Semistructured Models, Queries and Algebras in the Big Data Era

Yannis Papakonstantinou on the contributions of UCSD’s database lab
Rana Alotaibi, Michalis Petropoulos, Kian Win Ong, Jules Testard, Romain Vernoux etal
the feedback of the AsterixDB and Couchbase teams
Yingyi Bu, Mike Carey, Don Chamberlin, Gerald Sangudi
and the lessons of 3 decades of SIGMOD/PODS research
on non-relational models and queries

with the support of National Science Foundation, Informatica & Couchbase

Semistructured Data have Arrived

SQL-on-Hadoop

NoSQL

Others

Asterix

Google

bigquery
mostly in the form of JSON...

```
{
  location: 'Alpine',
  readings: [
    {
      time: timestamp('2014-03-12T20:00:00'),
      ozone: 0.035,
      no2: 0.0050
    },
    {
      time: timestamp('2014-03-12T22:00:00'),
      ozone: 'm',
      co: 0.4
    }
  ]
}
```

In a genuine JSON database the data are collections of JSON elements (arrays, objects/tuples, scalars)

JSON has many close relatives...

- Object-Oriented Data Model
- Nested Relational Data model
- Google’s buffer protocols
- Parquet
- ...

Fundamentally, our discussion on JSON data can/will carry over to those semistructured formats
XML is also semistructured data...

- but with a key difference

Semistructured Query Languages come along...

Declarative + Expressive Power + Semistructured Data

Be like SQL – but for semistructured data
but in a Tower of Babel...

SELECT AVG(temp) AS tavg
FROM readings
GROUP BY sid

db.readings.aggregate(
{$group: { _id: "$sid",
            tavg: { $avg: "$temp" } }}
)

MongoDB

readings -> group by sid = $.sid
into { tavg: avg($.temp) };

Jaql

a = LOAD 'readings' AS
    (sid:int, temp:float);
b = GROUP a BY sid;
c = FOREACH b GENERATE
    AVG(temp);
DUMP c;

Pig

often in lieu of formal semantics...

![Stack Overflow logo]
with limited query abilities...

Growing out of key-value databases
or
SQL with special type UDFs

Major commercial systems still
far from the full-fledged power of prior
non-relational query languages – OQL, Xquery

SQL databases with JSON support come along also...

• In the form of JSON columns
• and query language extensions that enable
  • search and extraction from JSON columns
  • construction of JSON columns in the result
• but a “retrofit” job that beats the semistructured advantages
• SQL++: A Backwards-Compatible SQL, which can access both SQL and JSON data
  • Queries exhibit XQuery and OQL abilities, yet backwards compatible with SQL-92
    • If interested in a non-SQL-compliant transition of XQuery to JSON read JSONiq and/or the ASTERIXDB AQL
  
• Surveying the multiple and different languages and their semantics options with **Configurable SQL++**
  • Options modify the semantics of the language in order to morph into one of the languages

### Poll

I can write complex queries in

• OQL

• XQuery

• SQL
**SQL++**:
Data model + query language for querying both structured data (SQL) and semistructured data (native JSON)

**Formal Schemaless Semantics:**
Minimally adjusting tuple calculus to semistructured

**Full fledged, declarative, aggressively backwards-compatible with SQL**

**SQL++ Data Model**
(Also JSON++ data model)
A superset of SQL and JSON
- Extend SQL with arrays + nesting + heterogeneity

```json
{
  location: 'Alpine',
  readings: [
    {
      time: timestamp('2014-03-12T20:00:00'),
      ozone: 0.035,
      no2: 0.0050
    },
    {
      time: timestamp('2014-03-12T22:00:00'),
      ozone: 'm',
      co: 0.4
    }
  ]
}
```

- Array nested inside a tuple
- Heterogeneous tuples in collections
- Arbitrary compositions of array, bag, tuple
SQL++ Data Model
(also JSON++ data model)

