BOSS
BIG DATA OPEN SOURCE SYSTEMS

VLDB2020 TOKYO AUG.31 SEP.04
46TH INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES

JSON Analytics with Apache AsterixDB

Glenn Galvizo
Ian Maxon
Gift Sinthong

Michael Carey
Dmitry Lychagin
Till Westmann

UCIRVINE

Couchbase

THE APACHE® SOFTWARE FOUNDATION

What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
  - Basic aggregation and grouping (vs. SQL)
- Analytical features of SQL++ (w/hands-on exercises)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

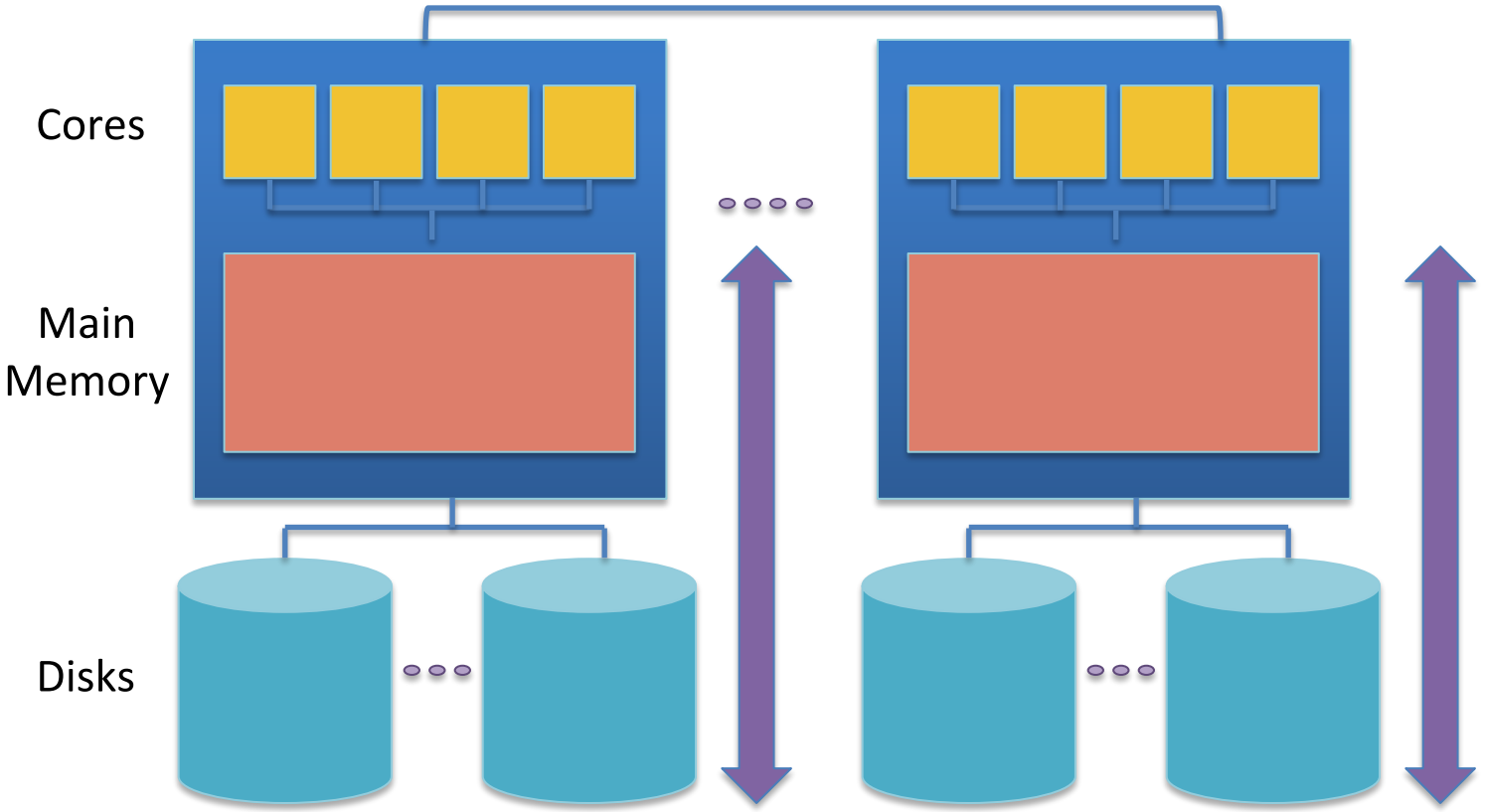
AsterixDB: "One Size Fits a Bunch!"

Wish-list:

- Able to **manage** data
- **Flexible** data model
- Full **query** capability
- Continuous data **ingestion**
- Efficient and robust **parallel** runtime
- Cost **proportional** to task at hand
- Support today's "**Big Data** data types"

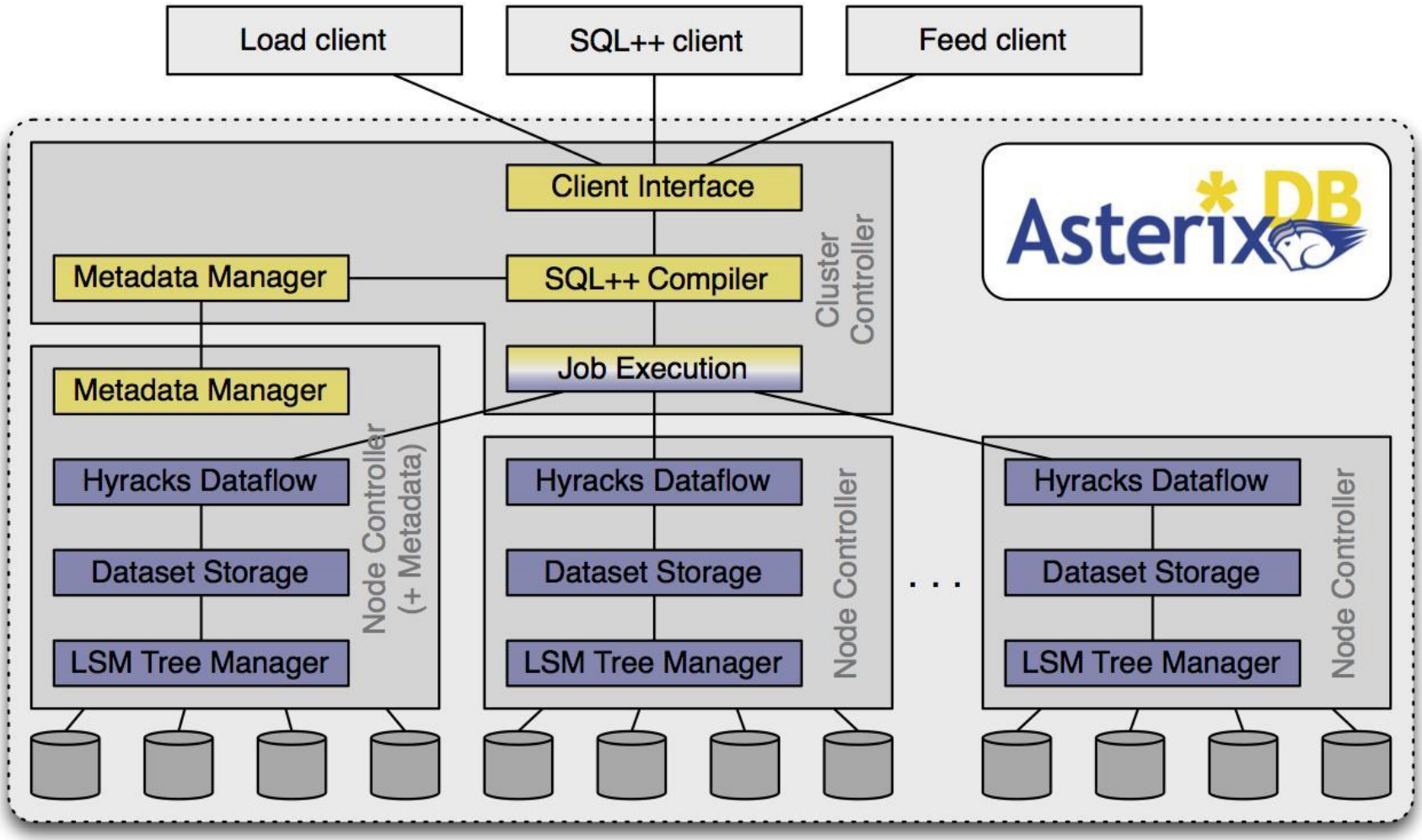
→ ***Parallel NoSQL DBMS*** ←

Just How Big is "Big Data"?

