aws re: Invent

OPN207

PartiQL: One query language for all of your data

Almann Goo

Principal Engineer
Amazon.com

Yannis Papakonstantinou

Senior Principal Scientist

Amazon Web Services





Agenda

What is PartiQL?

A walkthrough of PartiQL

Using and contributing to PartiQL in open source

Related breakouts

OPN308-R, -R1 – PartiQL: Solution integration and joining the community

Wednesday 4:00-5:00, Thursday 1:45-2:45

OPN405-R, -R1 – How to integrate PartiQL into your project

Thursday 11:30-12:30, Friday 10:00-11:30

What is PartiQL?





Diverse data sources...

- Data lakes
- Relational databases
- Document databases
- Files on filesystem

Many other query languages...

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

```
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
```

DocumentDB

Unified query language and model

- Format independence
- Storage independence

Format/storage independence



The same query should work on different data sources/format (modulo names)

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

```
{ sid: 2, temp: 70.1 }
{ sid: 2, temp: 49.2 }
{ sid: 1, temp: null }
JSON/Ion S3 Object
```

SQL compatibility

"I don't know what the query language of the future will be, but I know it will be called SQL."

Chris Suver

Distinguished Engineer Amazon.com

SQL backwards compatibility

sid	temp
2	70.1
2	49.2
1	null
A SQL	Table

A SQL query should have the same semantics whether it applied on a SQL table or its semi-structured counterpart

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

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

Anything doable with broadly-supported SQL should have the same meaning on JSON, Ion, Parquet, document stores, etc., when they represent collections of tuples containing scalars

Nested and semi-structured data

- First-class nested data
- Optional schema and query stability

Powerful and complete

Minimal and composable extensions

Where is it?

- Amazon Redshift
- Amazon Simple Storage Service (Amazon S3) Select
- Amazon Simple Storage Service Glacier Select
- Amazon Quantum Ledger Database (Amazon QLDB)
- Amazon.com internal systems
- More announcements forthcoming

A walkthrough of PartiQL's data model





PartiQL data model = ...

- = Ion
- = JSON + strong types (e.g., timestamps, decimals, binary data)

```
location: "Alpine",
readings: [
    time: 2014-03-12T20:00:00Z,
    ozone: 0.035,
    no2: 0.0050
   time: 2014-03-12T22:00:00Z,
    ozone: "m",
    co: 0.4
```

PartiQL data model = ...

- = Ion + SQL bags
 - Bags (tables)—unordered collections
- = JSON + strong types (e.g., timestamps, decimals, binary data) + bags

```
<< ...
    location: "Alpine",
    readings: <<</pre>
        time: 2014-03-12T20:00:00Z,
        ozone: 0.035,
        no2: 0.0050
        time: 2014-03-12T22:00:00Z,
        ozone: "m",
        co: 0.4
      } >>
```

PartiQL data model = SQL data types + ...

- + Nesting
- + Heterogeneity + sparseness

```
{
  vals: [
    [5, 10, true],
    [21, 2, "Abc", 6],
    [4, {lo: 3, exp: 4, hi: 7}, 2, 13, 6]
]
}
```

...arbitrary compositions of data types are allowed!

+ Dynamically typed (schema optional)

A walkthrough of the PartiQL query language



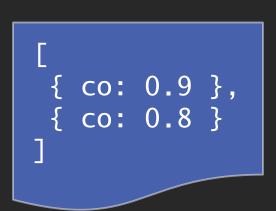


Flattening nested data

sensors

```
{readings:
 [\{v:1.3\}, \{v:2\}]
{readings:
 [\{v:0.7\}, \{v:0.7\}, \{v:0.9\}]
{readings:
 [\{v:0.3\}, \{v:0.8\}, \{v:1.1\}]
{readings:
 [{v:0.7}, {v:1.4}]
```

Find the highest two sensor reading values that are below 1.0, output as tuples with attribute "co"



