SQL++ Query Language
and its use in the FORWARD framework

Semistructured Databases have Arrived
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: 'a',      
      co: 8.4    
    }  
  ]}
Semistructured Query Languages come along...

SQL: Expressive Power and Adoption

JSON: Semistructured Data

but in a Tower of Babel...

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

MongoDB

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

Pig

FOR $r in collection("readings")
group by $r.sid
return ( tavg: avg($r.temp) )

with limited query abilities...

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

Far from the full-fledged power of prior non-relational query languages – OQL, Xquery

*academic projects being the exception (eg. ASTERIX, FORWARD)
in lieu of formal semantics...

Query languages were not meant for Ninjas

Outline

• What is SQL++?

• How we use SQL++ in UCSD’s FORWARD?

• Introduction to the formal survey of NoSQL, NewSQL and SQL-on-Hadoop

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

Formal Schemaless Semantics: adjusting tuple calculus to the richness of JSON

Full fledged, declarative, aggressively backwards-compatible with SQL

Configurable SQL++: Formal configuration options towards choosing semantics, understanding repercussions

Unifying: 11 popular databases/QLs captured by appropriate settings of Configurable SQL++
How is it used in the Big Data industry

• ASTERIXDB
• CouchDB
heading towards SQL++

How do we use it in UCSD

Automatic Incremental View Maintenance
Middleware Query Optimization & Execution: FORWARD virtual database query plans based on SQL++ algebra

based on a schemaless SQL++ algebra

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

A superset of SQL and JSON

• Extend SQL with arrays + nesting + heterogeneity

<table>
<thead>
<tr>
<th>Location: 'Alpine', readings:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[time: timestamp('2014-03-12T20:00:00'), ozone: 6.035, no2: 0.0050]</td>
</tr>
<tr>
<td>[time: timestamp('2014-03-12T22:00:00'), co: 0.4]</td>
</tr>
</tbody>
</table>

• 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], [4, {lo: 3, exp: 4, hi: 7}, 2, 13, 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: '2014-03-12T20:00:00', ozone: 0.035, no2: 0.0050 ], [ time: '2014-03-12T22:00:00', ozone: 0.4, co: 0.4 ] ] }
```
- Bags [ [ ] ]
- Enriched types

Exercise: Why the SQL++ data model is a superset of the SQL data model?

- a SQL table corresponds to a JSON bag that has homogeneous tuples
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
  }]
}
```

• With schema
• Or, schemaless
• Or, partial schema

From SQL to SQL++

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

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

Find sensors that recorded a temperature below 50

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

Backwards Compatibility with SQL

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

Find sensors that recorded a temperature below 50

sid | temp
--- | ----
2   | 70.1
2   | 49.2
1   | null
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

From Tuple Variables to Element Variables

```
readings: [1.3, 0.7, 0.3, 0.8]
```

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

Find the highest two sensor readings that are below 1.0

```
[1.3, 0.7, 0.3, 0.8]
ORDER BY r DESC
LIMIT 2
```

Element Variables

```
FROM readings AS r
```

```
B^{\text{MAXIMUM}} = B^{\text{MAXIMUM}} = \left\{ (r: 1.3), (r: 0.7), (r: 0.3), (r: 0.8) \right\}
```

```
WHERE r < 1.0
```

```
B^{\text{MAXIMUM}} = B^{\text{MAXIMUM}} = \left\{ (r: 0.7), (r: 0.3), (r: 0.8) \right\}
```

```
ORDER BY r DESC
```

```
B^{\text{MAXIMUM}} = B^{\text{MAXIMUM}} = \left\{ (r: 0.8), (r: 0.7), (r: 0.3) \right\}
```

```
LIMIT 2
```

```
SELECT r AS co
```

```
[ co: 0.8 ], [ co: 0.7 ]
```

```
B^{\text{SELECT}} = B^{\text{SELECT}} = \left\{ (r: 0.8), (r: 0.7) \right\}
```

```
B^{\text{SELECT}} = B^{\text{SELECT}} = \left\{ (r: 0.8), (r: 0.7) \right\}
```

```
B^{\text{SELECT}} = B^{\text{SELECT}} = \left\{ (r: 0.8), (r: 0.7) \right\}
```

```
B^{\text{SELECT}} = B^{\text{SELECT}} = \left\{ (r: 0.8), (r: 0.7) \right\}
```
Correlated Subqueries Everywhere