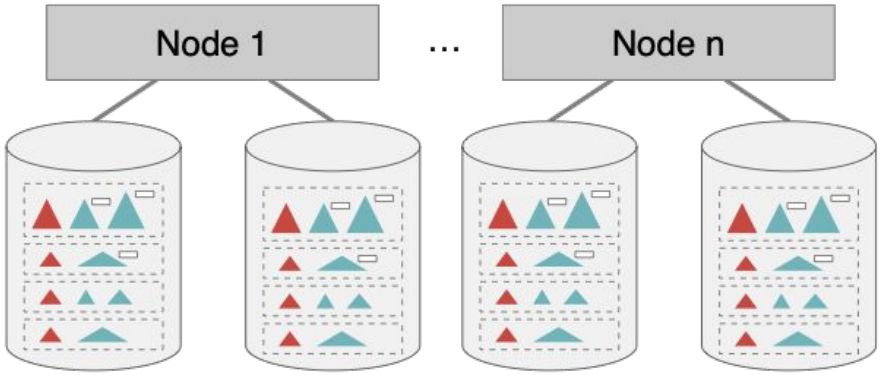


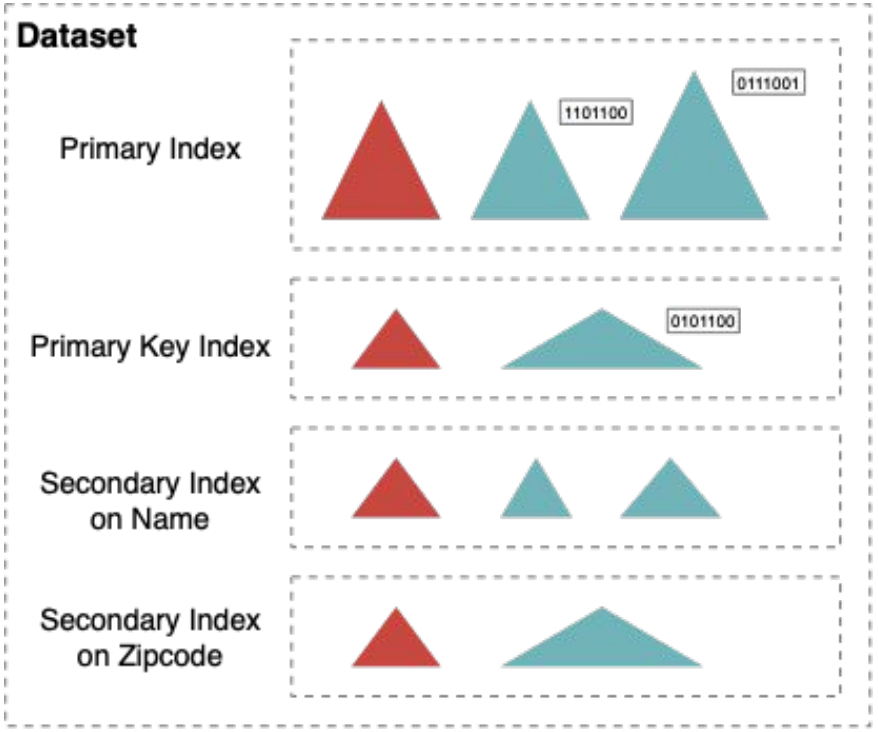
This is Big Data!

AsterixDB System Overview



LSM-Based Storage Management





- Each storage partition holds a logical hash partition of each dataset
- ADM objects (documents) themselves live in the primary index
- Indexes are LSM-based B+ trees, R-trees, or text indexes
- All indexes are **local** indexes

AsterixDB DDL

The home for our application (like a database)

```
CREATE DATAVERSE ShopALot;
USE ShopALot;

CREATE TYPE UsersType AS {
    user_id: string,
    email: string?,
    name: {
        first: string,
        last: string
    },
    phones: [{
        kind: string,
        number: string
    }]?
};
```

Shopper data description (largely optional)

Collection of shopper information

```
CREATE DATASET Users(UsersType)
   PRIMARY KEY user_id;

INSERT INTO Users (
{"user_id": "user007",
 "email": "jamesbond@gmail.com",
 "name": {"first": "James",
          "last": "Bond"},
 "phones": [{"kind": "MOBILE",
             "number": "007-123-4567"}]
});
```

A valid shopper object instance

AsterixDB DDL Alternatives


```
CREATE TYPE UsersType AS {
    user_id: string
};

CREATE DATASET Users(UsersType)
   PRIMARY KEY user_id;

INSERT INTO Users (
{"user_id": "user007",
 "email": "jamesbond@gmail.com",
 "name": {"first": "James",
          "last": "Bond"},
 "phones": [{"kind": "MOBILE",
             "number": "007-123-4567"}]
});
```

```
CREATE TYPE UsersType AS {
    user_id: UUID
};

CREATE DATASET Users(UsersType)
   PRIMARY KEY user_id AUTOGENERATED;

INSERT INTO Users (
{
 "email": "jamesbond@gmail.com",
 "name": {"first": "James",
          "last": "Bond"},
 "phones": [{"kind": "MOBILE",
             "number": "007-123-4567"}]
});
```

The system will add the user_id

AsterixDB DDL (ShopALot)

```
CREATE TYPE StoresType AS {
    store_id: string,
    name: string,
    address: {
        city: string,
        street: string,
        state: string,
        zip_code: integer
    },
    phone: string,
    categories: [string]
};

CREATE DATASET Stores(StoresType)
   PRIMARY KEY store_id;
```

```
CREATE TYPE ProductsType AS {
    product_id: string,
    category: string,
    name: string,
    description: string
 -- list_price: float?
};

CREATE DATASET Products(ProductsType)
   PRIMARY KEY product_id;
```

AsterixDB DDL (ShopALot)

```
CREATE TYPE OrdersType AS {
    order_id: string,
    user_id: string,
    store_id: string,
    total_price: float,
    time_placed: datetime,
    pickup_time: datetime?,
    time_fulfilled: datetime?,
    items: [{
        item_id: string,
        qty: integer,
        selling_price: float,
        product_id: string
    }]
};

CREATE DATASET Orders(OrdersType)
   PRIMARY KEY order_id;
```

```
CREATE TYPE StockedByType AS {
    product_id: string,
    store_id: string,
    qty: integer
};

CREATE DATASET StockedBy(StockedByType)
   PRIMARY KEY product_id, store_id;
```

