}\}$.

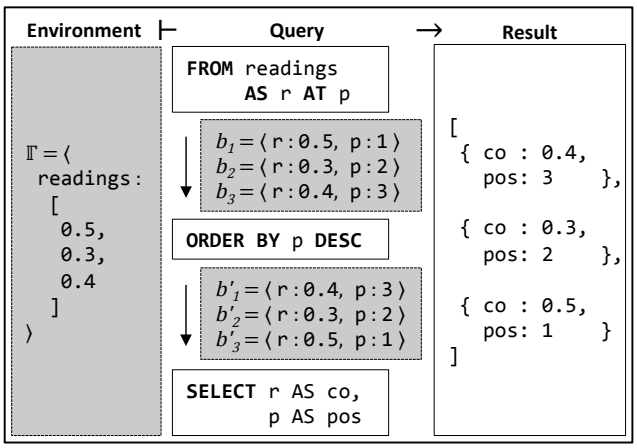

Fig. 14: Using AT (Position Variable) to Obtain Ordinal Positions for Input Order

y_l and y_r are position variables, J is the join type (LEFT/RIGHT/FULL OUTER join or INNER join) and e is an expression that may involve any of x_l, x_r, y_l, y_r and returns a boolean.

Suppose the enclosing query is evaluated in environment Γ. Let us call B_l the bag of binding tuples that would result from the single FROM item clause FROM l AS x_l AT y_l and respectively call B_r the bag of binding tuples that would result from FROM item clause FROM r AS x_r AT y_r. Then the inner join FROM clause outputs all binding tuples $b_l \| b_r$, $b_l \in B_l$, $b_r \in B_r$ for which $b_l \| b_r \| \Gamma \vdash e \rightarrow \texttt{true}$, i.e., the result of joining B_l and B_r on the condition e. A left outer join FROM clause outputs all the binding tuples that the inner join outputs, plus for each b_l that did not match a b_r, it outputs a binding tuple $b_{\text{out}} = b_l \| \langle x_r : \texttt{null}, y_r : \texttt{null} \rangle$[12]. The analogous applies for right/full outer joins.

3. A FROM item with a FLATTEN clause is a SQL++ extension (Figure 6, lines 25-27). Consider first the FROM clause
FROM(OUTER|INNER) FLATTEN(e AS x, $x.p$ AS y),
where OUTER/INNER is the *flatten type*, e is the *parent expression*, x is the *parent element variable*, $x.p$ is the *child path* and y is the *child element variable*. Notice that the child path $x.p$ is rooted at x.

Suppose the expression e evaluates into a collection c. The INNER FLATTEN clause outputs all binding tuples $\langle x : u,\ y : v \rangle$, such that $u \in c$ and $v \in u.p$, i.e., v is a descendant of u by following the path p.

[12] Notice that, instead of the SQL-compliant null, the semantics could have introduced missing or choose to not have attributes x_r and y_r in a subset of the binding tuples, which would be acceptable since SQL++ does not require homogeneity.

Analogous to outer joins, the OUTER FLATTEN clause outputs the same binding tuples as INNER FLATTEN, plus for each $u \in c$ where $u.p$ is the empty collection, outputs a binding tuple $\langle x : u,\ y : \texttt{null} \rangle$. Again, instead of the null, the semantics could have introduced missing or produce a binding tuple that does not have the variable y.

For example, Figure 15 shows a SQL++ query that uses INNER FLATTEN to unnest nested bag readings. In the FROM clause, for each parent element s, the child path s.readings evaluates to a bag of two scalar elements. Thus, the FROM clause outputs binding tuples $b_1 \ldots b_4$, each of which bind variables s and r. The WHERE clause outputs b'_1 and b'_2, which satisfy the expression r > 0.2.

Finally, consider a FROM clause with multiple items $f_1 \ldots f_n$. Whereas SQL only supports FROM items that are independent of each other, SQL++ further supports correlated FROM items. The expression(s) of f_i can utilize any variables output by the preceding items $f_1 \ldots f_{i-1}$. Consider FROM items $f_1, \ldots, f_i$, $i \leq n$. The result of the FROM is defined inductively: Assume that the clause FROM $f_1, \ldots, f_{i-1}$, where $i < n$ outputs binding tuples B_{i-1}. Then the FROM $f_1, \ldots, f_i$ outputs all binding tuples $b_{i-1} \| b_i$, such that $b_{i-1} \in B_{i-1}$, $b_i \in B'$, where B' is the set of binding tuples resulting from the evaluation of FROM f_i within $b_{i-1} \| \Gamma$.

Relationship between types of FROM items Notice that the INNER FLATTEN feature does not increase the expressiveness of SQL++, as it can be easily simulated. For example, the FROM clause of the query of Figure 15 can be written simply as FROM sensors AS s, s.readings AS r. Similarly, the JOIN (but not the right/left/full OUTER JOIN) can also be simulated by moving the join condition to the WHERE clause.

4.6.2 Classifying FROM abilities

Table 5 classfies each database's semantics for iterations. Partial support is denoted with ❋, whereas inapplicability is denoted with -.

1. Iterating over Subqueries: CQL, MongoDB, N1QL and Mongo JDBC support a subset of SQL++ where a FROM item is restricted to be a named collection (D1, F1, G1, K1). In particular, a FROM item is never a subquery, causing queries to be non-composable in the FROM clause.

2. Joins: CQL, MongoDB and N1QL do not support joins (D2, F2, G2). They support a subset of SQL++ where there is no JOIN clause, and the FROM clause can only contain a single item (i.e. there is no Cartesian product).

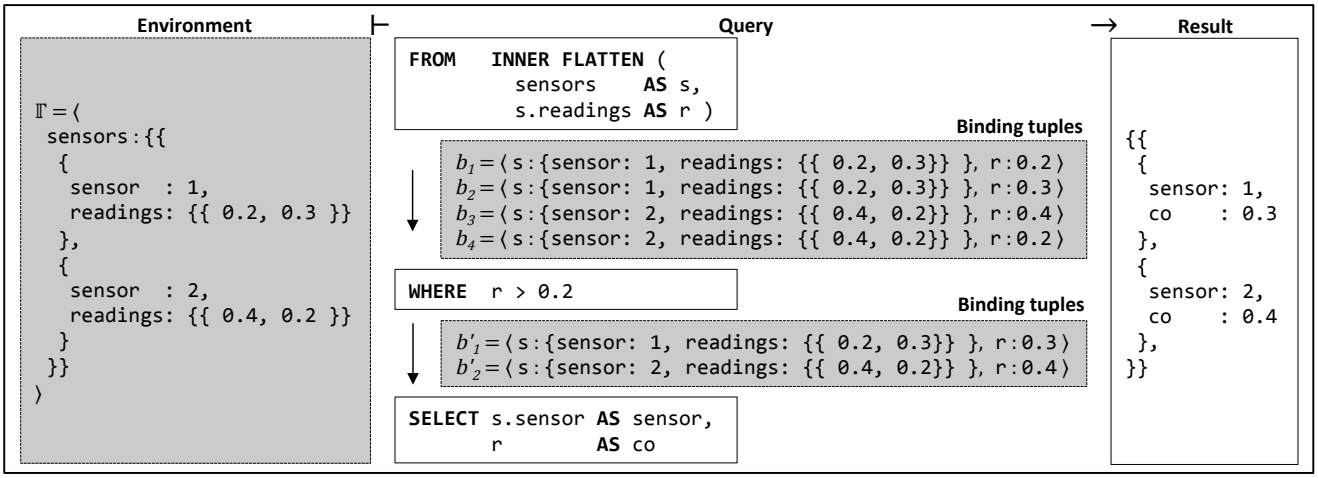

Fig. 15: Using `INNER FLATTEN` to Unnest Collections

	Database	1. Iterating over Subqueries	2. Joins	3. Unnesting (INNER FLATTEN)	4. Unnesting (OUTER FLATTEN)	5. Ordinal Positions
A	Hive	✓	❊	✓	✓	✓
B	Jaql	✓	❊	×	✓	✓
C	Pig	✓	❊	✓	×	✓
D	CQL	❊	×	×	×	-
E	JSONiq	✓	✓	✓	×	✓
F	MongoDB	❊	×	✓	×	-
G	N1QL	❊	×	✓	×	-
H	SQL	✓	✓	×	×	✓
I	AQL	✓	✓	✓	×	✓
J	BigQuery	✓	❊	×	✓	✓
K	Mongo JDBC	❊	✓	×	×	-

Table 5: Feature Matrix for Iterations, Joins, Unnesting and Ordinal Positions (`FROM` clause)

A few databases syntactically restrict join conditions: Hive, Jaql, Pig and BigQuery support only equi-joins, i.e. a join with the condition restricted to a conjunction of equalities (A2, B2, C2, J2). To simulate an inner join with an arbitrary condition, one workaround is to use `true` as the join condition and place the actual arbitrary condition in the `WHERE` clause, but this workaround does not apply for left/right/full outer joins. Furthermore, BigQuery does not support right/full outer joins.

Lastly and notably, Mongo JDBC augments MongoDB by simulating joins in its middleware (K2).

3-4. Unnesting: Hive, Pig, JSONiq, MongoDB, N1QL and AQL support unnesting that is equivalent to `INNER FLATTEN` (A3, C3, E3, F3, G3, I3), whereas Hive, Jaql and BigQuery support unnesting that is equivalent to `OUTER FLATTEN` (A4, B4, J4). Notably, Hive supports both semantics for unnesting.

Also, note that unnesting collections is supported by MongoDB but not Mongo JDBC (F3, K3), thus the limitation is of the UnityJDBC middleware.

5. Ordinal Positions: Both JSONiq and AQL support obtaining ordinal positions for input order in the same way as SQL++: by specifying a position variable in a `FROM` item (E5, I5). Hive, Jaql, Pig, SQL and BigQuery also support obtaining ordinal positions, albeit through the projection of a new attribute that captures the ordinal position (A5, B5, C5, H5, J5). The new attribute is constructed by `enumerate()` in Jaql, `RANK` in Pig, and the `ROW_NUMBER` window function in Hive, SQL and BigQuery. Note that Hive, Pig, SQL and BigQuery support obtaining ordinal positions from bags, since these databases support deterministic order on a bag as long as the bag is output by a subquery with an `ORDER BY`. (Section 4.10.2 further classifies deterministic ordering in output collections.)

Finally, CQL, MongoDB, N1QL and Mongo JDBC are denoted with - (D5, F5, G5, K5). In each of these databases: (i) `FROM` items are restricted to named collections (D1, F1, G1, K1) (ii) named collections are always bags. Effectively, these limitations cause a query to always iterate over an unordered collection, thus obtaining ordinal positions is inapplicable.

4.7 `WHERE` clause

4.7.1 SQL++ Formalism

As in SQL, the `WHERE` clause is a boolean expression e (Figure 6, line 6), which may include an `EXISTS` subquery (line 35). The `WHERE` clause inputs a bag of binding tuples B_{in}, and outputs a bag of binding tuples B_{out}. For each $b \in B_{\text{in}}$, the `WHERE` clause outputs b if

	Database	1. Selection
A	Hive	❊
B	Jaql	✓
C	Pig	✓
D	CQL	❊
E	JSONiq	✓
F	MongoDB	❊
G	N1QL	❊
H	SQL	✓
I	AQL	✓
J	BigQuery	❊
K	Mongo JDBC	❊

Table 6: Feature Matrix for Selection

$b \| \Gamma \vdash e \rightarrow \texttt{true}$, where Γ is the environment of the enclosing query. I.e., if b satisfies the condition e.

4.7.2 Classifying Selection

Table 6 classifies each database's semantics for selection, and partial support is denoted with ❊.

Hive, CQL, MongoDB, N1QL, BigQuery and Mongo JDBC provide only partial support for selection (A1, D1, F1, G1, J1, K1). Hive supports selecting on subqueries where correlation occurs only in the `WHERE` clause conditions of the subqueries, or selecting on subqueries that are uncorrelated. CQL, MongoDB, N1QL, BigQuery and Mongo JDBC do not support any subqueries in the `WHERE` clause.

CQL further restricts the `WHERE` clause as follows (D1). Each named collection stored in the database has a mandatory primary key comprising one or more attributes of the collection, and the stored tuples are clustered based on the primary key. The `WHERE` clause only supports a conjunction of conditions such that (i) each condition is a comparison between an attribute and a literal (ii) the attribute is part of the primary key, or a secondary index is defined on it (iii) the tuples selected by the `WHERE` clause are contiguous in the clustering order.

4.8 `SELECT` clause

The SQL++ data model supports collections containing more than flat tuples (see Section 3.1). For example, a collection's tuple can contain other nested collections. Moreover, a collection can contain directly (i.e. without intervening tuples) any arbitrary value, including scalars, collections, maps, `null` and `missing`. Respectively the `SELECT` clause of SQL++ can create such structures. The surveyed databases support varying capabilities for creating values other than collections of tuples.