FROM semantics

sensors

```
{readings:
 [\{v:1.3\}, \{v:2\}]
{readings:
 [\{v:0.7\}, \{v:0.7\}, \{v:0.9\}]
{readings:
 [\{v:0.3\}, \{v:0.8\}, \{v:1.1\}]
{readings:
 [\{v:0.7\}, \{v:1.4\}]
```



FROM sensors AS s, s.readings AS r



```
B^{out}_{FROM} = B^{in}_{WHERE} = <<
\langle s : \{readings:[\{v:1.3\}, ...]\}, r : \{v:1.3\} \rangle,
\langle s : \{readings:[\{v:1.3\}, ...]\}, r : \{v:2\} \rangle,
\langle s : \{readings:[\{v:0.7\}, ...]\}, r : \{v:0.7\} \rangle,
\langle s : \{readings:[\{v:0.7\}, ...]\}, r : \{v:0.7\} \rangle,
\langle s : \{readings:[\{v:0.7\}, ...]\}, r : \{v:0.9\} \rangle,
...
>>
```



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

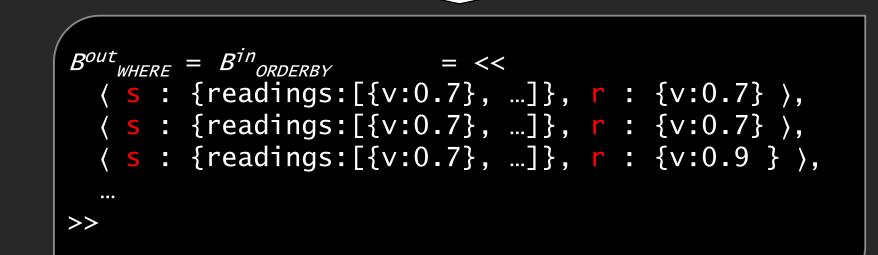
WHERE semantics (the usual)

sensors

```
{readings:
 [\{v:1.3\}, \{v:2\}]
{readings:
 [\{v:0.7\}, \{v:0.7\}, \{v:0.9\}]
{readings:
 [\{v:0.3\}, \{v:0.8\}, \{v:1.1\}]
{readings:
 [\{v:0.7\}, \{v:1.4\}]
```



```
FROM sensors AS s, s.readings AS r WHERE r.v < 1.0
```





```
ORDER BY r.v DESC
LIMIT 2
SELECT r.v AS co
```

ORDER BY semantics

sensors

```
{readings:
 [{v:1.3}, {v:2}]
{readings:
 [\{v:0.7\}, \{v:0.7\}, \{v:0.9\}]
{readings:
[\{v:0.3\}, \{v:0.8\}, \{v:1.1\}]
{readings:
 [{v:0.7}, {v:1.4}]
```



```
B^{out}_{ORDERBY} = B^{in}_{LIMIT} = [
\langle s : \{readings:[\{v:0.7\}, ...]\}, r : \{v:0.9\} \rangle,
\langle s : \{readings:[\{v:0.3\}, ...]\}, r : \{v:0.8\} \rangle,
\langle s : \{readings:[\{v:0.7\}, ...]\}, r : \{v:0.7\} \rangle,
...
```

LIMIT 2
SELECT r.v AS co

Outer flattening nested data

sensors

```
{sensor: 1,
readings:
[{v:1.3}, {v:2}]
{sensor: 2
readings: []
```

Flatten all readings, including sensors without readings

```
SELECT s.sensor, r.v AS co
FROM sensors AS s
LEFT CROSS JOIN
s.readings AS r
```



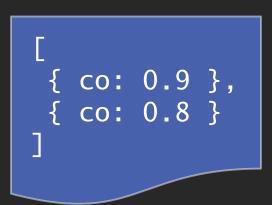
```
<<
    {sensor: 1, co: 1.3},
    {sensor: 1, co: 2.0},
    {sensor: 2, co: null}
>>
```

Tuples in tuples (object/structs)

sensors

```
{readings:
  [{event: {v: 1.3, time: ...},
   {event: {v: 2.0, time: ...}
{readings:
  [{event: {v: 0.7, time: ...}]
```