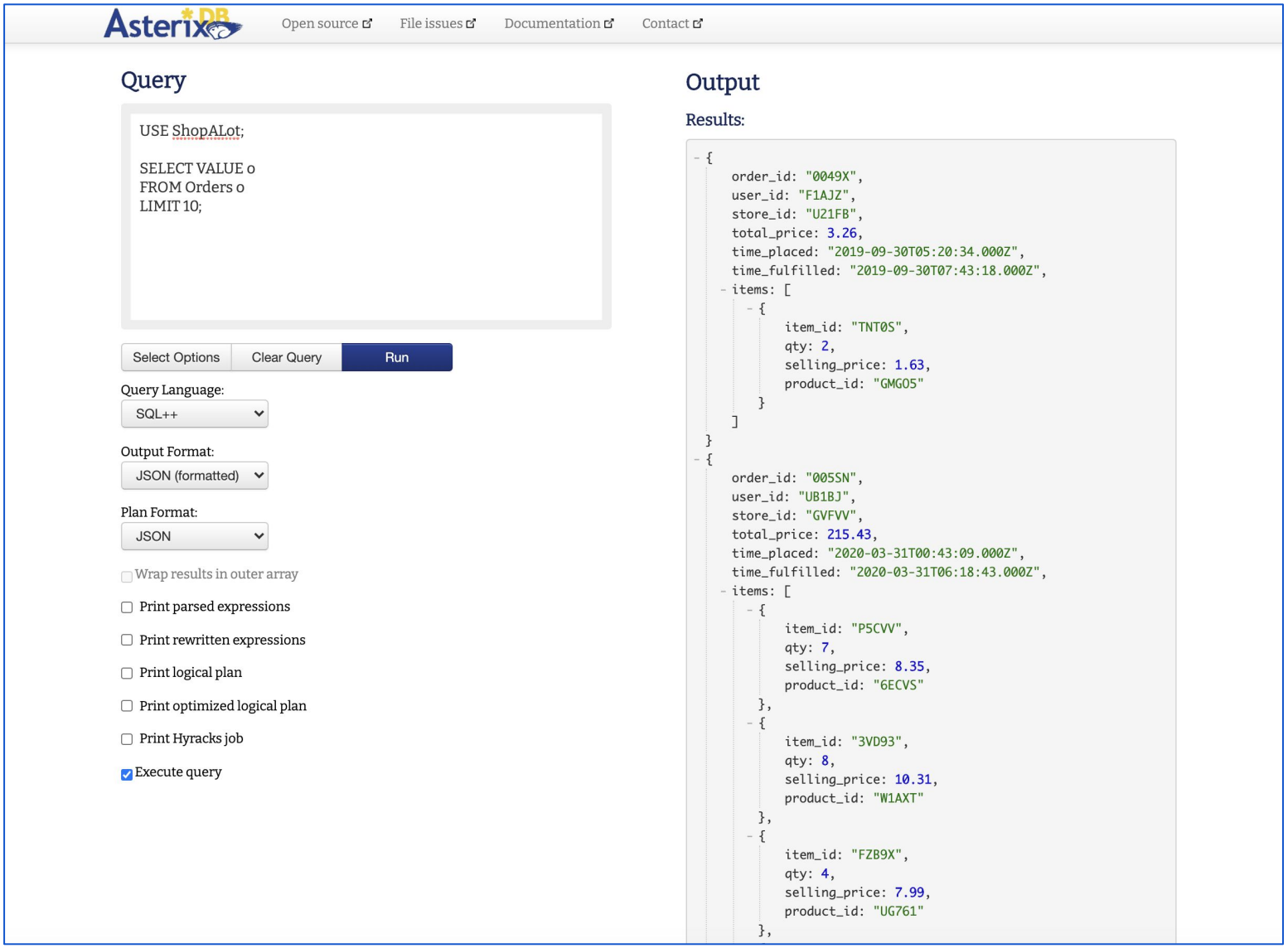
Example Data (ShopALot)


```
{
    order_id: "00DT0",
    user_id: "KJD6S",
    store_id: "P4TYX",
    total_price: 68.84,
    time_placed: "2020-05-22T16:16:13.000Z",
    time_fulfilled: "2020-05-22T19:53:37.000Z",
  - items: [
      - {
            item_id: "37X45",
            qty: 8,
            selling_price: 7.37,
            product_id: "P4XL5"
        },
      - {
            item_id: "SAB4K",
            qty: 2,
            selling_price: 4.94,
            product_id: "ZQLZO"
        }
    ]
}
```



```
- {
    order_id: "ACCVI",
    user_id: "YN31W",
    store_id: "QTB4W",
    total_price: 37.5,
    time_placed: "2020-08-31T12:35:07.000Z",
    pickup_time: "2020-08-31T15:47:48.000Z",
  - items: [
      - {
            item_id: "03SOB",
            qty: 5,
            selling_price: 7.5,
            product_id: "SSYOY"
        }
    ]
}
```

Let’s Give It a Try … !



What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
  - Basic aggregation and grouping (vs. SQL)
- Analytical features of SQL++ (w/hands-on exercises)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

Connecting to AsterixDB

- Visit https://tujun.ga/roundrobin.php
- You will be redirected to an AsterixDB instance with all data for the tutorial preloaded and indexes created
- Feel free to give the instance a query, like:

```
USE ShopALot;

SELECT VALUE o
FROM Orders o
LIMIT 10;
```

What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
  - Basic aggregation and grouping (vs. SQL)
- Analytical features of SQL++ (w/hands-on exercises)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
  - Basic aggregation and grouping (vs. SQL)
- Analytical features of SQL++ (w/hands-on exercises)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

Just Like SQL...

```
SELECT user_id, email
FROM Users
WHERE email LIKE "%gmail.com"
LIMIT 3;
```

```
[
  {
    user_id: "001PR",
    email: "gonzalezjennifer42787@gmail.com"
  },
  {
    user_id: "007GA",
    email: "cou704@gmail.com"
  },
  {
    user_id: "007GQ",
    email: "kri59334@gmail.com"
  }
]
```

Just Like SQL...

```
SELECT user_id, email
FROM Users
WHERE email LIKE "%gmail.com"
LIMIT 3;
```

```
SELECT u.email, o.time_placed
FROM Users u, Orders o
WHERE u.user_id = o.user_id
  AND o.total_price > 200
ORDER BY o.total_price DESC
LIMIT 3;
```

```
[
  {
    "email": "thomas89979@hotmail.com",
    "time_placed": "2020-06-19T11:23:56.000Z"
  },
  {
    "email": "kirk.ter478@gmail.com",
    "time_placed": "2020-07-01T04:08:55.000Z"
  },
  {
    "email": "gonzalez855@yahoo.com",
    "time_placed": "2020-02-15T03:48:09.000Z"
  }
]
```

Just Like SQL...

```
SELECT user_id, email
FROM Users
WHERE email LIKE "%gmail.com"
LIMIT 3;
```

```
SELECT u.email, o.time_placed
FROM Users u, Orders o
WHERE u.user_id = o.user_id
  AND o.total_price > 200
ORDER BY o.total_price DESC
LIMIT 3;
```

```
SELECT u.email, o.time_placed
FROM Users u JOIN Orders o
  ON u.user_id = o.user_id
WHERE o.total_price > 200
ORDER BY o.total_price DESC
LIMIT 3;
```