A superset of SQL and JSON

- Extend SQL with arrays + nesting + heterogeneity + permissive structures

```
[ [5, 10, 3],
  [21, 2, 15, 6],
```

- Arbitrary compositions of array, bag, tuple

---

SQL++ Data Model
(also JSON++ data model)

A superset of SQL and JSON

- Extend SQL with arrays + nesting + heterogeneity + permissive structures

- Extend JSON with bags and enriched types

```
{
  location: 'Alpine',
  readings: {
    {
      time: timestamp('2014-03-12T20:00:00'),
      ozone: 0.035,
      no2: 0.0050
    },
    {
      time: timestamp('2014-03-12T22:00:00'),
      ozone: 'm',
      co: 0.4
    }
  }
}
```

- Bags {
  ...
}

- Enriched types
SQL++ Data Model
(also JSON++ data model)
A superset of SQL and JSON
• Extend SQL with arrays + nesting + heterogeneity
  + permissive structures + permissive schema
• Extend JSON with bags and enriched types

{ location: 'Alpine',
  readings: {{
    time: timestamp('2014-03-12T20:00:00'),
    ozone: 0.035,
    no2: 0.0050
  },
  { time: timestamp('2014-03-12T22:00:00'),
    ozone: 'm',
    co: 0.4
  }}
}

Comparisons

• OQL without schema

• Simpler than XML and the XQuery data model

• Unlike labeled trees (the favorite XML abstraction of XPath and XQuery research) makes the distinction between tuple constructor and list/array/bag constructor
Sample XML Document

```xml
<department>
  <project name="NexT" url="next.org"/>
  <project name="KadoP" url="kadop.net"/>
  <faculty>
    <person> <inproject>NexT</inproject> <inproject>KadoP</inproject> 
      <name> <first>John</first> <last>Smith</last> </name> 
      <mail>j@u.edu</mail>
    </person>
    <person> 
      <name> <first>Mary</first> <last>Jones</last> </name> 
      <mail>m@u.edu</mail> <mail>m@acm.org</mail>
    </person>
  </faculty>
  <students>
    <person> <inproject>KadoP</inproject> 
      <name> <first>Lily</first> <last>Liu</last> </name> <mail>lil@u.edu</mail>
    </person>
  </students>
</department>
```

Sample XML Document

```
Sample XML Document
```

```
Sample XML Document
```

```
Sample XML Document
```
From SQL to SQL++

• Goal: Queries exhibit XQuery and OQL abilities, yet backwards compatible with SQL-92
  • If interested in a non-SQL-compliant transition of XQuery to JSON read JSONiq and/or the ASTERIXDB AQL
• SQL Backwards-compatible because
  • easy to teach
  • easy to extend functionalities of SQL systems
• In the process of adoption by AsterixDB and Couchbase
• Adopted by UCSD’s FORWARD middleware
Backwards Compatibility with SQL

```
readings : {{
  { sid: 2, temp: 70.1 },
  { sid: 2, temp: 49.2 },
  { sid: 1, temp: null }
}}
```

```
SELECT DISTINCT r.sid
FROM readings AS r
WHERE r.temp < 50
```

Find sensors that recorded a temperature below 50

<table>
<thead>
<tr>
<th>sid</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>70.1</td>
</tr>
<tr>
<td>2</td>
<td>49.2</td>
</tr>
<tr>
<td>1</td>
<td>null</td>
</tr>
</tbody>
</table>

From SQL to SQL++

- Goal: Queries that input/output any JSON type, yet backwards compatible with SQL
- Adjust SQL syntax/semantics for heterogeneity, complex input/output values
- Achieved with few extensions and mostly by SQL limitation removals
  - OQL influence
- Minimal SQL++ core
  - SQL and the full SQL++ are syntactic sugar over a small SQL++ core
From Tuple Variables to Element Variables

**Semantics: Bindings to Element Variables**

- **readings:** 
  
  \[
  [1.3, \\
  0.7, \\
  0.3, \\
  0.8]
  \]