4.8.1 SQL++ Formalism

1. For creating tuples, the `SELECT TUPLE` clause specifies a list of `SELECT` items (Figure 6, lines 4, 14, 16): `SELECT` [`TUPLE`] e_1 `AS` a_1`,` $\ldots$`,` e_n `AS` a_n
   For backwards-compatibility with SQL, the `TUPLE` keyword is optional in SQL++. Each `SELECT` item comprises an expression e_i and an output attribute a_i. The `SELECT TUPLE` clause inputs a bag (resp. array, if `ORDER BY` is present) of binding tuples B_{in} and outputs a bag (resp. array) of tuples. For each input binding tuple $b \in B_{\text{in}}$, the `SELECT TUPLE` clause outputs a tuple $t = \{a_1 : v_1, \ldots, a_n : v_n\}$, where $b \| \Gamma \vdash e_i \rightarrow v_i$, $1 \leq i \leq n$, and Γ is the environment of the enclosing query.
   Figures 16 and 17 show an example of using `SELECT TUPLE` to create tuples containing nested collections. In Figure 16, the `SELECT TUPLE` clause inputs binding tuples b_1 (resp. b_2), and outputs a tuple with attributes `sensor` and `readings`. Notice that the inner query (which outputs the nested collection `readings`) is parameterized by the outer query's `s` element variable in the `WHERE` clause. Figure 17 shows the first evaluation of the inner query within the environment $\Gamma_1 = b_1 || \Gamma_0$.
2. For creating arbitrary values (i.e. tuples and non-tuples), the `SELECT ELEMENT` clause specifies an expression e (Figure 6, line 15). Similar to `SELECT TUPLE`, the `SELECT ELEMENT` clause inputs a bag (resp. array, if `ORDER BY` is present) of binding tuples B_{in} and outputs a bag (resp. array) of values. For each input binding tuple $b_j \in B_{\text{in}}$, the `SELECT ELEMENT` clause outputs a value v_j, where $b_j \| \Gamma \vdash e \rightarrow v_j$, and Γ is the environment of the enclosing query.
   Figure 18 shows an example of using `SELECT ELEMENT` to project non-tuples. The SFW query inputs a bag of heterogeneous elements, and outputs the identical bag.

SQL++ provides syntactic sugar for specifying *collection comprehensions* [34]. `{{` e_s `|` m`:`e_f `}}` is a shortcut that is equivalent to the following query: `SELECT ELEMENT` e_s `FROM` e_f `AS` m. Figure 19 shows an example of a collection comprehension and its equivalent query.

4.8.2 Classifying Projection

Table 7 classifies each database's semantics for projection, where partial support is denoted with ❊.

1. Projecting Tuples Containing Nested Collections: Jaql, JSONiq and AQL fully support projecting tuples containing nested collections (B1, E1 and I1).

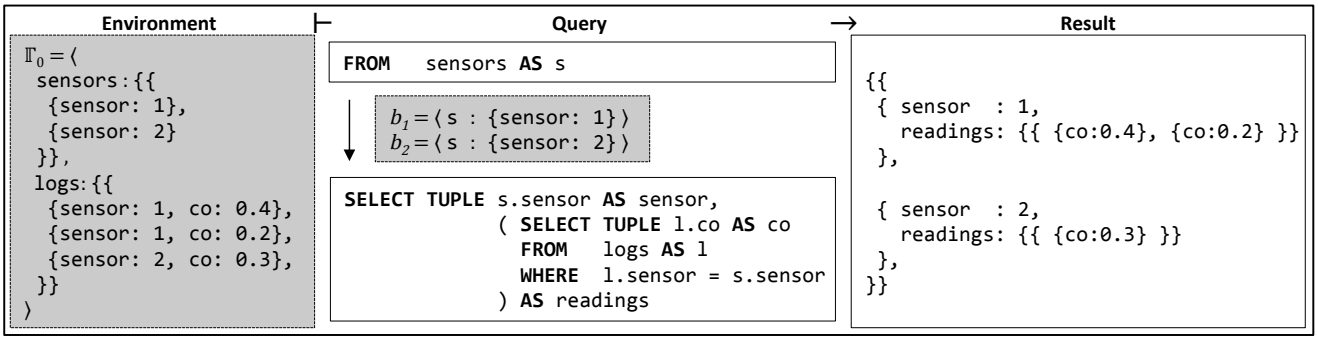

Fig. 16: Using SELECT TUPLE to Project Tuples Containing Nested Collections (outer query)

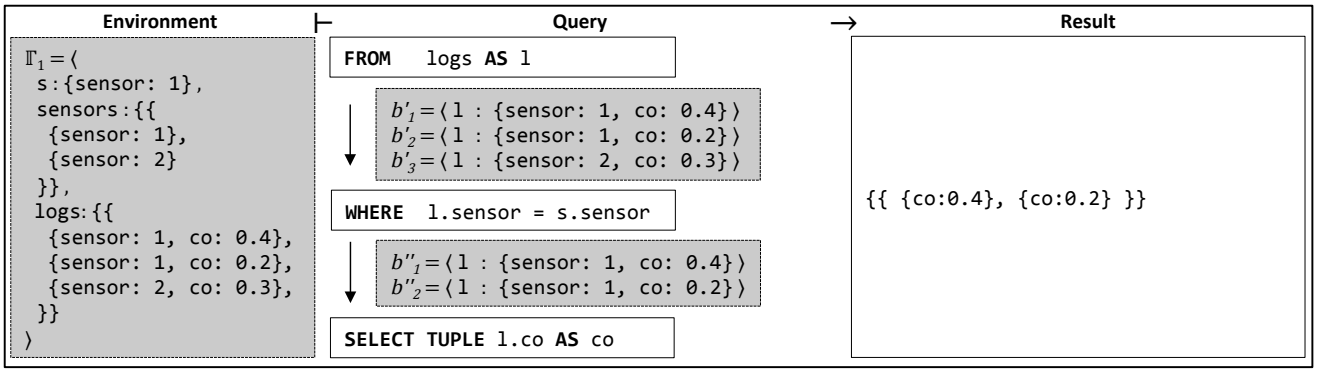

Fig. 17: Using SELECT TUPLE to Project Tuples Containing Nested Collections (inner query)

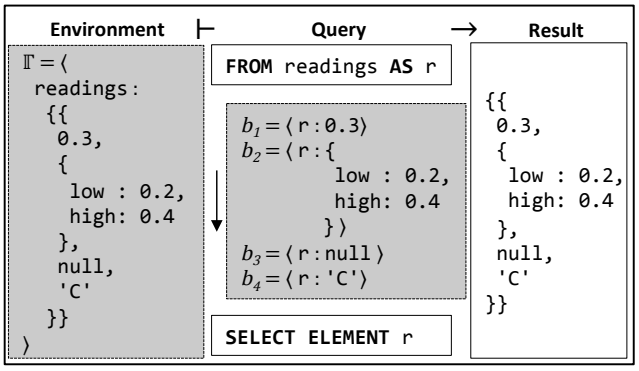

Fig. 18: Using SELECT ELEMENT to Project Non-Tuples

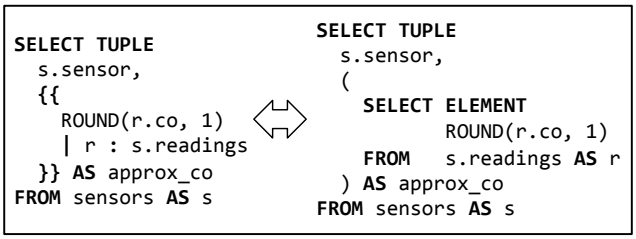

Fig. 19: Syntactic Sugar vs Equivalent Query for Collection Comprehension

	Database	1. Projecting Tuples Containing Nested Collections	2. Projecting Non-Tuples
A	Hive	❊	×
B	Jaql	✓	✓
C	Pig	❊	×
D	CQL	×	×
E	JSONiq	✓	✓
F	MongoDB	❊	❊
G	N1QL	❊	❊
H	SQL	×	×
I	AQL	✓	✓
J	BigQuery	×	×
K	Mongo JDBC	×	×

Table 7: Feature Matrix for Projection

Hive, Pig, MongoDB and N1QL are denoted with ❊ (A1, C1, F1, G1), as they support a subset of SQL++ where SELECT TUPLE is restricted to comprise only paths and collection comprehensions, but not subqueries. This is a limitation since a subquery supports various clauses such as JOIN, GROUP BY, UNION etc. whereas a collection comprehension does not. Moreover, Hive and MongoDB further restrict a collection comprehension such that its output (i.e. e_s) comprises only paths, but not arbitrary expressions such as functions (e.g. ROUND in Figure 19).

Recall from Section 3.1 that BigQuery supports repeated attributes in lieu of supporting nested collections. Due to this data model limitation, BigQuery is

denoted with × for projecting tuples containing nested collections (J1). Nonetheless, BigQuery supports path expressions over repeated attributes, which effectively behave like the collection comprehensions supported by Hive and MongoDB.

Finally, notice that list comprehensions are supported in MongoDB, but not Mongo JDBC (F1, K1).

2. Projecting Non-Tuples: Jaql, JSONiq and AQL have full support for projecting non-tuples (B2, E2 and I2). In contrast, Hive, Pig, CQL, SQL, BigQuery and Mongo JDBC only project tuples (A2, C2, D2, H2, J2 and K2). Finally, MongoDB and N1QL are denoted with ⁕, as they support a subset of SQL++ where the expression of `SELECT ELEMENT` is restricted to be only a path or collection comprehension, but not a subquery.

4.9 `GROUP BY` clause

In SQL, the `GROUP BY` clause partitions its input tuples into groups. SQL++ and certain surveyed databases are more general than SQL in the use of the resulting groups: They allow the explicit use of the groups in the `SELECT`, `HAVING` and `ORDER BY` clauses, which potentially perform complex computations on the groups, leading to results of any type (e.g. nested collections). In contrast, SQL always provides the groups to one or more aggregation functions that output scalar results.

In SQL, two tuples are in the same group if their group-by attributes are equal or, more generally, if their *grouping expressions* (line 7 of Figure 6) are equal. However, `GROUP BY` uses a grouping equality function, called `IS NOT DISTINCT FROM`, which differs from the `=` equality function (discussed in Section 4.12): Whereas `null = null` evaluates to `null`, `null IS NOT DISTINCT FROM null` evaluates to `true`. Consequently, two tuples with `null` as attribute `a` will be placed in the same group with `GROUP BY a`. For brevity, we use ≡ to denote `IS NOT DISTINCT FROM`, and refer to it as the *identity test* since it returns `true` when its two arguments are identical and `false` otherwise. SQL++ and the surveyed databases analogously use the identity test (rather than the `=` equality function) as the grouping equality function. Unlike SQL, the SQL++ identity test must also compare complex values, values of different types and `missing`. The survey observes that the identity checks of many surveyed databases are incomplete, in the sense that they throw errors when comparing certain values.

4.9.1 SQL++ Formalism

The syntax of the `GROUP BY` clause is
`GROUP BY` e_1 `AS` x_1`, ...,` e_m `AS` x_m
where each e_i is a *grouping expression* and each x_i is a *grouping variable*. In SQL, each e_i evaluates to a scalar or `null` expression. SQL++ extends e_i to also evaluate to a complex or `missing`. In addition, a grouping expression e_i can evaluate to values of different types (i.e. heterogeneous values) across the equivalence groups, unlike SQL which restricts each e_i to always evaluate to homogeneous values across equivalence groups.

Similar to SQL semantics, the SQL++ `GROUP BY` clause utilizes the identity test. The bag of input binding tuples B_{in} are partitioned using ≡ into the minimal number of equivalence groups $B_1 \dots B_o$ such that any two binding tuples $b, b' \in B_{\text{in}}$ are in the same equivalence group if and only if every expression e_i evaluates to the identical value v_i given b and given b', i.e., $b||\Gamma \vdash e_i \mapsto v_i$ and $b'||\Gamma \vdash e_i \mapsto v_i$, where Γ is the environment of the query. For each B_k $(1 \leq k \leq o)$, the `GROUP BY` clause outputs a binding tuple $b_k =$
$\langle x_1 : v_1,\ \dots,\ x_m : v_m,\ \texttt{group} : B_k \rangle$.