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

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

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

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

Utilization of Input Order

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

Find the highest two sensor readings that are below 1.0

Constructing Nested Results – Subqueries Again

(SELECT 1.co AS co
FROM logs AS l
WHERE l.sensor = s.sensor) AS readings

ε =
    {sensors: {{
        sensor: 1,
        readings: {{
            co: 0.4
        },
        {co: 0.2}
    }},
    sensor: 2},
    logs: {{
        sensor: 1,
        co: 0.4,
        {co: 0.2},
        {co: 0.3},
    }}
}
Constructing Nested Results

\[
E_1 = 
\{ 
  s: \{ \text{sensor: 1}, \text{sensor: 2} \}, 
  \text{logs:} \{ 
    (\text{sensor: 1, co: 0.4}, \text{sensor: 1, co: 0.2}) 
  \} 
\} 
\]

\[
\text{SELECT 1.co AS co} 
\]

Iterating over Attribute-Value pairs

\[
Q = \{ \text{reading:} \{ \text{no2: 0.7, co: [0.5, 2]} \} \} 
\]

\[
\text{SELECT \{ gas: g, val: v \} FROM readings AS \{ g:v \} WHERE g = \{ \text{no2: 0.7, co: [0.5, 2]} \}} 
\]

INNER Vs OUTER Correlation:
Capturing Outerjoins, OUTER FLATTEN

\[
F = \{ \text{readings:} \{ \text{co: [0.7], no2: [0.5, 2]} \} \} 
\]

\[
\text{SELECT \{ gas: g, val: v \} FROM readings AS \{ g:v \} WHERE g = \{ \text{no2: 0.7, co: [0.5, 2]} \}} 
\]

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

\[
\{ 
  \text{gas: "co", num: null}, 
  \{ \text{gas: "no2", num: 0.7} \}, 
  \{ \text{gas: "so2", num: 0.5} \} 
\} 
\]

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

\[ \mathbb{A} = \langle \text{readings: \{co : [], no2: [0.7], so2: [0.5, 2] \}} \rangle \]

FROM readings AS \{g:v\}
INNER CORRELATE \v AS \n
SELECT ELEMENT \{ gas: g, val: v \}

\[ \{ \{ \text{gas: "no2", num: 0.7}, \{ \text{gas: "so2", num: 0.5}, \{ \text{gas: "co", num: 2} \}\} \} \]

INNER Vs OUTER Correlation: Capturing Outerjoins, OUTER FLATTEN

\[ \mathbb{A} = \langle \text{readings: \{co : [], no2: [0.7], so2: [0.5, 2] \}} \rangle \]

FROM readings AS \{g:v\}
OUTER CORRELATE \v AS \n
SELECT ELEMENT \{ gas: g, val: v \}

\[ \{ \{ \text{gas: "no2", num: 0.7}, \{ \text{gas: "so2", num: 0.5}, \{ \text{gas: "co", num: 2} \}\} \} \]

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
- Configurability: making SQL++ a unifying and configurable language
Algebras do not need schemaful inputs

\[ \Sigma_1 = \{ \\
\text{s: \{sensor: 1, sensors: \{ \\
\text{sensor: 1}, \\
\text{sensor: 2} \}, logs: \{ \\
\text{1: co: 0.4}, \\
\text{2: co: 0.2} \}} \} \} \]

**Outline**

- What is SQL++?
- How we use SQL++ in UCSD's FORWARD?
  - SQL++ uses in integration
- Introduction to the formal survey of NoSQL, NewSQL and SQL-on-Hadoop

**SQL++ based Virtual Database**
Use Case 1: Hide the ++ source features behind SQL views

Use Case 2: Semantics-Aware Pushdown
Push-Down Challenges: Limited capabilities & semantic variations

- How to simulate incompatible semantics/missing features?
- Sources like SQL and MongoDB cannot execute the entirety of SQL++
- How to efficiently push down computation?