- **SELECT** 
  
  \[
  \text{co} \quad \text{AS} \quad r
  \]

- **FROM** 
  
  \[
  \text{readings} \quad \text{AS} \quad r
  \]

- **WHERE** 
  
  \[
  r < 1.0
  \]

- **ORDER BY** 
  
  \[
  r \quad \text{DESC}
  \]

- **LIMIT** 
  
  \[
  2
  \]

Find the highest two sensor readings that are below 1.0.
SQL++

```
SELECT r AS co
FROM readings AS r
WHERE r < 1.0
ORDER BY r DESC
LIMIT 2
```
No need for homogeneity of bindings

Correlated Subqueries Everywhere

Find the highest two sensor readings that are below 1.0
Correlated Subqueries Everywhere

FROM sensors AS s,
  s AS r

WHERE r < 1.0
ORDER BY r DESC
LIMIT 2
SELECT r AS co

sensors : [          
  [1.3, 2],        
  [0.7, 0.7, 0.9],  
  [0.3, 0.8, 1.1],  
  [0.7, 1.4]        
]
Correlated Subqueries Everywhere

FROM sensors AS s,
   s AS r
WHERE r < 1.0
ORDER BY r DESC

SELECT r AS co

B_{out}^{ORDERBY} = B_{in}^{LIMIT} = [ 
   \langle s: [0.7, 0.7, 0.9], r: 0.9 \rangle, 
   \langle s: [0.3, 0.8, 1.1], r: 0.8 \rangle, 
   \langle s: [0.7, 0.7, 0.9], r: 0.7 \rangle, 
   \langle s: [0.7, 1.4], r: 0.7 \rangle, 
   \langle s: [0.3, 0.8, 1.1], r: 0.3 \rangle 
]

LIMIT 2

B_{out}^{LIMIT} = B_{in}^{SELECT} = [ 
   \langle s: [0.7, 0.7, 0.9], r: 0.9 \rangle, 
   \langle s: [0.3, 0.8, 1.1], r: 0.8 \rangle 
]

Composability: A Powerful Idea from Functional Programming Languages

- Variables generate environments within which queries operate
- Initial environment provides named top-level values
- The OQL legacy: When evaluating a subquery names and variables behave no different
Constructing Nested Results

Discussion of Environments

\[ \Gamma_0 = \begin{cases} \text{sensors: } \{ \\
\{ \text{sensor: 1} \}, \\
\{ \text{sensor: 2} \} \\
\} , \\
\text{logs: } \{ \\
\{ \text{sensor: 1, co: 0.4} \}, \\
\{ \text{sensor: 1, co: 0.2} \}, \\
\{ \text{sensor: 2, co: 0.3} \} \\
\} \end{cases} \]

FROM sensors AS s

\[ B^{out}_{\text{from}} = B^{in}_{\text{select}} = \{ \begin{cases} \text{SELECT } l.co AS co \\
\text{FROM logs AS l} \\
\text{WHERE } l.sensor = s.sensor \\
\end{cases} \} \]

AS readings

SELECT s.sensor AS sensor,

{{
{s: \{sensor: 1\}},
{s: \{sensor: 2\}}
}}
How to deal with order:
Create order Vs Presume order

- SQL assumes bags as inputs but may output lists by using ORDER BY
  - not enough when input has order
- XQuery adopted order preservation semantics
  - eg, filtering a list will produce a list where qualified elements retain their original order
  - often becomes a burden
    eg, when order is dictated for join results

Extending SQL with Utilization of Input Order

```
SELECT r AS co, x AS x, y AS y
FROM sensors AS s
WHERE r < 1.0
ORDER BY r DESC
LIMIT 2
```

Find the highest two sensor readings that are below 1.0

```json
[{
co: 0.9,
x: 3,
y: 2
},
{
co: 0.8,
x: 2,
y: 3
}]
```
Order Preservation by Ordering according to the Input Order

sensors:
[
[1.3, 2]
[0.7, 0.7, 0.9]
[0.3, 0.8, 1.1]
[0.7, 1.4]
]