The grouping variables $x_1 \dots x_m$ and `group` can be utilized in subsequent `HAVING`, `ORDER BY` and `SELECT` clauses. For SQL-compatibility, SQL++ also supports the syntactic sugar of allowing the use of an e_i in lieu of x_i in these three clauses and omitting x_i in the `GROUP BY` clause. Therefore, the two SFW queries below are equivalent:

(1)		(2)	
`SELECT`	$f(x_i), \dots$	`SELECT`	$f(e_i), \dots$
`FROM`	$\dots$	`FROM`	$\dots$
`GROUP BY`	e_i `AS` $x_i, \dots$	`GROUP BY`	$e_i, \dots$
`HAVING`	$f'(x_i, \dots)$	`HAVING`	$f'(e_i, \dots)$
`ORDER BY`	$f''(x_i), \dots$	`ORDER BY`	$f''(e_i), \dots$

The `group` variable, which binds to each group, can be used in expressions (including SFW subqueries) of the `HAVING`, `ORDER BY` and `SELECT` clauses. In combination with functions that input collections and output scalars (i.e., aggregate functions such as `sum()`, `max()` etc.) SQL++ supports the aggregation functionality of SQL. In addition, SQL++ also supports complex transformations on `group` before providing it to an aggregate function. For example, Query 1 of Figure 20 groups readings by respective sensors, and outputs for each sensor (1) its id, (2) a nested bag of its readings, (3) the number of readings, (4) the average of its carbon monoxide (i.e. `co`) readings. The query's `GROUP BY` clause inputs three binding tuples (b_1, b_2, b_3), partitions them into two equivalence groups (sensors `1` and `2`), and outputs two binding tuples (b'_1, b'_2). Notice that the nested bag `readings` is produced by a conventional SFW subquery, whose `FROM` clause binds

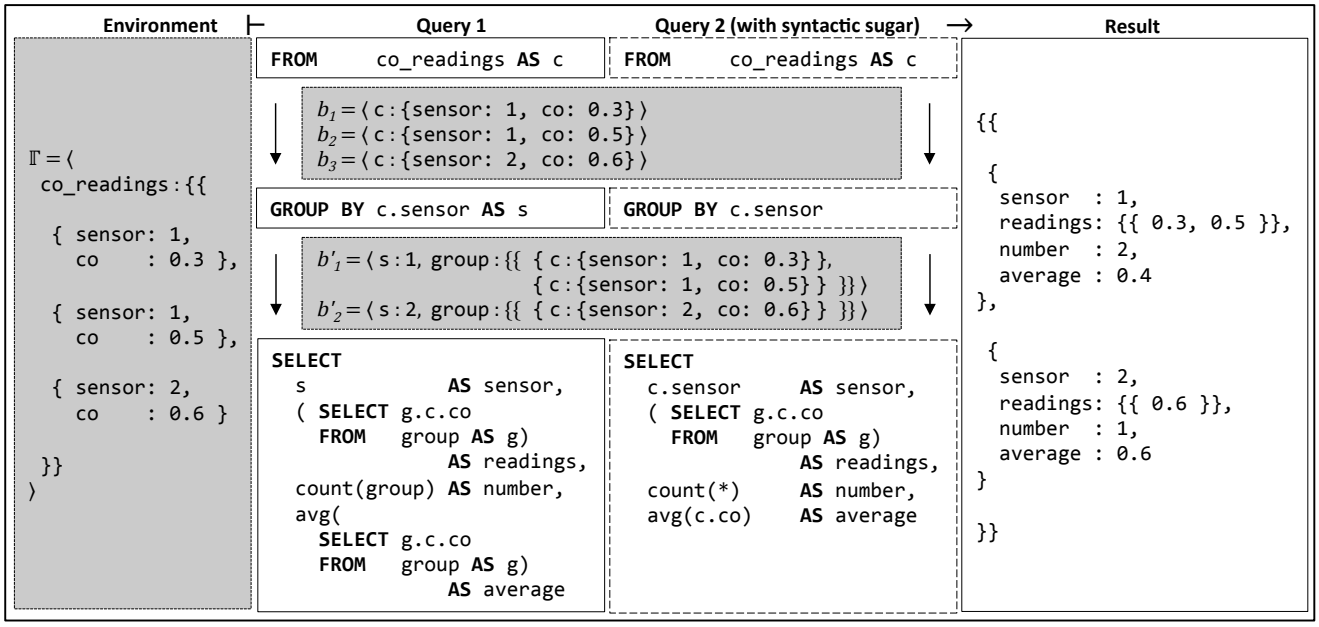

Fig. 20: Using `GROUP BY` and Aggregation Functions

`g` to each tuple of `group`, and outputs the `c.co` of each tuple. The number of readings is produced by the `count()` function, which inputs a collection of elements and returns an integer. In this example, the input collection is `group`. Similarly, the average is produced by providing to `avg()` the result of a nested query over `group`[13].

Unlike SQL, where the semantics of the aggregation functions are coupled with the `GROUP BY` clause, the SQL++ `GROUP BY` is only responsible for creating equivalence groups. Nevertheless, in the interest of syntactic compatibility with SQL, SQL++ introduces the following syntactic sugar in the `SELECT`, `HAVING` and `ORDER BY` clauses: Consider any of these clauses having an expression $f(e)$, where f is an SQL aggregation function (such as `sum()`, `avg()`) and the expression e directly refers to variables $x_1, \ldots, x_n$ of the tuples of `group`. Then $f(e)$ is syntactic sugar for $f($`SELECT` e' `FROM group AS` $g)$, where e' results from substituting each x_i in e with $g.x_i$[14]. For the special case where an aggregation function f (such as `count()`) needs the entire `group` as-is, we define that $f(*)$ stands for $f($`group`$)$. For example, Query 2 of Figure 20 utilizes these syntactic sugar, and has the aggregations

[13] One may wonder why the tuples of `group` have the form `{ c: {sensor: ...} }` as opposed to the simpler form `{sensor: ...}`. This is because the `FROM` clause that precedes `GROUP BY` may in the general case join multiple collections and output more variables than just `c`.

[14] SQL-compatibility also requires syntactic sugar for queries such as `SELECT avg(x.a) FROM c AS x`, where it is implied that all elements of `c` become a single group that is aggregated.

```
1  group_by_params  →
2    {
3      [complex            : (boolean|error) ,]
4      [type_mismatch      : (false  |error) ,]
5      [null_eq_null       : (true   |error) ,]
6      [null_eq_value      : (false  |error) ,]
7      [missing_eq_missing : (true   |error) ,]
8      [missing_eq_value   : (false  |error) ,]
9      [null_eq_missing    : (false  |error) ]
10   }
```

Fig. 21: BNF Grammar for Group-By Config Parameters

`count(*)` and `avg(c.co)`. Notice that `c` is not a top-level variable in the input binding tuples of the `SELECT` clause. Consequently, `avg(c.co)` is equivalent to: `avg(SELECT g.c.co FROM group AS g)` (Query 1).

A *group-by config* (Figure 6, line 47) specifies the identity test capabilities of various databases. Figure 21 shows the BNF grammar of the 7 group-by config parameters, and Figure 22 shows the semantics of the $\equiv$ identity test function with respect to these config parameters. The `complex` parameter specifies whether to return a boolean `true`/`false` or throw an error when comparing two complex values (line 3). The `type_mismatch` parameter specifies whether to return `false` or throw an error when comparing two values of different types (line 4). The remaining 5 parameters specify the semantics when comparing with `null`/`missing` values: `null_eq_null` when comparing two `null` values (line 5), `null_eq_value` when comparing `null` to a value that is not `null`/`missing` (line 6), `missing_eq_missing` when comparing two

$$x \equiv y \rightarrow \begin{cases} f_{\texttt{scalar}}(x,y) & \text{if } x \text{ and } y \text{ are scalar} \\ f_{\texttt{complex}}(x,y) & \text{if } x \text{ and } y \text{ are complex} \\ \texttt{@group\_by.type\_mismatch} & \text{if } x \text{ (resp. } y\text{) is scalar } \wedge \ y \text{ (resp. } x\text{) is complex} \\ \texttt{@group\_by.null\_eq\_null} & \text{if } x \text{ and } y \text{ are } \texttt{null} \\ \texttt{@group\_by.missing\_eq\_missing} & \text{if } x \text{ and } y \text{ are } \texttt{missing} \\ \texttt{@group\_by.null\_eq\_missing} & \text{if } x \text{ (resp. } y\text{) is } \texttt{null} \wedge \ y \text{ (resp. } x\text{) is } \texttt{missing} \\ \texttt{@group\_by.null\_eq\_value} & \text{if } x \text{ (resp. } y\text{) is } \texttt{null} \wedge \ y \text{ (resp. } x\text{) is not } \texttt{null/missing} \\ \texttt{@group\_by.missing\_eq\_value} & \text{if } x \text{ (resp. } y\text{) is } \texttt{missing} \wedge \ y \text{ (resp. } x\text{) is not } \texttt{null/missing} \end{cases}$$

$$f_{\texttt{scalar}}(x,y) \rightarrow \begin{cases} f_{\texttt{sql}}(x,y) & \text{if } x \text{ and } y \text{ are strings (resp. numbers / booleans)} \\ \texttt{@group\_by.type\_mismatch} & \text{if } x \text{ and } y \text{ are different types} \end{cases}$$

$$f_{\texttt{complex}}(x,y) \rightarrow \begin{cases} \texttt{error} & \text{if } \texttt{@group\_by.complex} \text{ is } \texttt{error} \\ f_{\texttt{array}}(x,y) & \text{if } \texttt{@group\_by.complex} \text{ is } \texttt{boolean} \wedge \ x \text{ and } y \text{ are arrays} \\ (\text{resp. } f_{\texttt{bag}}, f_{\texttt{tuple}}, f_{\texttt{map}}) & (\text{resp. bags, tuples, maps}) \\ \texttt{@group\_by.type\_mismatch} & \text{if } \texttt{@group\_by.complex} \text{ is } \texttt{boolean} \wedge \ x \text{ and } y \text{ are different types} \end{cases}$$

$$f_{\texttt{array}}(x,y) \rightarrow \begin{cases} x\texttt{[1]} \equiv y\texttt{[1] AND } \ldots \texttt{ AND } x\texttt{[}n\texttt{]} \equiv y\texttt{[}n\texttt{]} & \text{if } x \text{ and } y \text{ each has } n \text{ elements} \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$f_{\texttt{bag}}(x,y) \rightarrow \begin{cases} v_1 \equiv v_1 \texttt{ AND } \ldots \texttt{ AND } v_n \equiv v_n & \text{if } x \text{ and } y \text{ each consists of elements } v_1 \ldots v_n \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$f_{\texttt{tuple}}(x,y) \rightarrow \begin{cases} x.a_1 \equiv y.a_1 \texttt{ AND } \ldots \texttt{ AND } x.a_n \equiv y.a_n & \text{if } x \text{ and } y \text{ each consists of attributes } a_1 \ldots a_n \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$f_{\texttt{map}}(x,y) \rightarrow \begin{cases} x\texttt{->}k_1 \equiv y\texttt{->}k_1 \texttt{ AND } \ldots \texttt{ AND } x\texttt{->}k_n \equiv y\texttt{->}k_n & \text{if } x \text{ and } y \text{ each consists of keys } k_1 \ldots k_n \\ \texttt{false} & \text{otherwise} \end{cases}$$

Fig. 22: Semantics for the `GROUP BY` Identity Test (≡) with respect to `GROUP BY` Config Parameters

`missing` values (line 7), `missing_eq_value` when comparing `missing` to a value that is not `null`/`missing` (line 8), and `null_eq_missing` when comparing `null` with `missing` (line 9). These parameters return `true`, `false` or throw an error (lines 5-9). For example, SQL's semantics are specified with: `@group_by {null_eq_null: true, null_eq_value: false}`. (Other parameters are not applicable since the SQL data model does not support complex values, missing values and heterogeneity.) Note that we will utilize the same config parameters in Section 4.11 to specify the equality used internally by the bag/set operators, and Section 4.12 to specify the `=` equality function.

4.9.2 Classifying Grouping to Construct Tuples of Nested Collections

Table 8 classifies each database's support for grouping. Hive, Jaql, Pig, JSONiq, MongoDB, N1QL and AQL fully support using grouping to construct tuples containing nested collections, in the same way as using the `GROUP BY` clause and the `group` variable (A1, B1, C1, E1, F1, G1, I1).

Both SQL and Mongo JDBC support the `GROUP BY` clause but not the `group` variable (H1, K1), whereas CQL does not support `GROUP BY` altogether (D1).

Recall from Section 3.1 that BigQuery supports repeated attributes in lieu of supporting nested collections. Due to this data model limitation, BigQuery is denoted with × (J1). Nonetheless, BigQuery supports a similar form of `GROUP BY` / `group` which constructs tuples containing repeated attributes.