Find the highest two sensor reading values that are below 1.0, output as tuples with attribute "co"



Composing with SQL features (e.g., subqueries)

sensors

```
{sensor: 1,
readings:
[{v:1.3}, {v:2}]
{sensor: 2,
readings:
[\{v:0.7\}, \{v:0.7\}, \{v:0.9\}]
```

Find all tuples that have an average greater than 1.0

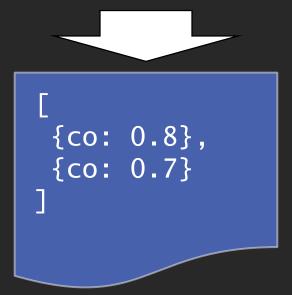
```
SELECT s.sensor, r.v AS co
FROM
       sensors AS s
WHERE
  (SELECT AVG(r.v) FROM s.readings AS r) > 1.5
  <<
   {sensor: 1, co: [{v:1.3}, {v:2}]},
  >>
```

Ranging over anything

readings

Range over an array of numbers (not tuples) and find the highest two sensor readings that are below 1.0

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

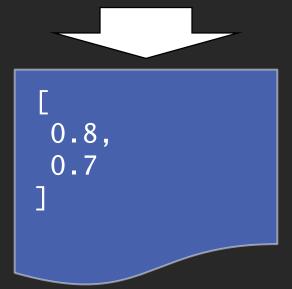


Projecting non-tuples

readings

Range over an array of numbers (not tuples) and find the highest two sensor readings that are below 1.0

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



Flexibility in error cases

- Dynamic typing means things can go wrong at runtime
 - Common in cases like a data lake
- Consider:
 - FROM coll AS v
 - coll may not be a bag or array
 - SELECT x.foo AS bar, y[25] AS baz
 - x might not be a tuple/struct/object or have a foo attribute
 - y might not be an array or have an element at position 25
- PartiQL supports both a strict mode and a permissive mode

Additional features

```
Pivoting and unpivoting over
attribute/value pairs of tuples (a.k.a. objects a.k.a. structs)
key/value pairs of maps
Constructing new, nested PartiQL structures
via SELECT VALUE subqueries
```

Role of schema & working with schema-less

via aggregating into complex values

PartiQL as a unifying query language for diverse services





A unifying query language

- Adopted by multiple services within AWS
- Data model and query language for integrating queries and views
 - Amazon Redshift: database + Amazon S3 data

Combining relational tables and JSON/Ion/etc.

Amazon Redshift tables

area_temp

sensor_loc

area	toocold
1	30
2	50
3	null

sid	area
1	1
2	2

S3 object

readings

{sid: 2,

temperatures: [70.1, 49.2]}

{sid: 1,

temperatures: [71.0]}

Find the sensors (sids) that recorded a temperature that is too cold for their area



{sid: 2}

or

sid 2

Open-source PartiQL





Contributing

- Open-source charter—driven by our tenets
- GitHub organization
 - https://github.com/partiql/
- Forums
 - https://community.partiql.org/

Specification

- https://github.com/partiql/partiql-spec
- GitHub issues or forum submissions for features/clarifications
- Pull requests on the LaTeX for fixes/additions

Reference implementation

- Implemented in Kotlin for JVM
- Read-Eval-Print-Loop for experimenting with PartiQL
- Embeddable and customizable for adding query support to your application
- https://github.com/partiql/partiql-lang-kotlin

Looking toward the future

- Alternative implementations
- Analytic engine integration
- Database integration
- Data format integration
- Specification work (e.g., data manipulation, data definition)

Questions?

- Query language
 - Pivoting and unpivoting
 - Constructing new, nested PartiQL structures
 - Role of schema and schema-less
- Mapping into existing storage systems and databases
- Utilizing open-source implementation

What happened to schema?