Find the first two sensor readings that are below 1.0

(What if) Automated Order Preservation

SELECT r AS co, x AS x, y AS y FROM sensors AS s AT y, s AS r AT x WHERE r < 1.0 ORDER BY y ASC, x ASC LIMIT 2
Iterating over Attribute-Value pairs

Constructing tuples with data-dependent number of Attribute-Value pairs
INNER Vs OUTER Correlation: Capturing Outerjoins, OUTER FLATTEN

\[
\Gamma = (\text{readings:{co : [], no2: [0.7], so2: [0.5, 2]}})
\]

FROM
\[
\text{readings AS \{g:v, v AS n\}}
\]

INNER CORRELATE
\[
\text{v AS n}
\]

SELECT ELEMENT
\[
\{ \text{gas: g, num: n } \}
\]

\[
[\{\text{gas: "no2", num: 0.7}\}, \{\text{gas: "so2", num: 0.5}\}, \{\text{gas: "so2", num: 2}\}]
\]
INNER Vs OUTER Correlation: Capturing Outerjoins, OUTER FLATTEN

\[
\Gamma = \{ \\
\text{readings:} \{ \\
\quad \text{co: [ ],} \\
\quad \text{no2: [0.7],} \\
\quad \text{so2: [0.5, 2]} \\
\} \\
\}
\]

\[
\{ \langle \text{v: [ ], n: missing} \rangle, \\
\langle \text{v: [0.7], n: 0.7} \rangle, \\
\langle \text{v: [0.5, 2], n: 0.5} \rangle, \\
\langle \text{v: [0.5, 2], n: 2} \rangle \}
\]

SELECT ELEMENT
\{ \text{gas: g,} \\
\text{num: n } \}

An OQLish GROUP BY

Alike OQL and unlike SQL's semantics, the SELECT clause semantics make sense just by knowing the incoming variables (and environment)
The SQL++ Extensions to SQL

- Goal: Queries that input/output any JSON type, yet backwards compatible with SQL

- Adjust SQL syntax/semantics for heterogeneity, complex values

- Achieved with few extensions and SQL limitation removals
  - Element Variables
  - Correlated Subqueries (in FROM and SELECT clauses)
  - Optional utilization of Input Order
  - Grouping independent of Aggregation Functions
  - ... and a few more details
What SQL aspects have been omitted

- SQL’s *
  - Could be done using ranging over attributes
  - It is easily replaced by ability to project in SELECT full tuples (or full anythings)
  - An optimizable implementation needs schema
    - Consider the view V = SELECT * FROM R r, S s, T t WHERE ...
    - How do you push a condition?
      - SELECT ... FROM V v WHERE v.foo = sth

- Selected SQL operations (UNION) that rely on attributes having a certain order in their tuples

What SQL aspects have been omitted ... (temporarily?)

- SQL’s Implicit Coercion
  - WHERE 5 = (SELECT ... FROM ... WHERE ...)
  - SQL coerces a tuple that contains a single attribute
    - 5 = [ {foo: 5} ]
    - Instead write WHERE 5 = tonum(SELECT ... FROM ... WHERE ...)

- Should we adopt two equalities a’ la XQuery?
  - A true deep equality, where 5 not equal to [ {foo: 5} ]
  - A coerced equality

- XQuery pushed coercion to its limits
  - Eg, equality of lists means that some elements in the lists are equal
  - x = y AND y = z does not imply x = z
Options, options, options...  
Esp. when we deal with absence of information and no typing

What if in

• FROM coll v
  • coll is not a bag/array (eg, is a tuple or a scalar)

• SELECT x.foo AS bar
  • x has no foo

• x.foo = x.bar
  • when different types or missing sides

• For good or bad many options have been taken!