	Database	1. Grouping to Construct Tuples of Nested Collections
A	Hive	✓
B	Jaql	✓
C	Pig	✓
D	CQL	×
E	JSONiq	✓
F	MongoDB	✓
G	N1QL	✓
H	SQL	×
I	AQL	✓
J	BigQuery	×
K	Mongo JDBC	×

Table 8: Feature Matrix for Grouping

4.9.3 Classifying Grouping on Non-Scalar and Heterogeneous Values

Table 9 classifies each database's semantics for grouping on non-scalar and heterogeneous values in terms of a SQL++ group-by config. For conciseness, each cell shows only the first letter of a parameter value. Partial support is denoted with ✻, whereas inapplicability is denoted with -. For example, SQL is denoted with - in Column 1, since it lacks support for complex values, and

	Database	1. complex	2. type_mismatch	3. null_eq_null	4. null_eq_value	5. missing_eq_missing	6. missing_eq_value	7. null_eq_missing
A	Hive	b	-	t	f	-		
B	Jaql	b	f	t	f	-		
C	Pig	b*	-	t	f	-		
D	CQL	-						
E	JSONiq	e	f	t	f	t	f	f
F	MongoDB	b	f	t	f	t	f	f
G	N1QL	b	f	t	f	t	f	f
H	SQL	-	-	t	f	-		
I	AQL	e	e	t	f	-		
J	BigQuery	e	-	t	f	-		
K	Mongo JDBC	e	e	t	f	t	f	*

Table 9: Feature Matrix for Grouping on Non-Scalar and Heterogeneous Values

its semantics never utilize `@group_by.complex`. CQL is denoted with - as it does not support the `GROUP BY` clause (row D), and we do not discuss it further.

1. Complex Values: Hive, Jaql, MongoDB and N1QL are the only databases that fully support grouping on complex values (A1, B1, F1, G1). Pig only supports deep equality for maps and tuples, but not bags (C1).

JSONiq, AQL and BigQuery throws an error when grouping on complex values (E1, I1, J1).

Surprisingly, Mongo JDBC also throws an error when grouping on complex values (K1), even though MongoDB supports it (F1). The underlying reason is because Mongo JDBC performs grouping in its middleware, which has different semantics from grouping in MongoDB.

2. Type Mismatch: Jaql, JSONiq, MongoDB and N1QL group two values of different types into separate equivalence groups (B2, E2, F2, G2), whereas AQL and MongoJDBC throw an error (I2, K2).

Recall from Section 3.2.2 that Hive, Pig, SQL and BigQuery are fixed schema databases. Since these databases only support homogeneous collections, grouping values of different types is not applicable, and the databases are denoted with - (A2, C2, H2, J2).

For the same reasons described above, Mongo JDBC deviates from MongoDB's grouping semantics when grouping values of different types (K2).

3-4. Null Values: All databases group `null` values identically as SQL (H3-4): two `null` values are in the same equivalence group, whereas a `null` value and a value that is not `null`/`missing` are in different groups.

5-6. Missing Values: All 4 databases that support the `missing` value, namely JSONiq, MongoDB, N1QL and Mongo JDBC handle `missing` and `null` values in a symmetrical fashion during grouping (E3-6, F3-6, G3-6, K3-6).

7. Null and Missing Values: JSONiq, MongoDB and N1QL group `null` and `missing` into different equivalence groups (E7, F7, G7). Only Mongo JDBC has the confusing behavior of grouping `null` and `missing` into the same equivalence group (K7).

4.10 `ORDER BY` clause

In SQL's `ORDER BY` clause, the `NULLS FIRST` and `NULLS LAST` keywords indicate whether a `null` value is ordered before or after all other values. Unlike SQL, the SQL++ `ORDER BY` also supports an order expression evaluating to heterogeneous values, complex values and `missing`. The surveyed databases analogously support the SQL++ `ORDER BY` at various levels.

4.10.1 SQL++ Formalism

As in SQL, the `ORDER BY` clause syntax is:
`ORDER BY` $e_1 \dots e_m$ [`ASC`|`DESC`] (Figure 6, lines 11), where $e_1 \dots e_m$ is a list of *ordering expressions*. In SQL++ a SFW query with `ORDER BY` outputs an array, whereas a SFW query without `ORDER BY` outputs a bag. This is in contrast to SQL (and many surveyed databases that do not support arrays), where the output of `ORDER BY` is an implied list (see Section 3.1).

The `ORDER BY` clause sorts its input using the *total order function* $<^o$, i.e., $<^o$ returns `true` or `false` when comparing any two values. Similar to SQL, the presence of bag/set operators (such as `UNION` and `UNION ALL`) in a SFW query results in different semantics for `ORDER BY` as follows:

1. **SFW without bag/set operators:** The `ORDER BY` clause inputs a bag of binding tuples B_{in}, thus each ordering expression e_i can utilize variables from the preceding `FROM` or `GROUP BY` clauses. The `ORDER BY` clause outputs an array of sorted binding tuples B_{out}. Consider sorting in ascending order when T specifies `ASC`. Let $b_1, \dots, b_n$ be the binding tuples in B_{in}, and for each $b_j \in B_{\text{in}}$, let $v_{j,1} \dots v_{j,m}$ be the evaluation of the ordering expressions $e_1 \dots e_m$. Given two input binding tuples b_j and b_k (i) b_j appears before b_k in B_{out} if $v_{j,1} <^o v_{k,1}$ is `true` (ii) b_k appears before b_j if $v_{k,1} <^o v_{j,1}$ is `true` (iii) ties are broken by comparing $v_{j,2}$ and $v_{k,2}$ otherwise, and so on. If ties remain up till $v_{j,m}$ and $v_{k,m}$, $<^o$ nondeterministically outputs `true` or `false`, and the order between b_j and b_k is nondeterministic. Sorting in descending order is defined analogously.

Recall from Section 4.2 that `ORDER BY` is evaluated before `SELECT`. For SQL-compatibility, given `SELECT` e_i `AS` a_i, SQL++ also supports the syntactic sugar of using a_i in lieu of e_i in the `ORDER BY` clause. Therefore, both SFW queries below are equivalent:

(1) `SELECT` e_i `AS` a_i `FROM` ... `ORDER BY` a_i

(2) `SELECT` e_i `AS` a_i `FROM` ... `ORDER BY` e_i

2. **With bag/set operators:** Consider SFW queries of the form: q_1 `UNION` q_2 `ORDER BY` $e_1 \ldots e_m$ T, where q_1 and q_2 are also SFW queries. As discussed in Section 4.11, the `UNION` clause is evaluated after the `SELECT` and outputs a bag of values, which is in turn input by `ORDER BY`. Thus, in the presence of bag/set operators, instead of binding tuples, the `ORDER BY` clause inputs a bag of values C = `{{`u_1`, ..., `u_n`}}` and outputs an array of sorted values.
   The ordering expressions $e_1 \ldots e_m$ use the special `ELEMENT` variable to refer to the elements (values) of the input bag. Suppose the enclosing query q of the `ORDER BY` clause is evaluated within environment Γ. For each $u \in C$, let $v_1 \ldots v_m$ be the evaluation of the ordering expressions $e_1 \ldots e_m$ within environment $\Gamma' = \langle$`ELEMENT` $: u\rangle||\Gamma$. Given two input element values u_j and u_k, the `ORDER BY` clause orders them by comparing the results of the ordering expressions when evaluated in the environment $\langle$`ELEMENT` $: u_j\rangle||\Gamma$ with the results of the ordering expressions in $\langle$`ELEMENT` $: u_k\rangle||\Gamma$.
   .
   For SQL-compatibility, SQL++ allows the `ELEMENT` variable to be omitted from ordering expressions when the `ELEMENT` variable binds to a tuple u (as is the case for SQL tables). Ordering expressions $e_1 \ldots e_m$ are evaluated within environment $\Gamma' = u||\langle$`ELEMENT` $: u\rangle||\Gamma$. That is, u is treated as a binding tuple, which binds the variables utilized in $e_1 \ldots e_m$. Hence when u is a tuple one may or may not use the `ELEMENT`.

Automatic order preservation SQL++ also provides the `ORDER PRESERVE` clause in lieu of the `ORDER BY`. The `ORDER PRESERVE` is applicable only in the absence of `GROUP BY`. It orders the output elements according to the order of the respective input elements, as is formally explained by the equivalence of the two queries below:

```
1  order_by_params  →
2    {
3      |complex            : (boolean   |error) ,|
4      |type_mismatch      : (boolean   |error) ,|
5      |null_lt_null       : (     false|error) ,|
6      |null_lt_value      : (true|false|error) ,|
7      |missing_lt_missing : (     false|error) ,|
8      |missing_lt_value   : (true|false|error) ,|
9      |null_lt_missing    : (true|false|error) |
10     |type_order         : [ type_name , ...] |
11   }
```

Fig. 23: BNF Grammar for Order-By Config Parameters

(1) `SELECT` $*$ `FROM` e_1 `AS` v_1 `AT` p_1, ... e_n `AS` v_n `AT` p_n `ORDER BY` $p_1, \ldots, p_n$

(2) `SELECT` $*$ `FROM` e_1 `AS` v_1, ... e_n `AS` v_n `ORDER PRESERVE`

SQL++ allows an ordering expression e_i to evaluate to a scalar or `null` value (as in SQL), and extends it to also evaluate to a complex or `missing` value. Furthermore, e_i can evaluate to values of different types (i.e. heterogeneous values), unlike SQL which restricts each e_i to always evaluate to homogeneous values. The semantics of $<^o$ with respect to complex, `missing` and heterogeneous values is specified by an *order-by config* (Figure 6, line 48).

Figure 23 shows the BNF grammar of the order-by config parameters, and Figure 24 shows the semantics of the $<^o$ function with respect to these order-by config parameters. An order-by config has 8 parameters, and the first 7 (Figure 23, lines 3-9) are analogous to those of a group-by config (Section 4.9, Figure 21). A key difference is that `type_mismatch` accepts `boolean` (instead of `false`). Unlike the $\equiv$ identity test function which is symmetric, the $<^o$ function is a non-symmetric relation, thus `boolean` specifies that the function returns `true`/`false` based on the relative ordering between the types of the arguments. This ordering is specified in the 8th parameter `type_order` (line 10), which accepts a list comprising shallow type names (e.g. `array`, `tuple`). For example, a `type_order` parameter with the following pattern: `[..., number, ..., string, ...]`, and the `type_mismatch` parameter set to `boolean` will result in `1 < 'a'` $\rightarrow$ `true`. Finally, notice that `null_lt_value`, `missing_lt_value` and `null_lt_missing` accepts both `true` and `false`. For example, SQL's semantics for `ORDER BY ... NULLS FIRST` is specified with the following order-by config: `@order_by {null_lt_null: false, null_lt_value: true}`. (Other parameters are not applicable since the SQL data model does not support complex values, missing values and heterogene-

$$x <^o y \rightarrow \begin{cases} f_{\texttt{scalar}}(x,y) & \text{if } x \text{ and } y \text{ are scalar} \\ f_{\texttt{complex}}(x,y) & \text{if } x \text{ and } y \text{ are complex} \\ f_{\texttt{type}}(x,y) & \text{if } x \text{ (resp. } y\text{) is scalar } \wedge \ y \text{ (resp. } x\text{) is complex} \\ \texttt{@order\_by.null\_lt\_null} & \text{if } x \text{ and } y \text{ are } \texttt{null} \\ \texttt{@order\_by.missing\_lt\_missing} & \text{if } x \text{ and } y \text{ are } \texttt{missing} \\ \texttt{@order\_by.null\_lt\_missing} \text{ (resp. } \texttt{@order\_by.missing\_lt\_null}\text{)} & \text{if } x \text{ (resp. } y\text{) is } \texttt{null} \wedge \ y \text{ (resp. } x\text{) is } \texttt{missing} \\ \texttt{@order\_by.null\_lt\_value} \text{ (resp. } \texttt{@order\_by.value\_lt\_null}\text{)} & \text{if } x \text{ (resp. } y\text{) is } \texttt{null} \\ & \wedge\ y \text{ (resp. } x\text{) is not } \texttt{null/missing} \\ \texttt{@order\_by.missing\_lt\_value} \text{ (resp. } \texttt{@order\_by.value\_lt\_missing}\text{)} & \text{if } x \text{ (resp. } y\text{) is } \texttt{missing} \\ & \wedge\ y \text{ (resp. } x\text{) is not } \texttt{null/missing} \end{cases}$$

$$f_{\texttt{type}}(x,y) \rightarrow \begin{cases} \texttt{error} & \text{if } \texttt{@order\_by.type\_mismatch} \text{ is } \texttt{error} \\ \texttt{true} & \text{if } \texttt{@order\_by.type\_mismatch} \text{ is } \texttt{boolean} \ \wedge \quad \text{the type of } x \text{ precedes the type of } y \text{ in } \texttt{@order\_by.type\_order} \\ \texttt{false} & \text{if } \texttt{@order\_by.type\_mismatch} \text{ is } \texttt{boolean} \ \wedge \ \neg(\text{the type of } x \text{ precedes the type of } y \text{ in } \texttt{@order\_by.type\_order}) \end{cases}$$