Issues automatically handled by the query processor. Plenty of query rewriting problems, including novel ones on semi-structured operators.

SQL++ captures Semantics Variations

- Semantics of "less-than" comparison are different across sources: <sql, <mongodb, etc.

- **Config Parameters** to capture these variations

```
@lt {
  complex : deep,
  type_mismatch: false,
  null_lt_null: false,
  null_lt_value: false,
  ...
}( 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 SQL++
Don’t give a standard
Teach how to reach a standard

• Fast evolving space
• Standard would be premature
• Configuration options itemize and explain the space options
• Rationalize discussion

Advise on consistent settings of the configuration options

• Not every configuration option setting is sensible
• Configuration options have to subscribe to common sense
  \[ \text{iff } x = y \text{ then } [x] = y \]
  \[ \text{if } x = y \text{ and } y = z \text{ then } x = z \]
  • which enables rewritings
• Have to be consistent with each other
  Eg, if true AND missing is missing, then false OR missing is missing

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
  \{ \{ user, list of displayed products \} \}
  and FORWARD produces materialized view
  \{ \{ 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

(part of) the Google Map model
```ruby
<% unit google.maps.Maps %>
{
  markers: [ {
    position: {
      latitude : number,
      longitude: number
    }
  } ]
}
<% end unit %>
```

Outline
- What is SQL++?
- How we use SQL++ in UCSD’s FORWARD?
  - SQL++ uses in integration and (live) analytics applications
- Introduction to the formal survey of NoSQL, NewSQL and SQL-on-Hadoop
Surveyed Databases

SQL - on Hadoop

NoSQL

mongoDB

Jaql

CQL

N1QL

AQL

MongoDB driver

SQL - on Hadoop

NoSQL

mongoDB

Jaql

CQL (Cassandra)

Jaql (IBM)

Pig

Hive

• SQL-like syntax
• Hadoop clusters

Jaql

• Algebraic syntax
• Hadoop cluster(s)

CQL (Cassandra)

• Created at Facebook
• SQL-like syntax

MongoDB

• JSON-like syntax
• Most used NoSQL database

JSONiq

• Based on XQuery
• 28msec

N1QL (Couchbase)

• SQL-like syntax
• No production release

SQL

• SQL standard
• Also comprises NewSQL

AQL (AsterixDB)

• FLWOR syntax
• Research database (UCI)

BigQuery (Google)

• Previously ”Dremel”
• SQL-like syntax

MongoJDBC

• SQL-like syntax
• MongoDB via JDBC driver

SQL++ Removes Superficial Differences

• SQL++ covers 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
Outline

- What is SQL++?
- How do we use SQL++ in UCSD’s FORWARD?
- Introduction to the formal survey of NoSQL, NewSQL, and SQL-on-Hadoop
  - Methodology
    - Example 1: data model (data values)
    - Example 2: query language (SELECT clause)
    - Example 3: semantics (path)
    - Example 4: semantics (equality function)

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: 0.035,
  co: 0.0050
}
```
2. SQL++ BNF for values:

```

Example: Data values
```

3. Feature matrix:

```

Example: Data values
```

4. Discussion of the results:

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

```

Example: Data values
```
Example: SELECT clause

1. SQL++ example:
   • Projecting nested collections:
     
     ```sql
     SELECT 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>SQL+</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Example: SELECT clause

4. Discussion of the results:

   • Not well supported features
   • 3 languages support them entirely (same cluster as for data values)
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:
Example: Equality Function

• Config parameters for equality:

```json
@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:

<table>
<thead>
<tr>
<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</td>
<td>None</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Jaql</td>
<td>None</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Pig</td>
<td>None</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CQL</td>
<td>None</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>JSONiq</td>
<td>None</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MongoDB</td>
<td>None</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>N1QL</td>
<td>None</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>SQL</td>
<td>None</td>
<td>NA</td>
<td>NA</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</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 secondary analytics application out of semistructured, declarative platforms