The SQL/JSON way to incorporating access to JSON files and JSON columns

For those who really want their FROM clause to only have collections of tuples

• UDF’s and constructs for turning the file into a table
  • FROM ... json(file, pattern) AS j ... WHERE c AND c’
  • assuming the goal is relational results
  • pattern can escalate to a full blown SQL++ FROM
  • and WHERE assuming conditions c need to be explicitly pushed into json computations
    • think CORRELATE that has both structural and value based conditions

• Piggybacking on nested features of SQL for JSON columns
  • FROM ... json(t.jsoncol, pattern) AS j ...
From Language to Algebra as an extension of the Relational Algebra

- Operator SCAN for scanning FROM collections into bindings
- Operators for INNER CORRELATE, OUTER CORRELATE
- Operator for evaluating nested subqueries
- Operator for constructing (as needed in SELECT)
- Operators for picking the bindings that turn into an output collection or a tuple

How options complicate middleware
SQL++ Virtual and/or Materialized Views

FORWARD Integration Database

SQL++ Query Processor

SQL++ View Definitions

SQL++ Integrated Views

SQL++ Virtual View

SQL++ Virtual View

SQL++ Virtual View

SQL++ Virtual View

SQL++ Virtual View

SQL++ Added Value View

SQL Database

NewSQL Database

NoSQL Database

SQL-on-Hadoop Database

Java In-Memory Objects

Use Case 1:
Hide the ++ source features behind SQL views

SELECT m.sid, t AS temp
FROM measurements AS m,
    m.temperatures AS t

FORWARD Virtual Database

SQL View

measurements:

<table>
<thead>
<tr>
<th>sid</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>70.1</td>
</tr>
<tr>
<td>2</td>
<td>70.2</td>
</tr>
<tr>
<td>1</td>
<td>71.0</td>
</tr>
</tbody>
</table>

or its SQL++ equivalent

measurements: {{
    { sid: 2, temp: 70.1},
    { sid: 2, temp: 70.2},
    { sid: 1, temp: 71.0} }}

Couchbase query

measurements: {{
    { sid: 2, temperatures: [70.1, 70.2] },
    { sid: 1, temperatures: [71.0] } }}
Use Case 2: Semantics-Aware Pushdown

"Sensors that recorded a temperature below 50"

```
SELECT DISTINCT m.sid
FROM measurements AS m
WHERE m.temp < 50
```

```
{ $match: {temp: {$lt: 50}}},
{ $match: {temp: {$not: null}}}
```

- How to efficiently push down computation?
- Limited capabilities
  - Sources like SQL and MongoDB cannot execute the entirety of SQL++
- Many semantic variations
  - How to mediate incompatible features?
  - Step 1: understand the space of options!

Push-Down Challenges: Limited capabilities & semantic variations
• Semantics of "less-than" comparison are different across sources:
  \(<_{\text{sql}}, <_{\text{mongodb}}, \text{etc.}\)

• **Config Parameters** to capture these variations

```
@lt { MongoDB }
(x < y)
```

---

**Semantics Variations**

In NoSQL, NewSQL, SQL-on-Hadoop variation of semantics for:

• Paths
• Equality
• Comparisons

And all the operators that use them:

• Selections
• Grouping
• Ordering
• Set operations

Each of these features has a set of config parameters in Configurable SQL++
Survey around configuration options

- Fast evolving space
- Configuration options itemize and explain the space options
- Rationalize discussion towards standard
- And if no standard at least understand where we stand
- Survey

Surveyed Databases

SQL-on-Hadoop
- Hive
- Pig
- Jaql

NoSQL
- CQL
- MongoDB
- JSON
- JSONiq
- N1QL

Others
- AsterixDB
- Google BigQuery
- Unity JDBC
- MongoDB driver
SQL++ Removes Superficial Differences

- SQL++ covers sufficiently SQL, N1QL and QL research prototypes (e.g., UCI’s ASTERIX)
- Removing the current “Tower of Babel” effect
- Providing formal syntax and semantics

Surveyed features

15 feature matrices (1-11 dimensions each) classifying:

- Data values
- Schemas
- Access and construct nested data
- Missing information
- Equality semantics
- Ordering semantics
- Aggregation
- Joins
- Set operators
- Extensibility
Methodology

For each feature:
1. A formal definition of the feature in SQL++
2. A SQL++ example
3. A feature matrix that classifies each dimension of the feature
4. A discussion of the results, partial support and unexpected behaviors

All the results are empirically validated

Example: Data values

1. SQL++ example:

```sql
{
  location: 'Alpine',
  readings: [
    {
      time: timestamp('2014-03-12T20:00:00'),
      ozone: 0.035,
      no2: 0.0050
    },
    {
      time: timestamp('2014-03-12T22:00:00'),
      ozone: 'm',
      co: 0.4
    }
  ]
}
```
Example: Data values

2. SQL++ BNF for values:

```
1  value      →  defined_value
2  defined_value →  scalar_value
3  scalar_value →  complex_value
4  complex_value →  null
5  null
6  tuple_value →  collection_value
7  collection_value →  map_value
8  map_value

9  primitive_value →  primitive_value
10  primitive_value →  'string'
11  primitive_value →  number
12  primitive_value →  true
13  primitive_value →  false
14  primitive_value →  type ( primitive_value, ... )
15  type ( primitive_value, ... )
16  type ( primitive_value, ... )

17  array_value →  bag_value
18  bag_value →  [ value, ... ]
19  bag_value →  { { value, ... } }
20  bag_value →  { { value, ... } }
21  bag_value →  { { value, ... } }
```

Example: Data values

3. Feature matrix:

<table>
<thead>
<tr>
<th>Composability (top-level values)</th>
<th>Heterogeneity</th>
<th>Arrays</th>
<th>Bags</th>
<th>Sets</th>
<th>Maps</th>
<th>Tuples</th>
<th>Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hive</td>
<td>Bag of tuples</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Partial</td>
<td>Yes</td>
</tr>
<tr>
<td>Jaql</td>
<td>Any Value</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Pig</td>
<td>Bag of tuples</td>
<td>Partial</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>Partial</td>
<td>Yes</td>
</tr>
<tr>
<td>CQL</td>
<td>Bag of tuples</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>JSONiq</td>
<td>Any Value</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Bag of tuples</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>N1QL</td>
<td>Bag of tuples</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL</td>
<td>Bag of tuples</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>AQL</td>
<td>Any Value</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>BigQuery</td>
<td>Bag of tuples</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Bag of tuples</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL++</td>
<td>Any Value</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
4. Discussion of the results:

- Column-by-column comparison
- Partial support (65k scalar elements)
- Identify clusters (who supports JSON?)

<table>
<thead>
<tr>
<th>Database</th>
<th>Composability (top-level values)</th>
<th>Heterogeneity</th>
<th>Arrays</th>
<th>Bags</th>
<th>Sets</th>
<th>Maps</th>
<th>Tuples</th>
<th>Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hive</td>
<td>Bag of tuples</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Jaql</td>
<td>Any Value</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Pig</td>
<td>Bag of tuples</td>
<td>Partial</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CQL</td>
<td>Bag of tuples</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>Partial</td>
<td>Yes</td>
</tr>
<tr>
<td>JSONiq</td>
<td>Any Value</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Bag of tuples</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>N1QL</td>
<td>Bag of tuples</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL</td>
<td>Bag of tuples</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>AQL</td>
<td>Any Value</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BigQuery</td>
<td>Bag of tuples</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MongoDB JDBC</td>
<td>Bag of tuples</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQL++</td>
<td>Any Value</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Example: Data values

Example: SELECT clause

1. SQL++ example:

- Projecting nested collections:

```sql
SELECT TUPLE
  s.lat, s.long,
  (SELECT r.ozone
   FROM readings AS r
   WHERE r.location = s.location)
FROM
  sensors AS s
```

"Position and (nested) ozone readings of each sensor?"