$$f_{\texttt{scalar}}(x,y) \rightarrow \begin{cases} f_{\texttt{sql}}(x,y) & \text{if } x \text{ and } y \text{ are strings (resp. numbers / booleans)} \\ f_{\texttt{type}}(x,y) & \text{if } x \text{ and } y \text{ are different types} \end{cases}$$

$$f_{\texttt{complex}}(x,y) \rightarrow \begin{cases} \texttt{error} & \text{if } \texttt{@order\_by.complex} \text{ is } \texttt{error} \\ f_{\texttt{array}}(x,y) & \text{if } \texttt{@order\_by.complex} \text{ is } \texttt{boolean} \ \wedge \ x \text{ and } y \text{ are arrays} \\ (\text{resp. } f_{\texttt{bag}}, f_{\texttt{tuple}}, f_{\texttt{map}}) & \text{(resp. bags, tuples, maps)} \\ f_{\texttt{type}}(x,y) & \text{if } \texttt{@order\_by.complex} \text{ is } \texttt{boolean} \ \wedge \ x \text{ and } y \text{ are different types} \end{cases}$$

$$f_{\texttt{array}}(x,y) \rightarrow \begin{cases} x\texttt{[}j\texttt{]} <^o y\texttt{[}j\texttt{]} & \text{if } x \text{ and } y \text{ are equal up to the first } j-1 \text{ elements and not equal on the } j\text{-th element} \\ & \text{i.e. } (\forall 1 \le i \le j-1,\ x\texttt{[}i\texttt{]=}y\texttt{[}i\texttt{]} \rightarrow \texttt{true}) \ \wedge \ \neg(x\texttt{[}j\texttt{]=}y\texttt{[}j\texttt{]} \rightarrow \texttt{true}) \\ \texttt{true} & \text{if } x \text{ has fewer elements than } y \quad \wedge \quad x \text{ and } y \text{ are equal up to all elements of } x \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$f_{\texttt{bag}}(x,y) \rightarrow \{ f_{\texttt{array}}(x',y') \quad \text{where } x' \text{ (resp. } y'\text{) is an array constructed from } x \text{ (resp. } y\text{) sorted in lexical order}$$

$$f_{\texttt{tuple}}(x,y) \rightarrow \begin{cases} f_{\texttt{array}}(x',y') & \text{where } x' \text{ (resp. } y'\text{) is an array } [a_1, v_1, \ldots, a_m, v_m] \\ & \text{such that } a_1 \ldots a_n \text{ are the attribute names of } x \text{ (resp. } y\text{) sorted in lexical order,} \\ & \text{each } a_i \text{ maps to attribute value } v_i \text{ in } x \text{ (resp. } y\text{)} \end{cases}$$

$f_{\texttt{map}}(x,y)$ is defined analogously as $f_{\texttt{tuple}}(x,y)$

Legend:

1. SQL++ values are lexically ordered as follows: booleans $\prec$ numbers $\prec$ strings $\prec$ arrays $\prec$ tuples $\prec$ maps $\prec$ bags $\prec$ `null` $\prec$ `missing`.

Fig. 24: Semantics for the `ORDER BY` $<^o$ Function with respect to `ORDER BY` Config Parameters

ity.) Note that we will utilize the same config parameters in Section 4.13 to specify the `<` less-than function.

4.10.2 Classifying Deterministic Ordering in Output Collections

Unlike SQL++, where a query's output is an array in the presence of `ORDER BY` and a bag in the absence thereof, in all surveyed databases a query's output has implicit order. In the presence of `ORDER BY` the surveyed databases output bags that are implied lists. Recall from Section 3.1 that an implied list is a bag where a sequence of `next` calls returns elements according to a deterministic order. The list is "implied" in the sense that there is no data model feature (or API) that distinguishes between a list and a bag. Rather, it is the programmer's responsibility to know when she accesses a bag that is an implied list versus a "plain" bag with no deterministic order. Unlike an array, an implied list does not support random access by ordinal position,

	Database	1. Output with `ORDER BY`	2. Output without `ORDER BY`
A	Hive	Bag (Implied List)	Bag
B	Jaql	Array	Array (random order)
C	Pig	Bag (Implied List)	Bag
D	CQL	Bag (Implied List)	Bag
E	JSONiq	Seq	Seq
F	MongoDB	Bag (Implied List)	Bag
G	N1QL	Bag (Implied List)	Bag
H	SQL	Bag (Implied List)	Bag
I	AQL	Array	Array (random order)
J	BigQuery	Bag (Implied List)	Bag
K	Mongo JDBC	Bag (Implied List)	Bag

Table 10: Output of `ORDER BY` queries

neither allows the use of the elements' ordinal positions in subsequent queries. Therefore the order produced by a subquery cannot be utilized by its enclosing query. For example, if the SQL++ `ORDER BY` output implied

lists (instead of arrays) the query:

```
SELECT   x.a
FROM     (SELECT t.a FROM t ORDER BY t.b)
         AS x AT p
ORDER BY p
```

would have nondeterministally-ordered output. Similarly, the order of a view cannot be utilized by queries that use the view.

SQL outputs bags that are implied lists. Hive, Pig, CQL, MongoDB, N1QL, BigQuery and Mongo JDBC also output bags that are implied lists in the presence of `ORDER BY`, as indicated in Table 10.

Conversely, Jaql and AQL always output an array (B1-2, I1-2), even though the array's order is nondeterministic (marked as "random" in Table 10) in the absence of `ORDER BY`. Similarly, JSONiq queries always output JSONIQ sequences (as explained in Section 4.5.1), even though the sequence's order is nondeterministic when (i) the `unordered` flag is present and (ii) the `ORDER BY` clause is absent.

JSONiq is the only database that supports automatic order preservation.[15] All other databases output results with deterministic order when `ORDER BY` is present, and nondeterministic order when `ORDER BY` is absent.

Finally, we note a few classification subtleties. Whereas Table 10 classifies values that are output by a SFW query, Table 2 (Section 3.1.2) classifies values that are input/output by any query (i.e. SFW query or expression query). This results in Table 10 comprising only arrays/bags, whereas Table 2 shows that Jaql, JSONiq and AQL support the top-level value to be any arbitrary value. This also explains why Table 10 shows that an AQL query outputs only arrays, whereas Table 2 shows that AQL supports both arrays and bags. In AQL, bags are restricted to occur only in literals and named values, not as the top-level value output by a SFW query.

4.10.3 Classifying Ordering of Non-Scalar and Heterogeneous Values

Table 11 classifies each database's semantics for ordering on non-scalar and heterogeneous values in terms of the SQL++ order-by config. For symmetry between `GROUP BY` and `ORDER BY` with respect to non-scalar and heterogeneous values, we shade a cell gray in Table 11 if it is identical to its corresponding cell in Table 9 (Section 4.9.3), and focus the following discussion only on non-shaded cells. For conciseness, each cell shows only the first letter of a parameter value. Partial support is denoted with ✻, whereas inapplicability is denoted with -. CQL restricts an `ORDER BY` expression to be a clustering attribute, i.e. an attribute that determines the order of storage, which must comprise values that are scalar, homogeneous and non-null. Therefore, it is marked with - (row D) and we do not discuss it further.

	Database	1. complex	2. type_mismatch	3. null_lt_null	4. null_lt_value	5. missing_lt_missing	6. missing_lt_value	7. null_lt_missing	8. type_order
A	Hive	b	-	f	t	-			t_A
B	Jaql	b	b	f	t	-			t_B
C	Pig	e	-	f	t	-			t_C
D	CQL	-							
E	JSONiq	e	e	f	t	f	f	t	t_E
F	MongoDB	b✻	b	f	t	f	t	f	t_F
G	N1QL	b	b	f	t	f	t	f	t_G
H	SQL	-	-	f	t	-			t_H
I	AQL	e	e	f	t	-			t_I
J	BigQuery	e	-	f	t	-			t_J
K	Mongo JDBC	b	b	f	t	f	t	f	t_K

$t_A = t_C = t_E = t_H = t_I = t_J =$ `[...]`
$t_B =$ `[array, tuple, boolean, string, number]`
$t_F =$ `[number, string, tuple, array, boolean]`
$t_G =$ `[boolean, number, string, array, tuple]`
$t_K =$ `[number, string, tuple, array, boolean]`

Table 11: Feature Matrix for Ordering on Non-Scalar and Heterogeneous Values

1. Complex Values: Pig supports complex values for grouping, but throws an error for ordering (C1). Conversely, Mongo JDBC supports complex values for ordering, but throws an error for grouping (K1).

MongoDB is denoted with b✻ (F1), as it returns boolean results that are inexplicable across multiple experiments.

2. Type Mismatch: Jaql, MongoDB and N1QL orders two values of different types utilizing `type_order` (B2, F2, G2). This is consistent with grouping two values of different types into separate equivalence groups.

JSONiq supports type mismatches for grouping, but throws an error for ordering (E2). Conversely, Mongo JDBC supports type mismatches for ordering, but throws an error for grouping (K2).

3-4. Null Values: All databases order `null` values identically as SQL's `ORDER BY ... NULL FIRST` (H3-4).

5-7. Missing Values: JSONiq orders `missing` as the biggest value (E5-7), whereas MongoDB, N1QL and Mongo JDBC orders `missing` as the smallest value (F5-7, G5-7, K5-7). This is consistent with grouping `null`

[15] Indeed, due to JSONiq's XQuery legacy, automatic order preservation is the default. Nevertheless, JSONiq goes beyond XQuery behavior by supporting an `unordered` flag that allows one to disable the default order preservation and then the output order is nondeterministic. Generally, nondeterministic order, which in turn creates more opportunity for query optimization.

```
1  bag_op_params    →
2    {
3      [bag_mismatch         : (heterogeneous|error) ,]
4      [coerce_null          : (singleton_bag|empty_bag|error) ,]
5      [coerce_missing       : (singleton_bag|empty_bag|error) ,]
6      [coerce               : (singleton_bag          |error) ,]
7      [complex              : (boolean|error) ,]
8      [type_mismatch        : (false  |error) ,]
9      [null_eq_null         : (true   |error) ,]
10     [null_eq_value        : (false  |error) ,]
11     [missing_eq_missing   : (true   |error) ,]
12     [missing_eq_value     : (false  |error) ,]
13     [null_eq_missing      : (true   |false|error) ]
14   }
```

Fig. 25: BNF Grammar for Bag-Op Config Parameters

and `missing` into separate equivalence groups. (Caveat: Recall that Mongo JDBC confusingly groups `null` and `missing` together.)

8. Type Order: Jaql, MongoDB, N1QL and Mongo JDBC support a total order when ordering values of different types (B8, F8, G8, K8).

Since Mongo JDBC delegates sorting to MongoDB, it has the same total order as MongoDB (F8, K8).

4.11 `UNION` / `INTERSECT` / `EXCEPT` clauses

The surveyed databases support to varying degrees SQL's `UNION ALL`, `INTERSECT ALL` and `EXCEPT ALL` bag operators, as well as their duplicate-eliminating counterparts the `UNION`, `INTERSECT` and `EXCEPT`. While SQL bag operators always input/output bags of flat, homogeneous tuples, queries in the surveyed databases input/output arbitrary values, such as collections of nested heterogeneous tuples.

4.11.1 SQL++ Formalism

As in SQL, a bag operator[16] is specified with: $q\ S\ q'$ (Figure 6, lines 9-10), which comprises a left query q, the bag operator S and a right query q'. The bag operator S may be `UNION`, `INTERSECT` or `EXCEPT` and may be optionally suffixed with `ALL`. Let $q \rightarrow v$ and $q' \rightarrow v'$, where v and v' are arbitrary values. The `ALL` variations input bags and output a bag of values, which may have duplicate values even if the input does not. Presence of `ALL` specifies that the output may have duplicate elements, while absence requires duplicate elimination, in which case the output is an implicit set.