Just Like SQL…

```
SELECT user_id, email
FROM Users
WHERE email LIKE "%gmail.com"
LIMIT 3;
```

```
SELECT u.email, o.time_placed
FROM Users u, Orders o
WHERE u.user_id = o.user_id
  AND o.total_price > 200
ORDER BY o.total_price DESC
LIMIT 3;
```

```
SELECT store_id, count(*) AS cnt
FROM Orders
GROUP BY store_id
HAVING count(*) > 0
ORDER BY cnt DESC
LIMIT 3;
```

```
[
  { "store_id": "1RMXY",
    "cnt": 121
  },
  { "store_id": "2TM62",
    "cnt": 120
  },
  { "store_id": "70GOX",
    "cnt": 112
  }
]
```

… Almost!

```
SELECT email, time_placed
FROM Users, Orders
WHERE Users.user_id = Orders.user_id
  AND total_price > 200
ORDER BY total_price DESC
LIMIT 3;
```

```
ASX1074: Cannot resolve ambiguous alias
reference for identifier total_price
(in line 6, at column 7)
[CompilationException]
```

… Almost!

```
SELECT email, time_placed
FROM Users, Orders
WHERE Users.user_id = Orders.user_id
  AND total_price > 200
ORDER BY total_price DESC
LIMIT 3;

SELECT u.email, o.time_placed
FROM Users u, Orders o
WHERE u.user_id = o.user_id
  AND o.total_price > 200
ORDER BY o.total_price DESC
LIMIT 3;
```


```
[
  {
    "email": "thomas89979@hotmail.com",
    "time_placed": "2020-06-19T11:23:56.000Z"
  },
  {
    "email": "kirk.ter478@gmail.com",
    "time_placed": "2020-07-01T04:08:55.000Z"
  },
  {
    "email": "gonzalez855@yahoo.com",
    "time_placed": "2020-02-15T03:48:09.000Z"
  }
]
```

… Almost!

```
SELECT u.email, o.time_placed
FROM Users, Orders
WHERE Users.user_id = Orders.user_id
  AND total_price > 200
ORDER BY total_price DESC
LIMIT 3;

SELECT u.email, o.time_placed
FROM Users u, Orders o
WHERE u.user_id = o.user_id
  AND o.total_price > 200
ORDER BY o.total_price DESC
LIMIT 3;

SELECT *
FROM Users u, Orders o
WHERE u.user_id = o.user_id
  AND o.total_price > 200
ORDER BY o.total_price DESC
LIMIT 3;
```


```
[
  {
    "u": {
       "user_id": "XCPVZ",
       "email": "thomas89979@hotmail.com",
       "name": { "first": "Christine",
                 "last": "Thomas" },
       "phone": [
          { "type": "MOBILE",
            "number": "001-931-747-6904x197" }
          ]
        },
    "o": {
       "order_id": "G6BT1",
       "user_id": "XCPVZ",
       "store_id": "XGK64",
       "total_price": 716.8,
       "time_placed": "2020-06-19T11:23:56.000Z",
       "time_fulfilled": "2020-06-19T17:22:35.000Z",
        "items": [
             { item_id: "CWSP9",
               "qty": 10,
               "selling_price": 71.68,
               product_id: "X0401" }
           ]
         }
       },
     ...
```

Added VALUE

```
SELECT VALUE product_id
FROM StockedBy
WHERE store_id = "C4N2L";
```

```
[
  "T1P2J",
  "TJHLQ",
  "MUFUS"
]
```

Added VALUE

```
SELECT VALUE product_id
FROM StockedBy
WHERE store_id = "C4N2L";

SELECT VALUE {
  "StoreName": s.name,
  "Quantity": sb.qty
}
FROM StockedBy sb, Stores s
WHERE sb.store_id = s.store_id
  AND sb.store_id = "C4N2L";
```

```
[
  {
    "StoreName": "Sheetz",
    "Quantity": 46
  },
  {
    "StoreName": "Sheetz",
    "Quantity": 38
  },
  {
    "StoreName": "Sheetz",
    "Quantity": 34
  }
]
```

Added VALUE

```
SELECT VALUE product_id
FROM StockedBy
WHERE store_id = "C4N2L";
```

```
SELECT VALUE {
  "StoreName": s.name,
  "Quantity": sb.qty
}
FROM StockedBy sb, Stores s
WHERE sb.store_id = s.store_id
  AND sb.store_id = "C4N2L";
```

```
SELECT s.name AS StoreName,
       sb.qty AS Quantity
FROM StockedBy sb, Stores s
WHERE sb.store_id = s.store_id
  AND sb.store_id = "C4N2L";
```

Added VALUE

```
SELECT VALUE product_id
FROM StockedBy
WHERE store_id = "C4N2L";

SELECT VALUE {
  "StoreName": s.name,
  "Quantity": sb.qty
}
FROM StockedBy sb, Stores s
WHERE sb.store_id = s.store_id
  AND s.store_id = "C4N2L";

SELECT VALUE {
  "StoreName": s.name,
  "Stocks": (SELECT VALUE sb.product_id
             FROM StockedBy sb
             WHERE sb.store_id = s.store_id)
}
FROM Stores s
WHERE s.store_id = "C4N2L";
```

```
[
  {
    "StoreName": "Sheetz",
    "Stocks": [
      "MUFUS",
      "T1P2J",
      "TJHLQ"
    ]
  }
]
```

Quiz Time!

A
```
SELECT *
FROM Orders
WHERE total_price =
   (SELECT MAX(total_price) FROM Orders);
```

B
```
SELECT o1.*
FROM Orders o1
WHERE o1.total_price =
   (SELECT MAX(o2.total_price) FROM Orders o2);
```

C
```
SELECT o1.*
FROM Orders o1
WHERE o1.total_price =
   (SELECT VALUE MAX(o2.total_price) FROM Orders);
```

D
```
SELECT o1.*
FROM Orders o1
WHERE o1.total_price =
   (SELECT VALUE MAX(o2.total_price) FROM Orders o2)[0];
```

Q: Which query retrieves the orders that have the highest total price?

SQL Pitfalls and the Value of VALUE

SQL++ "best guesses" that Orders is a field of Orders

```
   SELECT *
A  FROM Orders
   WHERE total_price =
      (SELECT MAX(total_price) FROM Orders);
```

```
Type mismatch: expected
value of type multiset
or array, but got the
value of type object (in
line 6, at column 34)
[TypeMismatchException]
```

SQL Pitfalls and the Value of VALUE

```
   SELECT *                                                    []
A  FROM Orders
   WHERE total_price =
      (SELECT MAX(total_price) FROM Orders);

   SELECT o1.*
B  FROM Orders o1
   WHERE o1.total_price =
      (SELECT MAX(o2.total_price) FROM Orders o2);
```

Standard SQL would apply "flat world" row/column coercion magic

SQL Pitfalls and the Value of VALUE

```
   SELECT *                                                    []
A  FROM Orders
   WHERE total_price =
      (SELECT MAX(total_price) FROM Orders);

   SELECT o1.*
B  FROM Orders o1
   WHERE o1.total_price =
      (SELECT MAX(o2.total_price) FROM Orders o2);

   SELECT o1.*
C  FROM Orders o1
   WHERE o1.total_price =
      (SELECT VALUE MAX(o2.total_price) FROM Orders);
```

SQL++ SELECT statements always return collections (not scalars)