- Projecting non-tuples:

```sql
SELECT ELEMENT ozone FROM readings
```

"Bag of all the (scalar) ozone readings?"
### Example: SELECT clause

#### 3. Feature matrix:

<table>
<thead>
<tr>
<th></th>
<th>Projecting tuples containing nested collections</th>
<th>Projecting non-tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hive</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>Jaql</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Pig</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>CQL</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>JSONiq</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Partial</td>
<td>Partial</td>
</tr>
<tr>
<td>N1QL</td>
<td>Partial</td>
<td>Partial</td>
</tr>
<tr>
<td>SQL</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>AQL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>BigQuery</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MongoDB</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SQL++</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Not well supported features
- 3 languages support them entirely (same cluster as for data values)

#### 4. Discussion of the results:
We use *config parameters* to encompass and compare various semantics of a feature:

- Minimal number of independent dimensions
- 1 dimension = 1 config parameter
- SQL++ formalism parametrized by the config parameters
- Feature matrix classifies the values of each config parameter

### Example: Paths

- Config parameters for tuple navigation:

```plaintext
@tuple_nav {
  absent   : missing,
  type_mismatch: error,
}
(x.y)
```
Example: Paths

• The feature matrix classifies, for each language, the value of each config parameter:

<table>
<thead>
<tr>
<th></th>
<th>Missing</th>
<th>Type mismatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hive</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>Jaql</td>
<td>Null</td>
<td>Error#</td>
</tr>
<tr>
<td>Pig</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>CQL</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>JSONiq</td>
<td>Missing#</td>
<td>Missing#</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Missing</td>
<td>Missing</td>
</tr>
<tr>
<td>N1QL</td>
<td>Missing</td>
<td>Missing</td>
</tr>
<tr>
<td>SQL</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>AQL</td>
<td>Null</td>
<td>Error</td>
</tr>
<tr>
<td>BigQuery</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>MongoJDBC</td>
<td>Missing#</td>
<td>Missing#</td>
</tr>
<tr>
<td>SQL++</td>
<td>@path</td>
<td>@path</td>
</tr>
</tbody>
</table>

Example: Equality Function

• Config parameters for equality:

```javascript
@eq {
    complex : error,
    type_mismatch : false,
    null_eq_null : null,
    null_eq_value : null,
    missing_eq_missing: missing,
    missing_eq_value : missing,
    missing_eq_null : missing
} ( x = y )
```
### Example: Equality Function

- The feature matrix classifies, for each language, the value of each config parameter:

  - No real cluster
  - Some languages have multiple (incompatible) equality functions
  - Some edge cases cannot happen due to other limitations (SQL has no complex values)

<table>
<thead>
<tr>
<th>Complex Type mismatch</th>
<th>Complex</th>
<th>Type mismatch</th>
<th>Null = Null</th>
<th>Null = Value</th>
<th>Missing = Missing</th>
<th>Missing = Value</th>
<th>Missing = Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hive (&lt;=, &lt;=&gt;)</td>
<td>Err, Err</td>
<td>Err, Null</td>
<td>Null, True</td>
<td>Null, False</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Pig</td>
<td>Boolean</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>JSONiQ (=, deep-equal())</td>
<td>Err, Boolean</td>
<td>Err, False</td>
<td>True, True</td>
<td>False, False</td>
<td>Missing, True</td>
<td>Missing, False</td>
<td>Missing, False</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Boolean</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>N1QL</td>
<td>Boolean</td>
<td>False</td>
<td>Null</td>
<td>Null</td>
<td>Missing</td>
<td>Missing</td>
<td>Missing</td>
</tr>
<tr>
<td>SQL</td>
<td>N/A</td>
<td>Err</td>
<td>Null</td>
<td>Null</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>AQL</td>
<td>Err</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>BigQuery</td>
<td>Err</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MongoJDBC</td>
<td>Boolean</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>N/A</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>SQL++</td>
<td>@equal</td>
<td>@equal</td>
<td>@equal</td>
<td>@equal</td>
<td>@equal</td>
<td>@equal</td>
<td>@equal</td>
</tr>
</tbody>
</table>