The semantics of a bag operator is specified by a *bag-op config* (Figure 6, line 51). Figure 25 shows the BNF grammar of bag-op config parameters, and Figure 26 shows the semantics of bag operators with respect to bag-op config parameters. The `bag_mismatch` parameter is applicable when inputs v and v' are bags, and elements of v are heterogeneous with respect to elements of v' (Figure 25, line 3). It specifies whether to output: a heterogeneous bag, or throw an error. The `coerce_null` parameter specifies how to coerce `null` values for v and v': `singleton_bag` specifies coercing from `null` to `{{ null }}`, `empty_bag` specifies coercing to `{{ }}`, `error` specifies throwing an error (line 4). The `coerce_missing` parameter analogously specifies how to coerce `missing` values (line 5). The `coerce` parameter specifies whether to coerce a scalar/tuple/map into a singleton bag, or throw an error (line 6). Note that arrays for v and v' are always coerced into bags by discarding ordinal positions. The remaining seven parameters (lines 7-13) specify the equality semantics used internally by the bag operator (i.e. $\overset{@}{=}$ in Figure 26), and are identical to the seven group-by config parameters (Section 4.9, Figure 21). For example, the semantics for SQL's bag operators are specified with the following bag-op config parameters: `@bag_op {bag_mismatch: error, null_eq_null: true, null_eq_value: false}`. (Other parameters are not applicable since the SQL data model does not support complex values, missing values and heterogeneity.)

[16] SQL's bag operators are often referred to as set operators, while the term set is mathematically incorrect.

	Database	1. UNION ALL	2. INTERSECT ALL	3. EXCEPT ALL	4. UNION	5. INTERSECT	6. EXCEPT
A	Hive	✓	×	×	×	×	×
B	Jaql	✓	×	×	×	×	×
C	Pig	✓	×	×	×	×	×
D	CQL	×	×	×	×	×	×
E	JSONiq	✓	×	×	×	×	×
F	MongoDB	×	×	×	×	×	×
G	N1QL	×	×	×	×	×	×
H	SQL	✓	✓	✓	✓	✓	✓
I	AQL	×	×	×	×	×	×
J	BigQuery	✓	×	×	×	×	×
K	Mongo JDBC	✓	✓	✓	×	×	×

Table 12: Feature Matrix for Bag Operators

4.11.2 Classifying Bag Operators on Non-Scalar and Heterogeneous Values

Table 12 classifies each database's support for bag operators independent of their semantics, whereas Table 13 classifies each database's semantics for bag operators on non-scalar and heterogeneous values in terms of a SQL++ bag-op config. Any feature that is classified as × in Table 12 is denoted as - (i.e. inapplicable) in the corresponding columns of Table 13:

In Table 12:

$$
x\ \texttt{BAGOP}\ y \rightarrow \begin{cases}
f_{\texttt{BAGOP}}(x, y) & \text{if } x \text{ and } y \text{ are bags, } x \uplus y \text{ is homogeneous} \\
f_{\texttt{BAGOP}}(x, y) & \text{if } x \text{ and } y \text{ are bags, } x \uplus y \text{ is heterogeneous, } \texttt{@bag\_op.bag\_mismatch} \text{ is } \texttt{heterogeneous} \\
\texttt{error} & \text{if } x \text{ and } y \text{ are bags, } x \uplus y \text{ is heterogeneous, } \texttt{@bag\_op.bag\_mismatch} \text{ is } \texttt{error} \\
\hline
\text{array\_to\_bag}(x)\ \texttt{BAGOP}\ y & \text{if } x \text{ is an array} \\
x\ \texttt{BAGOP}\ \text{array\_to\_bag}(y) & \text{if } y \text{ is an array} \\
\hline
\texttt{\{\{}\ x\ \texttt{\}\}}\ \texttt{BAGOP}\ y & \text{if } x \text{ is a scalar/tuple/map, } \texttt{@bag\_op.coerce} \text{ is } \texttt{singleton\_bag} \\
x\ \texttt{BAGOP \{\{}\ y\ \texttt{\}\}} & \text{if } y \text{ is a scalar/tuple/map, } \texttt{@bag\_op.coerce} \text{ is } \texttt{singleton\_bag} \\
\texttt{error} & \text{if } x \text{ is a scalar/tuple/map or } y \text{ is a scalar/tuple/map, } \texttt{@bag\_op.coerce} \text{ is } \texttt{error} \\
\hline
\texttt{\{\{ null \}\}}\ \texttt{BAGOP}\ y & \text{if } x \text{ is } \texttt{null}, \texttt{@bag\_op.coerce\_null} \text{ is } \texttt{singleton\_bag} \\
\texttt{\{\{ \}\}}\ \texttt{BAGOP}\ y & \text{if } x \text{ is } \texttt{null}, \texttt{@bag\_op.coerce\_null} \text{ is } \texttt{empty\_bag} \\
x\ \texttt{BAGOP \{\{ null \}\}} & \text{if } y \text{ is } \texttt{null}, \texttt{@bag\_op.coerce\_null} \text{ is } \texttt{singleton\_bag} \\
x\ \texttt{BAGOP \{\{ \}\}} & \text{if } y \text{ is } \texttt{null}, \texttt{@bag\_op.coerce\_null} \text{ is } \texttt{empty\_bag} \\
\texttt{error} & \text{if } x \text{ is } \texttt{null} \text{ or } y \text{ is } \texttt{null}, \texttt{@bag\_op.coerce\_null} \text{ is } \texttt{error} \\
\hline
\texttt{\{\{ missing \}\}}\ \texttt{BAGOP}\ y & \text{if } x \text{ is } \texttt{missing}, \texttt{@bag\_op.coerce\_missing} \text{ is } \texttt{singleton\_bag} \\
\texttt{\{\{ \}\}}\ \texttt{BAGOP}\ y & \text{if } x \text{ is } \texttt{missing}, \texttt{@bag\_op.coerce\_missing} \text{ is } \texttt{empty\_bag} \\
x\ \texttt{BAGOP \{\{ missing \}\}} & \text{if } y \text{ is } \texttt{missing}, \texttt{@bag\_op.coerce\_missing} \text{ is } \texttt{singleton\_bag} \\
x\ \texttt{BAGOP \{\{ \}\}} & \text{if } y \text{ is } \texttt{missing}, \texttt{@bag\_op.coerce\_missing} \text{ is } \texttt{empty\_bag} \\
\texttt{error} & \text{if } x \text{ is } \texttt{missing} \text{ or } y \text{ is } \texttt{missing}, \texttt{@bag\_op.coerce\_missing} \text{ is } \texttt{error}
\end{cases}
$$

$$
f_{\texttt{UNION ALL}}(x, y) \rightarrow x \uplus y
$$

$$
f_{\texttt{INTERSECT\_ALL}}(x, y) \rightarrow \begin{cases}
\texttt{\{\{ \}\}} & \text{if } x \text{ is } \texttt{\{\{ \}\}} \\
\{\{u\}\} \uplus f_{\texttt{INTERSECT\_ALL}}(x \backslash \{\{u\}\},\ y \backslash \{\{v\}\}) & \text{if } \exists u \in x,\ \exists v \in y,\ u \overset{@}{=} v \text{ is } \texttt{true} \\
f_{\texttt{INTERSECT\_ALL}}(x \backslash \{\{u\}\},\ y) & \text{if } \exists u \in x,\ \forall v \in y,\ u \overset{@}{=} v \text{ is not } \texttt{true}
\end{cases}
$$

$$
f_{\texttt{EXCEPT\_ALL}}(x, y) \rightarrow \begin{cases}
\texttt{\{\{ \}\}} & \text{if } x \text{ is } \texttt{\{\{ \}\}} \\
f_{\texttt{EXCEPT\_ALL}}(x \backslash \{\{u\}\},\ y \backslash \{\{v\}\}) & \text{if } \exists u \in x,\ \exists v \in y,\ u \overset{@}{=} v \text{ is } \texttt{true} \\
\{\{u\}\} \uplus f_{\texttt{EXCEPT\_ALL}}(x \backslash \{\{u\}\},\ y) & \text{if } \exists u \in x,\ \forall v \in y,\ u \overset{@}{=} v \text{ is not } \texttt{true}
\end{cases}
$$

$$
f_{\texttt{UNION}}(x, y) \rightarrow \delta(f_{\texttt{UNION ALL}}(x, y))
$$

$$
f_{\texttt{INTERSECT}}(x, y) \rightarrow \delta(f_{\texttt{INTERSECT ALL}}(x, y))
$$

$$
f_{\texttt{EXCEPT}}(x, y) \rightarrow \delta(f_{\texttt{EXCEPT ALL}}(x, y))
$$

Legend:

1. BAGOP denotes one of: UNION, UNION ALL, INTERSECT, INTERSECT ALL, EXCEPT, EXCEPT ALL.

2. $x \uplus y$ denotes bag (multiset) union.

3. $x \backslash y$ denotes bag (multiset) difference.

4. $x \overset{@}{=} y$ denotes:

```
@eq{
  complex           : @bag_op.complex,
  type_mismatch     : @bag_op.type_mismatch,
  null_eq_null      : @bag_op.null_eq_null,
  null_eq_value     : @bag_op.null_eq_value,
  missing_eq_missing : @bag_op.missing_eq_missing,
  missing_eq_value  : @bag_op.missing_eq_value,
  null_eq_missing   : @bag_op.null_eq_missing
}( x = y )
```

5. $\delta(b)$ denotes a bag b with duplicates removed using $\overset{@}{=}$

Fig. 26: Semantics for Bag Operators and their Config Parameters

1. `UNION ALL`: All surveyed databases support the `UNION ALL` operator, except for CQL, MongoDB, N1QL and AQL (D1, F1, G1 and I1). Note that the respective equivalents of `UNION ALL` in Jaql and JSONiq are order-preserving: the operator inputs arrays, and output an array that is the concatenation of the input arrays.

2-3. `INTERSECT ALL`, `EXCEPT ALL`: SQL and Mongo JDBC are the only databases to support the `INTERSECT ALL` / `EXCEPT ALL` operators (H2-3, K2-3).

4-6. `UNION`, `INTERSECT`, `EXCEPT`: SQL is the only database that support the duplicate-eliminating operators (H4-6). An obvious workaround in the surveyed databases is to combine `UNION ALL`, `INTERSECT ALL` and `EXCEPT ALL` operators with a `DISTINCT` keyword in the `SELECT` clause.

In Table 13:

Note that CQL, MongoDB, N1QL and AQL do not support bag operators, and are denoted with - since the config parameters are inapplicable (rows D, F, G, I). For conciseness, each cell shows only the first one/two letters of a parameter value.

1. Collection mismatch: When the two input values are collections and one is heterogeneous with respect to the other, Jaql, Pig and JSONiq support outputing heterogeneous collections (B1, C1, E1). Conversely, Hive, SQL and BigQuery throw errors (A1, H1, J1). Mongo JDBC partially supports outputting a het-

	Database	1. bag_mismatch	2. coerce_null	3. coerce_missing	4. coerce	5. complex	6. type_mismatch	7. null_eq_null	8. null_eq_value	9. missing_eq_missing	10. missing_eq_value	11. null_eq_missing
A	Hive	e	-	-	-				-			
B	Jaql	h	eb	-	e				-			
C	Pig	h	-	-	-				-			
D	CQL						-					
E	JSONiq	h	sb	eb	sb				-			
F	MongoDB						-					
G	N1QL						-					
H	SQL	e	-	-	-	-	-	t	f		-	
I	AQL						-					
J	BigQuery	e	-	-	-				-			
K	Mongo JDBC	h*	-	-	-	e	f	t	f	t	f	t

Table 13: Feature Matrix for Bag-Op Config Parameters

erogenous bag (K1). When the two input bags contain tuples with the same attribute names, and the tuples are heterogeneous in the narrow sense that an attribute name maps to values of different types across tuples, Mongo JDBC supports outputting a heterogenous bag. Otherwise, for all other hetereogeneous input, Mongo JDBC throws an error.

2-4. Coercions: Both Jaql and JSONiq support coercions, but with different semantics. Jaql coerces `null` into an empty bag (B2), does not support `missing` (B3) and throws an error when the input is a scalar/tuple (B4). Whereas JSONiq coerces `null`, scalars and tuples into a singleton bag (E2,4), and `missing` into an empty bag (E3). Hive, Pig, SQL, BigQuery and Mongo JDBC are denoted with - (A2-4, C2-4, H2-4, J2-4, K2-4) since they do not support the `missing` value, or only support SFW queries as arguments to bag operators, and these SFW queries always output collections (instead of `null`, `missing`, scalars or tuples).