SQL Pitfalls and the Value of VALUE

```
   SELECT *
A  FROM Orders
   WHERE total_price =
      (SELECT MAX(total_price) FROM Orders);

   SELECT o1.*
B  FROM Orders o1
   WHERE o1.total_price =
      (SELECT MAX(o2.total_price) FROM Orders o2);

   SELECT o1.*
C  FROM Orders o1
   WHERE o1.total_price =
      (SELECT VALUE MAX(o2.total_price) FROM Orders o2);

   SELECT o1.*
D  FROM Orders o1
   WHERE o1.total_price =
      (SELECT VALUE MAX(o2.total_price) FROM Orders o2)[0];
```

```
[
  {
    "order_id": "G6BT1",
    "user_id": "XCPVZ",
    "store_id": "XGK64",
    "total_price": 716.8,
    "time_placed":
       "2020-06-19T11:23:56.000Z",
    "time_fulfilled":
       "2020-06-19T17:22:35.000Z",
    "items": [
      {
        "item_id": "CWSP9",
        "qty": 10,
        "selling_price": 71.68,
        "product_id": "X0401"
      }
    ]
  }
]
```

We know the subquery returns just one value, so we extract it this way

SQL Pitfalls and the Value of VALUE

A
```
SELECT *
FROM Orders
WHERE total_price =
   (SELECT MAX(total_price) FROM Orders);
```

B
```
SELECT o1.*
FROM Orders o1
WHERE o1.total_price =
   (SELECT MAX(o2.total_price) FROM Orders o2);
```

C
```
SELECT o1.*
FROM Orders o1
WHERE o1.total_price =
   (SELECT VALUE MAX(o2.total_price) FROM Orders);
```

D
```
SELECT o1.*
FROM Orders o1
WHERE o1.total_price =
   (SELECT VALUE MAX(o2.total_price) FROM Orders o2)[0];
```

Unnesting

```
SELECT o.order_id,
       o.user_id,
       i.product_id AS product,
       i.qty AS quantity
FROM Orders o UNNEST o.items i
WHERE i.qty > 30;
```

```
[
  {
    "order_id": "5IZ2R",
    "user_id": "3PB90",
    "product": "93NRR",
    "quantity": 33
  },
  {
    "order_id": "SW6PI",
    "user_id": "86OOD",
    "product": "KA8Q9",
    "quantity": 37
  }
]
```

Unnesting

```
SELECT o.order_id,
       o.user_id,
       i.product_id AS product,
       i.qty AS quantity
FROM Orders o UNNEST o.items i
WHERE i.qty > 30;
```

```
SELECT o.order_id,
       o.user_id,
       i.product_id AS product,
       i.qty AS quantity
FROM Orders o, o.items i
WHERE i.qty > 30;
```

Quantification

```
SELECT DISTINCT VALUE o.user_id
FROM Orders o
WHERE SOME i IN o.items
      SATISFIES i.selling_price >= 80.00;
```

```
[
  "FOAYZ",
  "OLRCD",
  "GBPXS",
  "RQ6FT"
]
```

Quantification

```
SELECT DISTINCT VALUE o.user_id
FROM Orders o
WHERE SOME i IN o.items
      SATISFIES i.selling_price >= 80.00;
```

```
[
  "KMK3F",
  "OE4HV",
  "XCPVZ"
]
```

```
SELECT DISTINCT VALUE o.user_id
FROM Orders o
WHERE EVERY i IN o.items
      SATISFIES i.selling_price >= 70.00;
```

Quantification

```
SELECT DISTINCT VALUE o.user_id
FROM Orders o
WHERE SOME i IN o.items
      SATISFIES i.selling_price >= 80.00;
```

```
[
  "KMK3F",
  "OE4HV",
  "XCPVZ"
]
```

```
SELECT DISTINCT VALUE o.user_id
FROM Orders o
WHERE EVERY i IN o.items
      SATISFIES i.selling_price >= 70.00;
```

```
SELECT DISTINCT VALUE o.user_id
FROM Orders o
WHERE EVERY i IN o.items
      SATISFIES i.selling_price >= 70.00
  AND ARRAY_COUNT(o.items) > 0;
```

Quantification

```
SELECT DISTINCT VALUE o.user_id
FROM Orders o
WHERE SOME i IN o.items
      SATISFIES i.selling_price >= 80.00;

SELECT DISTINCT VALUE o.user_id
FROM Orders o
WHERE EVERY i IN o.items
      SATISFIES i.selling_price >= 70.00;

SELECT DISTINCT VALUE o.user_id
FROM Orders o
WHERE array_count(o.items) > 0
  AND (EVERY i IN o.items
       SATISFIES i.selling_price >= 70.00);

SELECT u.name
FROM Users u
WHERE u.user_id IN ( ... );
```

```
[
  {
    "name": {
       "first": "Martin",
       "last": "Levy"
    }
  },
  {
    "name": {
       "first": "Kri",
       "last": "Gomez"
    }
  },
  {
    "name": {
       "first": "Christine",
       "last": "Thomas"
    }
  }
]
```

Remember the Data

Have I “Missed” Anything?

```
SELECT o.order_id,
       o.time_placed,
       o.time_fulfilled,
       o.total_price,
       o.user_id
FROM Orders o
WHERE total_price > 150.00
  AND o.time_fulfilled IS MISSING;
```


```
[
  {
    "order_id": "C1W04",
    "time_placed": "2020-08-31T13:28:36.000Z",
    "total_price": 221.28,
    "user_id": "HZ7V1"
  },
  {
    "order_id": "DTW97",
    "time_placed": "2020-08-31T08:00:20.000Z",
    "total_price": 153.41,
    "user_id": "B8WJY"
  },
  {
    "order_id": "SWRD1",
    "time_placed": "2020-08-31T09:14:00.000Z",
    "total_price": 190.7,
    "user_id": "HOGTV"
  }
]
```

Have I "Missed" Anything?

```
SELECT o.order_id,
       o.time_placed,
       o.time_fulfilled,
       o.total_price,
       o.user_id
FROM Orders o
WHERE total_price > 150.00
  AND o.time_fulfilled IS MISSING;
```

```
SELECT VALUE {
  "order_id": o.order_id,
  "time_placed": o.time_placed,
  "time_fulfilled": o.time_fulfilled,
  "total_price": o.total_price,
  "user_id": o.user_id
}
FROM Orders o
WHERE total_price > 150.00
  AND o.time_fulfilled IS MISSING;
```

```
[
  {
    "order_id": "C1W04",
    "time_placed": "2020-08-31T13:28:36.000Z",
    "total_price": 221.28,
    "user_id": "HZ7V1"
  },
  {
    "order_id": "DTW97",
    "time_placed": "2020-08-31T08:00:20.000Z",
    "total_price": 153.41,
    "user_id": "B8WJY"
  },
  {
    "order_id": "SWRD1",
    "time_placed": "2020-08-31T09:14:00.000Z",
    "total_price": 190.7,
    "user_id": "HOGTV"
  }
]
```

A CASE Study

```
SELECT VALUE {
  "order_id": o.order_id,
  "time_placed": o.time_placed,
  "time_fulfilled":
    CASE
      WHEN o.time_fulfilled IS MISSING
       THEN "TBD"
       ELSE o.time_fulfilled
    END,
  "total_price": o.total_price,
  "user_id": o.user_id
}
FROM Orders o
WHERE user_id = "QREX9"
LIMIT 3;
```

```
[
  {
    "order_id": "0PS02",
    "time_placed": "2020-08-31T10:44:47.000Z",
    "total_price": 58.63,
    "user_id": "QREX9",
    "time_fulfilled": "TBD"
  },
  {
    "order_id": "9L6V5",
    "time_placed": "2020-08-16T10:19:14.000Z",
    "total_price": 7.08,
    "user_id": "QREX9",
    "time_fulfilled": "2020-08-16T17:44:41.000Z"
  },
  {
    "order_id": "HE6O5",
    "time_placed": "2018-11-23T15:23:24.000Z",
    "total_price": 130.08,
    "user_id": "QREX9",
    "time_fulfilled": "2018-11-23T20:43:36.000Z"
  }
]
```