Optional schema

- Unstructured data (schema-less)
- Structured data (complete and precise schema)
- Semi-structured data (partial or open schema)

PartiQL with JSON without schema

myobj

```
"location": "Alpine",
"readings": [
    "time": "2014-03-12T20:00:00",
    "ozone": 0.035,
    "no2": 0.0050
 },
    "time": "2014-03-12T22:00:00",
    "ozone": "m",
    "co": 0.4
```

Find the readings since 2012

```
SELECT r.*
FROM myobj.readings AS r
WHERE CAST(r.time AS TIMESTAMP) > `2012-01-01`
```

PartiQL with Ion without schema

myobj

```
location: "Alpine",
readings: [
    time: 2014-03-12T20:00:00,
    ozone: 0.035,
    no2: 0.0050
    time: 2014-03-12T22:00:00,
    ozone: "m",
    co: 0.4
```

Find the readings since 2012

```
SELECT r.*
FROM myobj.readings AS r
WHERE r.time > `2012-01-01`
```

PartiQL with JSON with closed schema

myobj

```
"location": "Alpine",
"readings": [
    "time": "2014-03-12T20:00:00",
    "ozone": 0.035,
    "time": "2014-03-12T22:00:00",
    "ozone": "m",
    "co": 0.4
```

Find the readings since 2012

PartiQL with JSON with open schema

myobj

```
"location": "Alpine",
"readings": [
    "time": "2014-03-12T20:00:00",
    "ozone": 0.035,
   "no2": 0.0050
    "time": "2014-03-12T22:00:00",
    "ozone": "m",
    "co": 0.4
```

Find the readings since 2012

```
SELECT r.*
FROM myobj.readings AS r
WHERE r.time > `2012-01-01`
```

schema

Constructing new nested data





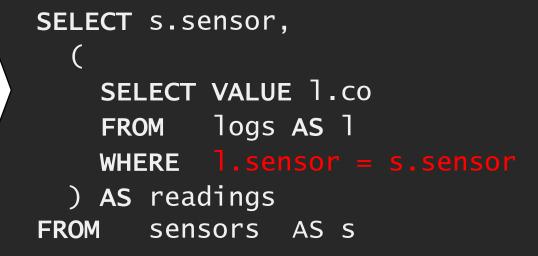
Constructing new nested data

logs

```
[
    {sensor: 1, co: 0.4},
    {sensor: 1, co: 0.2},
    {sensor: 2, co: 0.3},
...
]
```

sensors

```
[
    {sensor: 1},
    {sensor: 2}
]
```





```
<<
    {sensor: 1, readings: <<0.4, 0.2>>},
    {sensor: 2, readings: <<0.3>>},
    ...
>>
```

Pivoting and unpivoting over maps and tuples (objects)





Unpivoting a tuple as a collection

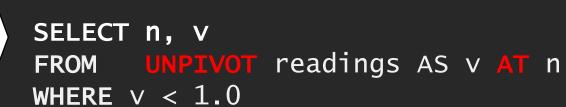
Treating tuples as tables—dealing with data in non-normal form

Unpivoting a tuple as a collection

readings

```
co: 1.3,
no2: 0.7,
co2: 0.3,
o2: 0.6
```

Return a collection of tuples from attributes in the source tuple that are less than 1.0



```
</
{n: "no2", v: 0.7},
{n: "co2", v: 0.3},
{n: "o2", v: 0.6},
>>
```

Pivoting a collection into a tuple

- Creating non-normal data for easier user interaction/visualization
- CSV/TSV exports

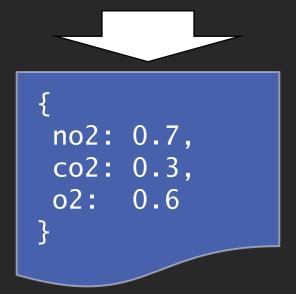
Pivoting a collection into a tuple

readings

```
[
    {n: "co", v: 1.3},
    {n: "no2", v: 0.7},
    {n: "co2", v: 0.3},
    {n: "o2", v: 0.6}
]
```

Find sensor readings that are below 1.0 and create a single tuple with those readings where the name field becomes the attribute





Thank you!







Please complete the session survey in the mobile app.