5-11. Equality config parameters: All bag operators utilize equality internally, with the exception of `UNION ALL`. But since only SQL and Mongo JDBC support bag operators beyond `UNION ALL` (Table 12), all other databases are denoted with - for config parameters that correspond to equality. In SQL, the equality semantics for the bag operators are consistent with that of the `GROUP BY` clause (H5-11), as the `IS NOT DISTINCT FROM` function is used internally within both the `GROUP BY` clause and the bag operators (see Section 4.9.3, Table 9). In Mongo JDBC however, the equality semantics for the bag operators are inconsistent with those of the `GROUP BY` clause (K5-11). The `INTERSECT`/`EXCEPT` bag operator is more lenient than `GROUP BY`, as the bag operator considers two values of different types to be non-equal (K6), whereas `GROUP BY` throws an error when grouping values of different types.

```
1  equal_params  →
2   {
3     |complex             : (boolean   |error) ,]
4     |type_mismatch       : (false|null|error) ,]
5     |null_eq_null        : (true |null|error) ,]
6     |null_eq_value       : (false|null|error) ,]
7     |missing_eq_missing  : (true |null|missing|error) ,]
8     |missing_eq_value    : (false|null|missing|error) ,]
9     |null_eq_missing     : (false|null|missing|error) ]
10  }
```

Fig. 27: BNF Grammar for Equality Config Parameters

4.12 Equality

The `=` equality function in SQL only compares scalar values, whereas the surveyed databases support (i) comparisons between complex values (e.g. `WHERE co = [0.3, 0.4]`) (ii) comparisons between values of different types due to heterogeneous collections (e.g. `WHERE co > 0.4 OR co = 'C'`). To classify the different equality semantics, SQL++ extends SQL with equality config parameters that specify the semantics of the equality function.

4.12.1 SQL++ Formalism

The semantics of the `=` equality function is specified by an *equality config* (Figure 6, line 49). Figure 27 shows the BNF grammar of the equality config parameters, which are reminiscent of group-by config parameters (Section 4.9, Figure 21). The differences are:

1. x=y can be specified to return `null` when x and y are different types (line 4).
2. x=`null` can be specified to return `null` (lines 5-6).
3. x=`missing` can be specified to return `null` or `missing` (lines 7-9).

For example, SQL's semantics for equality is specified with the following equality config: `@eq {type_mismatch: error, null_eq_null: null, null_eq_value: null}`. (Other parameters are not applicable since the SQL data model does not support complex or missing values.) The evaluation semantics for the `=` equality function are virtually identical to those of the $\equiv$ identity test function (Section 4.9, Figure 22) with two simple substitutions: (1) `@eq` instead of `@group_by` (2) `=` instead of $\equiv$.

Notice the bottom four definitions for the equality of complex values in Figure 22. Equality of two arrays (resp. bags/tuples/maps) x and y is defined as the conjunction (i.e. `AND`) of all pairwise `=` comparisons of their elements (resp. elements/attributes/keys).

	Database	1. complex	2. type_mismatch	3. null_eq_null	4. null_eq_value	5. missing_eq_missing	6. missing_eq_value	7. null_eq_missing
Ax	Hive (=)	e	e	n	n	-		
Ay	Hive (<=>)	e	e	t	f	-		
B	Jaql	b	n	n	n	-		
C	Pig	b*	e	n	n	-		
D	CQL	e	e	e	f*	-		
Ex	JSONiq (=)	e	e	t	f	m	m	m
Ey	JSONiq (deep-equal)	b	f	t	f	t	f	f
F	MongoDB	b	f	t	f	t	f	f
G	N1QL	b	f	n	n	m	m	m
H	SQL	-	e	n	n	-		
I	AQL	e	e	n	n	-		
J	BigQuery	e	e	n	n	-		
K	Mongo JDBC	b	f	t	f	-	f	f

Table 14: Feature Matrix for Equality

Therefore, x=y will return `null` when one or more pairwise comparisons return `null`/`missing`, as specified by SQL++'s ternary logic (Section 4.7). For example, `[1, null] = [1, null]` → `null` given the following equality config: `@eq {complex: boolean, null_eq_null: null}`. This behavior is extrapolated from SQL's usage of `null` to denote an unknown value: a complex value that contains one or more `null` values is also considered unknown for the purpose of equality comparisons.

4.12.2 Classifying Equality on Non-Scalar and Heterogeneous Values

Table 14 classifies each database's semantics for equality on non-scalar and heterogeneous values in terms of a SQL++ equality config. Each database supports a single `=` equality function, with notable exceptions Hive and JSONiq: Hive supports both `=` and `<=>` (rows Ax, Ay), whereas JSONiq supports both `=` and `deep-equal` (rows Ex, Ey). Each equality function of each database is classified separately.

We shade a cell gray in Table 14 if it is identical to its corresponding cell in Table 9 (Section 4.9.3), and comment on the inconsistencies of non-shaded cells. For conciseness, each cell shows only the first letter of a parameter value. Partial support is denoted with *, whereas inapplicability is denoted with -. For example, SQL is denoted with - in Column 1 (H1), since it lacks support for complex values, and its semantics never utilize `@eq.complex`.

1. Complex Values: Jaql, JSONiq (using `deep-equal`), MongoDB, N1QL and Mongo JDBC are the only databases that fully support deep equality on complex values (B1, Ey1, F1, G1, K1). Pig only supports deep equality for maps and tuples, but not bags (C1).

Surprisingly, Hive is more lenient for grouping than equality for complex values (Ax1, Ay1): whereas grouping creates different equivalence groups for complex values, equality throws an error.

The semantics of JSONiq's grouping are identical to the equality semantics of its `deep-equal` function (Ey2-7), except when comparing two complex values (Ey1). Whereas `deep-equal` returns `true`/`false`, both grouping and `=` (Ex1) throw an error.

Mongo JDBC mostly has the same equality semantics as MongoDB (F1-4, F6-7, K1-4, K6-7), which is an improvement over its grouping semantics which differ from MongoDB's.

2. Type Mismatch: Jaql, JSONiq (using `deep-equal`), MongoDB, N1QL and Mongo JDBC are also the only databases that do not throw an error when comparing two values of different types, returning either `null` (B2) or `false` (Ey2, F2, G2, K2).

Jaql's equality semantics is nonetheless consistent with its grouping semantics of grouping two values of different types into separate equivalence groups (B2).

Hive, Pig, SQL and BigQuery throw an error when comparing two values of different types (Ax2, Ay2, C2, H2, J2). This is consistent with their non-support of heterogeneous collections, and non-support of grouping values of different types.

We also note a subtlety on the interaction between the `=` equality function and the `<>` inequality function. In N1QL, both `x = y` and `x <> y` return `false` when `x` and `y` have different types. This breaks the equivalence `x <> y` ≡ `not(x = y)`, which is an equivalence preserved in SQL and all other surveyed databases.

3-4. Null Values: Hive (using `<=>`), JSONiq, MongoDB and Mongo JDBC return `true` when comparing two `null` values (Ay3, E3, F3, K3), and `false` when comparing `null` with a value that is not `null`/`missing` (Ay4, E4, F4, K4).

Many other databases, namely Hive (using `=`), Jaql, Pig, N1QL, AQL and BigQuery follow SQL in returning `null` when one of the compared values is `null` (Ax3-4, B3-4, C3-4, G3-4, H3-4, I3-4, J3-4). This is nonetheless consistent with SQL's grouping semantics.

CQL also returns `false` when comparing `null` with a value that is not `null`/`missing` (D4), but only supports the equality function in the `WHERE` clause (instead of all clauses). CQL throws an error during query compilation for conditions of the form `WHERE x = null` (D3).

5-7. Missing Values: Recall from Section 4.3 that only JSONiq, MongoDB, N1QL and Mongo JDBC support the `missing` value, thus all other databases are

```
1  less_than_params    →
2    {
3      [complex            : (boolean       |error) ,]
4      [type_mismatch      : (boolean   |null|error) ,]
5      [null_lt_null       : (     false|null|error) ,]
6      [null_lt_value      : (true|false|null|error) ,]
7      [missing_lt_missing : (     false|null|missing|error) ,]
8      [missing_lt_value   : (true|false|null|missing|error) ,]
9      [null_lt_missing    : (true|false|null|missing|error) ]
10     [type_order         : [ type_name , ...] ]
11   }
```

Fig. 28: BNF Grammar for Less-Than Config Parameters

denoted with -. MongoDB considers a `missing` value to equal another `missing` value (F5) but not `null` and other values (F6-7), whereas N1QL returns `missing` when comparing `missing` with any value (G5-7). Notice that N1QL's equality semantics is nonetheless consistent with its grouping semantics.

JSONiq behaves like MongoDB when using `deep-equal` (Ey5-7), yet behaves like N1QL when using `=` (Ex5-7). This is surprising as one would expect the two equality functions to differ only on shallow versus deep equality (Ex1-2, Ey1-2), and not when comparing `missing` values.

Similar to MongoDB, Mongo JDBC also considers `missing` to not equal `null` and other values (K6-7), but limitations in the current version of Mongo JDBC prevented us from testing the comparison of two `missing` values (K5).

4.13 Less-Than Comparisons

Analogous to equality, SQL++ extends SQL with less-than config parameters to specify the semantics of the `<` less-than function.

4.13.1 SQL++ Formalism

The semantics of the `<` less-than function[17] is specified by a *less-than config* (Figure 6, line 50). Figure 28 shows the BNF grammar of the less-than config parameters, which are reminiscent of order-by config parameters (Section 4.10, Figure 23). The differences are:

1. x`<`y can be specified to return `null` when x and y are different types (line 4).
2. `null<`y can be specified to return `null` (lines 5-6).
3. `missing<`y can be specified to return `null` or `missing` (lines 7-9).

For example, SQL's semantics for less-than is specified with the following less-than config: `@lt {type_mismatch: error, null_lt_null: null, null_lt_value: null}`. (Other parameters are not applicable since the SQL data model does not support complex or missing values.) The evaluation semantics for the `<` less-than function are virtually identical to those of the `ORDER BY` $<^o$ function (Section 4.10, Figure 24) with two simple substitutions: (1) `@lt` instead of `@order_by` (2) `<` instead of $<^o$.

Notice the $f_{\texttt{array}}$ definition in Figure 24, which handles `null` within complex values in an analogous fashion as equality comparisons (Section 4.12). A less-than comparison of two arrays (analogously bags/tuples/maps) x and y is defined as the corresponding pairwise `<` comparison for the first pairwise `=` comparison that is not `true`. For example, `[1, null] < [1, null]` → `null` given the following less-than config: `@lt {complex: boolean, null_lt_null: null}`. This behavior is extrapolated from SQL's usage of `null` to denote an unknown value: a complex value that contains one or more `null` values is also considered unknown for the purpose of less-than comparisons.

[17] A less-than config is analogously applicable to the functions `>`, `<=` and `>=`.

	Database	1. complex	2. type_mismatch	3. null_lt_null	4. null_lt_value	5. missing_lt_missing	6. missing_lt_value	7. null_lt_missing	8. type_order
A	Hive	e	e	n	n	-			-
B	Jaql	b✻	n	n	n	-			-
C	Pig	e	e	n	n	-			-
D	CQL	e	e	e	e	-			-
E	JSONiq	e	e	f	t	f	✻	✻	-
F	MongoDB	b✻	b	f	t	f	t	f	t_F
G	N1QL	b	b✻	n	n	m	m	m	✻
H	SQL	-	e	n	n	-			-
I	AQL	e	e	n	n	-			-
J	BigQuery	e	e	n	n	-			-
K	Mongo JDBC	b	b	f	t	-	t	f	t_K

t_F = [number, string, tuple, array, boolean]
t_K = [number, string, tuple, array, boolean]

Table 15: Feature Matrix for Less-Than Comparisons

4.13.2 Classifying Less-Than Comparisons on Non-Scalar and Heterogeneous Values

Table 15 classifies each database's semantics for less-than comparisons on non-scalar and heterogeneous values in terms of a SQL++ less-than config. We shade a cell gray in Table 15 if it is identical to its corresponding cell in Table 11 (Section 4.10.3), and focus the following discussion only on non-shaded cells.

For conciseness, each cell shows only the first letter of a parameter value. Partial support is denoted with ✻, whereas inapplicability is denoted with -.

1. Complex Values: Surprisingly, Hive supports ordering on complex values, but throws an error for less-than comparisons (A1).

Jaql is denoted with `b✻` (B1), as it returns boolean results that are inexplicable across multiple experiments. This is also surprising since Jaql provides ordering on complex values.

CQL throws an error for less-than comparisons of complex values (D1). This is consistent with its non-support of ordering on complex values.

2. Type Mismatch: Hive, Pig, CQL, SQL and BigQuery throw an error on less-than comparisons of two values with different types (A2, C2, D2, H2, J2). This is consistent with their non-support of heterogeneous collections, and non-support of ordering values of different types.

Jaql returns `null` on less-than comparisons of two values with different types (B2). This is consistent with its support of ordering two values with different types.

N1QL is denoted as having partial support for type mismatches (G2), as it exhibits the following inexplicable behavior. For two values `x` and `y` with different types, both `x < y` and `y < x` return `false`.

3-4. Null Values: Across all databases, the semantics for less-than comparisons on `null` values are consistent with that of ordering `null` values.

In addition, the semantics for less-than comparisons on `null` values are also consistent with that of equality comparisons on `null` values (Section 4.12.2). The only exception is CQL (D4), which throws an error on less-than comparisons between `null` and another value that is not `null`/`missing`.