Lab 1: Basic SQL++ Queries

1. List the first names of users that have placed orders with a total price greater than $500. *Only return a list of strings, not objects.* [19]
2. List the names and addresses of stores that have a stock of at least 45 products with “Wafer” in the name. [8]
3. List home phone numbers that start with “97” with the associated user's id. [19]
4. Get the names and phone numbers of stores that are in the state “WA” and have a category containing the substring “Personal”. [7]
5. Get the order id and pickup time from orders placed after 2020-08-31 at 7:30AM. If the pickup time is missing from the order, return the order id with the string “NOT SPECIFIED”. *Hint: compare the time placed with datetime(“2020-08-31T07:30:00.000Z”).* [82]

Lab 1: Q1 - Q2 Answers

Q1: List the first names of users that have placed orders with a total price greater than $500. *Only return a list of strings, not a list of objects.*	**Q2:** List the names and addresses of stores that have a stock of at least 45 products with "Wafer" in the name.
SELECT VALUE U.name.first **FROM** ShopALot.Users U, ShopALot.Orders O **WHERE** U.user_id = O.user_id **AND** O.total_price > 500;	**SELECT** S.name, S.address **FROM** ShopALot.Stores S, ShopALot.StockedBy SB**,** ShopALot.Products P **WHERE** SB.store_id = S.store_id **AND** SB.product_id = P.product_id **AND** P.name **LIKE** "%Wafer%" **AND** SB.qty > 45;

Lab 1: Q3 - Q4 Answers

Q3: List home phone numbers that start with “97” with the associated user's id.	**Q4:** Get the names and phone numbers of stores that are in the state “WA” and has a category with the substring “Personal”.
SELECT U.user_id, UP.number **FROM** ShopALot.Users U, U.phones UP **WHERE** UP.number **LIKE** "97%" **AND** UP.kind = "HOME";	**SELECT** S.name, S.phone **FROM** ShopALot.Stores S **WHERE** S.address.state = "WA" **AND** (**SOME** C **IN** S.categories **SATISFIES** C **LIKE** "%Personal%");

Lab 1: Q5 Answer

Q5: Get the order id and pickup time from orders placed after 2020-08-31 at 7:30AM. If the pickup time is missing from the order, return the order id with the string “NOT SPECIFIED”.

```
SELECT O.order_id,
    CASE (O.pickup_time IS MISSING)
    WHEN TRUE THEN "NOT SPECIFIED"
    ELSE O.pickup_time
    END AS pickup_time
FROM ShopALot.Orders O
WHERE O.time_placed >
datetime("2020-08-30T07:30:00.000Z");
```

What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
  - Basic aggregation and grouping (vs. SQL)
- Analytical features of SQL++ (w/hands-on exercises)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

SQL Grouping and Aggregation

```
SELECT s.address.state, COUNT(*) AS cnt
FROM Stores as s, Orders as o
WHERE s.store_id = o.store_id
GROUP BY s.address.state;
```

```
[
  {
    "state": "AK",
    "cnt": 28
  },
  {
    "state": "AL",
    "cnt": 546
  },
  {
    "state": "KY",
    "cnt": 206
  },
  {
    "state": "LA",
    "cnt": 399
  },

  ...
]
```

SQL Grouping and Aggregation

```
SELECT s.address.state, COUNT(*) AS cnt
FROM Stores as s, Orders as o
WHERE s.store_id = o.store_id
GROUP BY s.address.state;
```

s.address.state	s	o
AK	s_{THLUS}	o_{4QR5P}
	s_{THLUS}	o_{4WUE6}
	…	…
AL	s_{0HKZ3}	o_{0QDFV}
	s_{0HKZ3}	o_{0SVOR}
	s_{0HKZ3}	o_{125PT}
	s_{0HKZ3}	o_{2PJ4Y}
	…	…
… + 45 more		

28

546

SQL++ Aggregation (only)

```
SELECT u.email,
       ARRAY_COUNT(o.items) AS order_size
FROM Users AS u, Orders AS o
WHERE u.user_id = o.user_id
ORDER BY order_size DESC
LIMIT 3;
```

```
[
  {
    "email": "claire.evans@gmail.com",
    "order_size": 8
  },
  {
    "email": "and82566@yahoo.com",
    "order_size": 7
  },
  {
    "email": "Thompson1852@hotmail.com",
    "order_size": 7
  }
]
```

SQL++ Aggregation (only)

```
SELECT u.email,
       ARRAY_COUNT(o.items) AS order_size
FROM Users AS u, Orders AS o
WHERE u.user_id = o.user_id
ORDER BY order_size DESC
LIMIT 3;
```

```
[
  59.94
]
```

```
SELECT VALUE MAX(p.list_price)
FROM Products p
WHERE is_number(p.list_price);
```

Note: Field p.list_price has a few “dirty values” ("TBD", "TODO", "expensive", "pricey")

SQL++ Aggregation (only)

```
SELECT u.email,
       ARRAY_COUNT(o.items) AS order_size
FROM Users AS u, Orders AS o
WHERE u.user_id = o.user_id
ORDER BY order_size DESC
LIMIT 3;
```

```
[
  59.94
]
```

```
SELECT VALUE MAX(list_price)
FROM Products
WHERE is_number(list_price);
```

```
ARRAY_MAX(
  (SELECT VALUE list_price
   FROM Products
   WHERE is_number(list_price))
);
```

SQL++ Grouping (only)

```
SELECT s.address.state, g
FROM Stores AS s, Orders AS o
WHERE s.store_id = o.store_id
GROUP BY s.address.state GROUP AS g;
```

```
[
  {
    "state": "AK",
    "g": [
      {
        "s": {
          "store_id": "THLUS",
          "name": "Jackson Food Store",
          "address": {
            "street": "3354 Betty Cliff",
            "city": "Houston",
            "state": "AK",
            "zip_code": "99694"
          },
          "phone": "585.025.4631",
          "categories": [
            "Bread & Bakery",
            ...
            "Condiments, Spice, & Bake"
          ]
        },
        "o": {
          "order_id": "4WUE6",
          "user_id": "EIGF6",
          "store_id": "THLUS",
          "total_price": 25.34,
          "time_placed": "2020-03-22T01:29:03.000Z",
          "pickup_time": "2020-03-22T07:27:31.000Z",
          "time_fulfilled": "2020-03-22T13:26:00.000Z",
          "items": [
            {
              "item_id": "6TYQA",
              "qty": 2,
              "selling_price": 12.67,
              "product_id": "90T50"
            }
          ]
        }
      },
      ...
    ]
  },
  ...
]
```

SQL++ Groups and Querying

```
FROM Stores AS s, Orders AS o
WHERE s.store_id = o.store_id
GROUP BY s.address.state GROUP AS g
SELECT s.address.state,
       (SELECT g.s.store_id, g.s.name, g.o.order_id FROM g) AS so_pairs;
```

*This could be **any** query over the group!*
(Notice that FROM came first, BTW...)

```
[
  {
    "state": "AK",
    "so_pairs": [
      { "store_id": "THLUS", "name": "Jackson Food Store", "order_id": "4WUE6" },
      { "store_id": "THLUS", "name": "Jackson Food Store", "order_id": "61P1A" }
      ...
    ]
  },
  { "state": "AL",
    "so_pairs": [
      { "store_id": "0HKZ3", "name": "Border Station", "order_id": "0QDFV" },
      { "store_id": "0HKZ3", "name": "Border Station", "order_id": "2PJ4Y" }
      ...
    ]
  }
]
~
```