### How to use this survey?

- As a database user:
  - Understand the semantics of a (often underspecified) query language / be aware of the limitation of a database

- As a designer/architect of a database:
  - Produce formal specification of your query language
  - Align semantics with SQL's

- As a database researcher:
  - The results might change, but the survey methodology stays

- As a designer/architect of database middleware:
  - Understand what capability variations need to be encapsulated and simulated
The survey shows:

- The marketing clusters do not correspond to real capabilities
- Limited capabilities: matrices are sparse and fragmented (more pressure on source-specific rewriters and distributor)

The Future is Semi-Structured and Declarative

- Scalability
- Flexibility of Semistructured Data
- Power of Declarative:
  Automatic Optimization, View Maintenance
- The primary operational and the analytics queries out of semistructured, declarative platforms
Followup & Discussion

• Language and 4 systems along a few options http://arxiv.org/abs/1405.3631

  • A bit outdated

• Coming up: Extension with Algebra
**FORWARD: SQL++ Incremental View Maintenance and Application Visualization layer**

- The Incremental View Maintenance functionality:
  - SQL++ (Materialized) View Definition
    - SQL++ can also enable automatic Incremental View Maintenance!
      - With attention to replication of data in views
      - Opportunities by keys
    - Stream of inserts, deletes, updates on base data
    - Eg, Couchbase has a JSON web log showing
      `{{{user, list of displayed products } }},` and FORWARD produces materialized view
      `{ {product category, count, products: [{product, count}]}}}

- The Incremental View Maintenance module
  - Automatically and efficiently updates the materialized view to reflect the stream of changes

- CUSTOM dashboards, interactive pages & apps
  - The data models of visualization components (e.g. Google Maps) can be nicely captured with JSON models
  - The pages are SQL++ (JSON) views!
    - Mashups of the components views
  - SQL++ feeds and incrementally updates the page views

**Use case**

- From data to visualization with just SQL++ & markup
  - Ajax/Javascript visuals with no Ajax/Javascript mess
  - How to easily connect to today’s JS libraries
  - Custom Ajax visualizations & interfaces for IT personnel
(part of) the Google Map model

```javascript
<% unit google.maps.Maps %>
{
  markers: [ {
    position: {
      latitude : number,
      longitude: number
    }
  } ]
}
<% end unit %>
```

---

FORWARD: SQL++ Incremental View Maintenance and Application Visualization layer

- The Incremental View Maintenance functionality:
  - SQL++ (Materialized) View Definition
    
    Eg, Couchbase has a JSON web log showing
    ```json
    {{
      user,
      list of displayed products
    }}
    ```
    
    and FORWARD produces materialized view
    ```json
    {{
      product category,
      count,
      products: [{
        product,
        count
      }]
    }}
    ```
    
  - Stream of inserts, deletes, updates on base data
  
- The Incremental View Maintenance module
  
  - Automatically and efficiently updates the materialized view to reflect the stream of changes
  
  - SQL++ can also enable automatic Incremental View Maintenance!
    - With attention to replication of data in views
    - Opportunities by keys
Custom dashboards, interactive pages & apps

- The data models of visualization components (e.g. Google Maps) can be nicely captured with JSON models
- The pages are SQL++ (JSON) views!
  - Mashups of the components views
- SQL++ feeds and incrementally updates the page views

Use case

- From data to visualization with just SQL++ & markup
  - Ajax/JavaScript visuals with no Ajax/JavaScript mess
  - How to easily connect to today’s JS libraries
- Custom Ajax visualizations & interfaces for IT personnel

(for part of) the Google Map model

```{% unit google.maps.Maps %}
{
  markers: [ {
    position: {
      latitude : number,
      longitude: number
    }
  },
  ...
} ]
{% end unit %}```