5-7. Missing Values: JSONiq is denoted as having partial support for less-than comparisons on `missing` values (E5-7), as it exhibits the following inexplicable behavior. For any value of `x`, both `x < missing` and `missing < x` return `false`.

N1QL returns `missing` for less-than comparisons that involve `missing` (G5-7). This is consistent with its ordering of `missing` values. It is also consistent with its equality comparisons of `missing` values (Section 4.12.2).

Limitations in the current version of Mongo JDBC prevented us from testing the less-than comparison of two `missing` values (K5).

8. Type Order: MongoDB and Mongo JDBC are the only databases that support a total order between values of different types for less-than comparisons. The total order is denoted by t_F and t_K (F8, K8), which is the same total order as when ordering values of different types.

N1QL is denoted as having partial support, since it returns `true`/`false` for type mismatches (G2), but the total order between two values with different types is inexplicable.

All other databases do not return `true`/`false` for type mismatches (A2, B2, C2, D2, E2, H2, I2, J2), thus the total order is inapplicable.

	Language	1. External Language	2. MapReduce	3. Functions (Built-in)	4. Functions (Plugins)
A	Hive	✓	✓	✓	
B	Jaql	✓	✓		
C	Pig	✓	✓		
D	CQL				
E	JSONiq				
F	MongoDB	✓	✓		
G	N1QL				
H	SQL	✓	✓	✓	✓
I	AQL				
J	BigQuery			✓	
K	Mongo JDBC				

Table 16: Feature Matrix for Extensibility

5 Extensibility

Beyond the query capabilities of each language, extensibility features allow developers to workaround limitations of the language and/or data model. Table 16 presents various extensibility features of the surveyed languages. Nonetheless, the absence of ✓ in a cell leaves open the possibility of low-level APIs, undocumented features, extensions from third-party vendors etc. Unlike other languages that are yet to be standardized, SQL has been implemented in multiple databases, thus we use ✓ to indicate a feature being supported by at least one database (row H).

1. External Language: An external language such as Java or Python is typically more expressive than a query language, thus user-defined functions (UDFs) implemented in an external language can augment query capabilities of the host language. For example, Hive, Jaql and Pig support Java (A1, B1, C1), whereas Pig and MongoDB support JavaScript (C1, F1). As an example of a SQL engine with wide support for external languages (H1), Postgresql supports pgSQL, Tcl, Perl, Python, Java, PHP, R, Ruby, Scheme, and sh.

2. MapReduce: We refer to the MapReduce programming model as opposed to the eponymous system of Google. Since Hive, Jaql and Pig queries are executed in a Hadoop cluster, they also support running map/reduce functions within the cluster (A2, B2, C2). Parallel clusters other than Hadoop also support executing map/reduce functions. For example, SQL en-

gines such as Teradata SQL/MR [14] support calling map/reduce functions from a SQL query (H2), whereas MongoDB provides a proprietary map/reduce API (F2).

3. Functions (Built-in): Certain languages provide built-in functions for processing semi-structured data, without expanding these capabilities to the entire data model and/or query language. This is reminiscent of SQL engines that provide limited support for XML processing by storing XML as a custom data type, and providing functions for XPath processing. Hive and BigQuery support a subset of JSONPath [16], which adapts the XPath syntax for JSON data (A3, J3). Similarly, SQL engines such as Postgresql, VoltDB and MemSQL also support arrays and JSON as custom data types, and provide corresponding built-in functions to process them.

4. Functions (Plugins): Due to its academic roots and open source community, Postgresql is also well-known for its extensibility through plugins (i.e. extensions/modules) that are contributed by third-parties. For example, the `hstore` plugin provide a key-value store that is embedded within SQL, thus providing a custom data type and corresponding functions to store and query sparse, heterogeneous data.

Finally, we note that Table 16 focuses on features that augment the expressiveness of each surveyed language. We have intentionally omitted middleware (such as the Mongo-Hadoop Connector or Cassandra/Hadoop integration) that provide an alternative interface to querying the underlying data, either by translations to the surveyed language or bypassing it altogether by utilizing a low-level API. Sections 3-4 have shown that middleware (such as Mongo JDBC) can provide both enhancements and limitations over its underlying sources, and thus should be considered a language in its own right with unique query capabilities.

6 Future Work

We expect SQL++ and the classification methodology of this survey to be extensible for comparisons of even more features of semi-structured query languages. Features that are well-known in mainstream query languages can nevertheless interact with other semi-structured data model / query language features, thereby providing different design options for language semantics. A few examples include:

- **Aggregation Functions:** Certain SQL databases (e.g. Postgresql) support user-defined aggregate functions. These aggregate functions can follow the SQL specification of inputting only non-`null` values, or they can be customized to input all values.
- **Existential/Universal Quantification:** SQL supports existential and universal quantifiers `EXISTS`, `IN`, `ANY` and `ALL`.
- **Window Functions:** SQL 2003 supports window functions, which are functions evaluated over a window frame subset of an input collection.
- **Recursion** SQL supports fixed-point computation through recursive queries, whereas XQuery similarly supports recursive functions.
- **Coercion:** SQL supports using parentheses to coerce a table comprising a single row and single attribute into a scalar value. Also, XQuery supports a rich set of type conversion functions which are used extensively in the language's semantics.
- **Transitive Path Steps:** XPath supports the descendant-axis path step that navigates transitively through an unbounded number of collections.

References

1. S. Abiteboul, P. C. Fischer, and H.-J. Schek, editors. *Nested Relations and Complex Objects, Papers from the Workshop "Theory and Applications of Nested Relations and Complex Objects", Darmstadt, Germany, April 6-8, 1987*, volume 361 of *Lecture Notes in Computer Science*. Springer, 1989.
2. F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *DBPL*, pages 122–138, 1989.
3. A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.
4. K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.
5. A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1):68–79, 2000.
6. Cloudera Impala. `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html`.
7. Couchbase. `http://www.couchbase.com/`.
8. DB-Engines ranking. `http://db-engines.com/en/ranking`.
9. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.
10. 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.
11. E. Dumbill. What is Big Data?, 2012. `http://strata.oreilly.com/2012/01/what-is-big-data.html`.
12. D. Feinberg, M. Adrian, and N. Heudecker. Gartner magic quadrant for operational database management systems,

2013. http://www.gartner.com/technology/reprints.do?id=1-1M9YEHW&ct=131028&st=sb.
13. D. Florescu and G. Fourny. JSONiq: The history of a query language. *IEEE Internet Computing*, 17(5):86–90, 2013.
14. E. Friedman, P. M. Pawlowski, and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.
15. G. Jaeschke and H.-J. Schek. Remarks on the algebra of non first normal form relations. In *Proceedings of the ACM Symposium on Principles of Database Systems, March 29-31, 1982, Los Angeles, California, USA*, pages 124–138. ACM, 1982.
16. JSONPath. http://goessner.net/articles/JsonPath/.
17. J. G. Kobielus, S. Powers, B. Hopkins, B. Evelson, and S. Coyne. The Forrester Wave: Enterprise Hadoop solutions, 2012. http://www.forrester.com/Hadoop/fulltext/-/E-RES60755.
18. A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.
19. S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.
20. MemSQL. http://www.memsql.com/.
21. MongoDB. http://www.mongodb.org/.
22. Neo4j. http://www.neo4j.org/.
23. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.
24. K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases. *CoRR*, abs/1405.3631, 2014. http://arxiv.org/abs/1405.3631.
25. Protocol Buffers. http://developers.google.com/protocol-buffers/.
26. Redis. http://redis.io/.
27. 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/.
28. 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.
29. J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, K. Littlefield, and P. Tong. F1: the fault-tolerant distributed RDBMS supporting Google's ad business. In *SIGMOD Conference*, pages 777–778, 2012.
30. M. Stonebraker. New opportunities for New SQL. *Commun. ACM*, 55(11):10–11, 2012.
31. A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, pages 996–1005, 2010.
32. JDBC driver for MongoDB. http://www.unityjdbc.com/mongojdbc/mongosqltranslate.php.
33. VoltDB. http://voltdb.com/.
34. Wikipedia. List comprehension, 2014. Accessed Jan 31 2014. http://en.wikipedia.org/w/index.php?title=List_comprehension&oldid=593258069.
35. Wikipedia. Top type, 2014. Accessed Sep 30 2014. http://en.wikipedia.org/w/index.php?title=Top_type&oldid=627625444.

A FORWARD Query Processing Example

Figure 29 shows an example of how the FORWARD middleware evaluates queries over different databases with varying capabilities. Consider a PostgreSQL database and a MongoDB database, where the PostgreSQL database contains a `sensors` table and MongoDB contains a `measurements` array of JSON objects. The FORWARD query processor presents to its clients the virtual SQL++ views **V1** and **V2** of these databases. Notice that the virtual view **V2** of MongoDB is identical to its native data representation. Since the views are virtual, FORWARD query processor does not have a copy of the source data.

Suppose the client issues the federated query **Q**, which finds the average temperature reported by any reliably functioning sensor in a specific lat-long bounding box, where a sensor is deemed reliable only if none of its measurements are outside the range −40°F to 140°F. The query is decomposed into PostgreSQL and MongoDB subqueries that are efficient and compatible with the limited query capabilities of MongoDB. In particular, FORWARD first issues to PostgreSQL the query **Q1** that finds the ids of the sensors in the bounding box. Then, for each id, FORWARD issues to MongoDB the query **Q2** that tests whether the sensor is reliable and, if it is, it issues a second query **Q3** that finds the average of the temperature measurements. Notice that if MongoDB had supported nested queries, it would have been possible to issue a single MongoDB query for each id. Finally, the `coord_to_state()` function, which inputs coordinates and outputs the name of the corresponding state, is executed in the middleware.

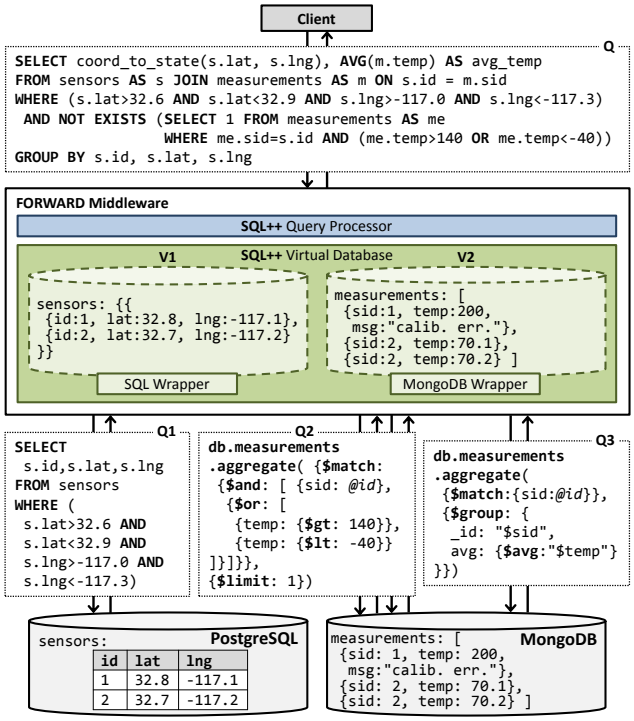

Fig. 29: FORWARD Query Processing Example

Internally, each query is translated into an algebraic plan, which extends an SQL algebraic plan in ways commensurate to the extensions of SQL into SQL++. Optimizing such a query plan involves a query optimizer that decides whether it is beneficial to first find the small set of sensor ids within the given bounding box, and then proceed to find measurements from MongoDB. Most interestingly, this optimizer must be aware of the limited query capabilities of the involved databases (as described in the feature matrices and configuration settings of this survey), so that the subqueries sent to a database are compatible with its capabilities.

B FORWARD Visualizations

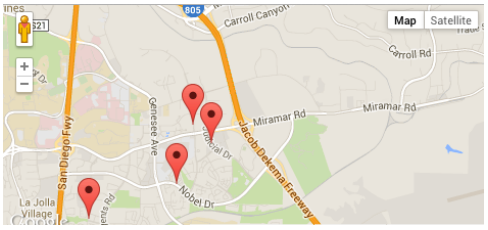

Fig. 30: Screenshot of FORWARD Visualization

```
<% define view cars as
    select id, lat, lng, packages
    from   geo_db.cars
           join delivery_db.orders using (car_id)
           join page.selected_cars using (car_id)
%>
<html>
<body>
  <% unit google.maps.Maps %>
  {
    markers : [
      <% for car in cars %>
        {
          position : {
            latitude  : <%= car.lat %>,
            longitude : <%= car.lng %>
          }
        }
      <% end for %>
    ]
  }
  <% end unit %>
</body>
</html>
```

Fig. 31: Complete Source Code for FORWARD Page