SQL Grouping and Aggregation Explained

```
SELECT s.address.state, COUNT(*) AS cnt
FROM Stores as s, Orders as o
WHERE s.store_id = o.store_id
GROUP BY s.address.state;
```

```
[
  {
    "state": "AK",
    "cnt": 28
  },
  {
    "state": "AL",
    "cnt": 546
  },
  {
    "state": "KY",
    "cnt": 206
  },
  {
    "state": "LA",
    "cnt": 399
  },
  ...
]
```

SQL Grouping and Aggregation Explained

```
SELECT s.address.state, COUNT(*) AS cnt
FROM Stores as s, Orders as o
WHERE s.store_id = o.store_id
GROUP BY s.address.state;
```

```
SELECT s.address.state, ARRAY_COUNT(g) AS cnt
FROM Stores as s, Orders as o
WHERE s.store_id = o.store_id
GROUP BY s.address.state GROUP AS g;
```

```
[
  {
    "state": "AK",
    "cnt": 28
  },
  {
    "state": "AL",
    "cnt": 546
  },
  {
    "state": "KY",
    "cnt": 206
  },
  {
    "state": "LA",
    "cnt": 399
  },
  ...
]
```

Lab 2: SQL++ Grouping and Aggregation Exercises

1. List the names of users that have placed exactly 14 orders.
2. For the two most frequent store categories, list the category itself along with the number of stores containing that category.
3. For stores with total sales less than $400, list the store ID and the orders associated with this store.

Lab 2: Q1 - Q2 Answers

Q1: List the names of users that have placed exactly 14 orders.	**Q2:** For the two most frequent store categories, list the category itself along with the number of stores containing that category.
SELECT U.name **FROM** ShopALot.Users U, ShopALot.Orders O **WHERE** U.user_id = O.user_id **GROUP BY** U.user_id, U.name **HAVING COUNT**(*) = 14;	**SELECT** SC, **COUNT**(*) AS category_count **FROM** ShopALot.Stores S**,** S.categories SC **GROUP BY** SC **ORDER BY COUNT**(*) **DESC** **LIMIT** 2;

Lab 2: Q3 Answer

Q3: For stores with total sales less than $400, list the store ID and the orders associated with this store.

```
SELECT O.store_id, store_orders
FROM ShopALot.Orders O
GROUP BY O.store_id
GROUP AS store_orders
HAVING SUM(O.total_price) < 400;
```

What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
  - Basic aggregation and grouping (vs. SQL)
- Analytical features of SQL++ (w/hands-on exercises)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

Beyond Grouped Aggregation

- Like standard SQL, SQL++ supports a collection of more advanced analytical clauses
  - Various ways to group data for aggregation
    - ROLLUP
    - CUBE
    - GROUPING SETS
  - Functions to aggregate “windows” of (ordered) data
    - ORDER BY
    - PARTITION BY
    - ROWS FOLLOWING/PROCEEDING, etc.
- Let’s have a look...

What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
  - Basic aggregation and grouping (vs. SQL)
- Analytical features of SQL++ (w/hands-on exercises)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

ROLL Call!

```
SELECT s.address.state, s.address.city,
       COUNT(s.store_id) AS stores
FROM Stores s
WHERE s.address.state LIKE "C%"
GROUP BY ROLLUP(s.address.state, s.address.city)
ORDER BY s.address.state, s.address.city;
```

```
[
    { "state": null, "city": null, "stores": 25 }
    { "state": "CA", "city": null, "stores": 23 }
    { "state": "CA", "city": "Acton", "stores": 1 }
    { "state": "CA", "city": "Anaheim", "stores": 1 }
    { "state": "CA", "city": "Arroyo Grande", "stores": 1 }
    { "state": "CA", "city": "Bridgeport", "stores": 1 }
    { "state": "CA", "city": "Cambria", "stores": 1 }
    { "state": "CA", ... }
    ...
    { "state": "CO", "city": null, "stores": 2 }
    { "state": "CO", "city": "Empire", "stores": 1 }
    { "state": "CO", "city": "Ridgway", "stores": 1 }
]
```

GROUP BY ROLLUP
(x,y,z)
=
GROUP BY GROUPING SETS
(x,y,z), (x,y), (x), ()
=
SELECT … GROUP BY x,y,z
UNION ALL
SELECT … GROUP BY x,y
UNION ALL
SELECT … GROUP BY x
UNION ALL
SELECT … GROUP BY ()

Be a CUBEist

```
SELECT s.address.state, year,
       ROUND(SUM(o.total_price)) AS sales
FROM Orders o JOIN Stores s ON o.store_id = s.store_id
LET year = GET_YEAR(DATETIME(o.time_placed))
WHERE s.address.state LIKE "C%"
GROUP BY CUBE(s.address.state, year)
ORDER BY s.address.state, year;
```

```
[
     { "state": null, "year": null, "sales": 69094.0 },
     { "state": null, "year": 2018, "sales": 8038.0 },
     { "state": null, "year": 2019, "sales": 17980.0 },
     { "state": null, "year": 2020, "sales": 43077.0 },
     { "state": "CA", "year": null, "sales": 64312.0 },
     { "state": "CA", "year": 2018, "sales": 7455.0 },
     { "state": "CA", "year": 2019, "sales": 16548.0 },
     { "state": "CA", "year": 2020, "sales": 40309.0 },
     { "state": "CO", "year": null, "sales": 4782.0 },
     { "state": "CO", "year": 2018, "sales": 583.0 },
     { "state": "CO", "year": 2019, "sales": 1431.0 },
     { "state": "CO", "year": 2020, "sales": 2768.0 },
]
```

GROUP BY CUBE (x,y,z)
=
GROUP BY GROUPING SETS
 (x,y,z),
 (x,y), (x,z), (y,z),
 (x), (y), (z),
 ()
=
SELECT … GROUP BY x,y,z
UNION ALL
SELECT … GROUP BY x,y
UNION ALL
SELECT … GROUP BY x,z
UNION ALL
SELECT … GROUP BY y,z
UNION ALL
SELECT … GROUP BY x
UNION ALL
SELECT … GROUP BY y
UNION ALL
SELECT … GROUP BY z
UNION ALL
SELECT … GROUP BY ()

What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
  - Basic aggregation and grouping (vs. SQL)
- Analytical features of SQL++ (w/hands-on exercises)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

Let's Do Windows

```
SELECT category, product_id, list_price,
       RANK() OVER (ORDER BY list_price DESC)
       AS rank
FROM Products
WHERE is_number(list_price)
ORDER BY rank;
```

WIN_FUNC() OVER(ORDER BY x)

Evaluation steps:

1. Order the whole input tuple stream by x
2. Compute window function for each tuple


```
[
  { "category": "Meat & Seafood", "product_id": "X0401", "rank": 1, "list_price": 59.94 },
  { "category": "Meat & Seafood", "product_id": "HW481", "rank": 2, "list_price": 34.97 },
  { "category": "Baby Care", "product_id": "Y7KB7", "rank": 3, "list_price": 32.99 },
  { "category": "Pet Care", "product_id": "4S9UJ", "rank": 4, "list_price": 28.29 },
  { "category": "Personal Care & Health", "product_id": "37YQC", "rank": 5, "list_price": 26.99 },
  { "category": "Pet Care", "product_id": "3QQEP", "rank": 5, "list_price": 26.99 },
  { "category": "Baby Care", "product_id": "84G67", "rank": 5, "list_price": 26.99 },
  { "category": "Baby Care", "product_id": "9S30I", "rank": 5, "list_price": 26.99 },
  { "category": "Personal Care & Health", "product_id": "YE4GB", "rank": 5, "list_price": 26.99 },
  { "category": "Pet Care", "product_id": "8IDLX", "rank": 10, "list_price": 26.29 },
   ...
]
```

Partitioned Windows

```
SELECT category, product_id, list_price,
       RANK() OVER (PARTITION BY category
                    ORDER BY list_price DESC)
       AS rank
FROM Products
WHERE is_number(list_price)
ORDER BY rank, category;
```

WIN_FUNC() OVER(PARTITION BY x
ORDER BY y)

Evaluation steps:

1. Partition tuple stream by x
2. Order tuples within each partition by y
3. Compute window function for each tuple within each partition


```
[
  { "category": "Baby Care", "product_id": "Y7KB7", "rank": 1, "list_price": 32.99 },
  { "category": "Beverages", "product_id": "Y6YC8", "rank": 1, "list_price": 22.99 },
  { "category": "Beverages", "product_id": "8VPBX", "rank": 1, "list_price": 22.99 },
  { "category": "Beverages", "product_id": "W2KMW", "rank": 1, "list_price": 22.99 },
  { "category": "Bread & Bakery", "product_id": "MUFUS", "rank": 1, "list_price": 6.49 },
  { "category": "Breakfast & Cereal", "product_id": "ALCBL", "rank": 1, "list_price": 10.99 },
   ...
  { "category": "Baby Care", "product_id": "84G67", "rank": 2, "list_price": 26.99 },
  { "category": "Baby Care", "product_id": "9S30I", "rank": 2, "list_price": 26.99 },
  { "category": "Bread & Bakery", "product_id": "G08JV", "rank": 2, "list_price": 5.99 },
   ...
]
```

Partitioned Windows *(cont.)*

```
WITH ranked AS (
  SELECT category, product_id, list_price, RANK() OVER (
         PARTITION BY category ORDER BY list_price DESC ) AS rank
  FROM Products
  WHERE is_number(list_price)
)
SELECT ranked.*
FROM ranked
WHERE rank <= 3
ORDER BY rank, category;
```

```
[
  { "category": "Baby Care", "product_id": "Y7KB7", "rank": 1, "list_price": 32.99 },
  { "category": "Beverages", "product_id": "Y6YC8", "rank": 1, "list_price": 22.99 },
  { "category": "Beverages", "product_id": "8VPBX", "rank": 1, "list_price": 22.99 },
  { "category": "Beverages", "product_id": "W2KMW", "rank": 1, "list_price": 22.99 },
  { "category": "Bread & Bakery", "product_id": "MUFUS", "rank": 1, "list_price": 6.49 },
   ...
  { "category": "Baby Care", "product_id": "84G67", "rank": 2, "list_price": 26.99 },
  { "category": "Baby Care", "product_id": "9S30I", "rank": 2, "list_price": 26.99 },
  { "category": "Bread & Bakery", "product_id": "G08JV", "rank": 2, "list_price": 5.99 },
   ...
]
```

Running Aggregates

```
SELECT year, month, monthly_sales,
       SUM(monthly_sales) OVER(ORDER BY month)
       AS running_total
FROM Orders o
LET year = GET_YEAR(DATETIME(o.time_placed)),
    month = GET_MONTH(DATETIME(o.time_placed))
GROUP BY year, month
LET monthly_sales = ROUND(SUM(o.total_price))
HAVING year = 2020
ORDER BY month;
```

AGG_FUNC() OVER(PARTITION BY x
ORDER BY y
frame_spec?)

Evaluation steps:

1. Partition tuple stream by x
2. Order tuples within each partition by y
3. Determine which tuples belong to the aggregation frame for each tuple within each partition
4. Compute aggregate function over each frame

Default frame_spec is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

```
[
     { "year": 2020, "month": 1, "monthly_sales": 37767.0, "running_total": 37767.0 }
     { "year": 2020, "month": 2, "monthly_sales": 34630.0, "running_total": 72397.0 }
     { "year": 2020, "month": 3, "monthly_sales": 72565.0, "running_total": 144962.0 }
     { "year": 2020, "month": 4, "monthly_sales": 92997.0, "running_total": 237959.0 }
     { "year": 2020, "month": 5, "monthly_sales": 95525.0, "running_total": 333484.0 }
     { "year": 2020, "month": 6, "monthly_sales": 97771.0, "running_total": 431255.0 }
     { "year": 2020, "month": 7, "monthly_sales": 106498.0, "running_total": 537753.0 }
     { "year": 2020, "month": 8, "monthly_sales": 103911.0, "running_total": 641664.0 }
]
```

More Windows

```
SELECT q, year, q_sales, q_sales_prev_year, q_sales_growth_pct
FROM (
  SELECT q, year, ROUND(SUM(o.total_price)) AS q_sales
  FROM Orders o
  LET year = GET_YEAR(DATETIME(o.time_placed)),
      q = GET_MONTH(DATETIME(o.time_placed)) DIV 4
  GROUP BY year, q
) AS qs
LET q_sales_prev_year = LAG(q_sales) OVER (PARTITION BY q ORDER BY year),
    q_sales_growth = (q_sales - q_sales_prev_year) / q_sales_prev_year,
    q_sales_growth_pct = TO_STRING( TO_BIGINT( 100 * q_sales_growth) ) || "%"
ORDER BY q, year;
```

LAG(x) OVER(PARTITION BY y, ORDER BY z)

Return previous value of x within the partition (or NULL if there's no previous tuple)

```
[
     { "q": 0, "year": 2018, "q_sales": 7096.0,   "q_sales_prev_year": null,
                                                  "q_sales_growth_pct": null }
     { "q": 0, "year": 2019, "q_sales": 56641.0,  "q_sales_prev_year": 7096.0,
                                                  "q_sales_growth_pct": "698%" }
     { "q": 0, "year": 2020, "q_sales": 144963.0, "q_sales_prev_year": 56641.0,
                                                  "q_sales_growth_pct": "155%" }
     { "q": 1 ... }, ... { "q": 2 … }, ...
]
```

Lab 3: Advanced Analytics

Q1	Q2
Create a report showing sales by product category each year. It should also include a total of sales for each category (over all years) and a grand total of all sales (all categories, all years). The report rows should be ordered by category and by year within each category. Hint: use datasets: Orders, Products Hint: to get order year use GET_YEAR(DATETIME(o.time_placed))	Create a report showing monthly sales and their running totals of products in the "Beverages" category in California in 2020 Hint: use datasets: Orders, Products, Stores Hint: to get order month use GET_MONTH(DATETIME(o.time_placed))
{ "category": null, "year": null, "sales": ... } { "category": "Baby Care", "year": null, "sales": ...} { "category": "Baby Care", "year": 2018, "sales": ...} { "category": "Baby Care", "year": 2019, "sales": ...} { "category": "Baby Care", "year": 2020, "sales": ...} { "category": "Beverages", "year": null, "sales": …} { "category": "Beverages", "year": 2018, "sales": ...} { "category": "Beverages", "year": 2019, "sales": ...} { "category": "Beverages", "year": 2020, "sales": ...} ...	{ "month": 1, "sales": ..., "running_total": ... } { "month": 2, "sales": ..., "running_total": ... } { "month": 3, "sales": ..., "running_total": ... } { "month": 4, "sales": ..., "running_total": ... } { "month": 5, "sales": ..., "running_total": ... } { "month": 6, "sales": ..., "running_total": ... } { "month": 7, "sales": ..., "running_total": ... } { "month": 8, "sales": ..., "running_total": ... }

Lab 3: Q1 - Q2 Answers

Q1	Q2

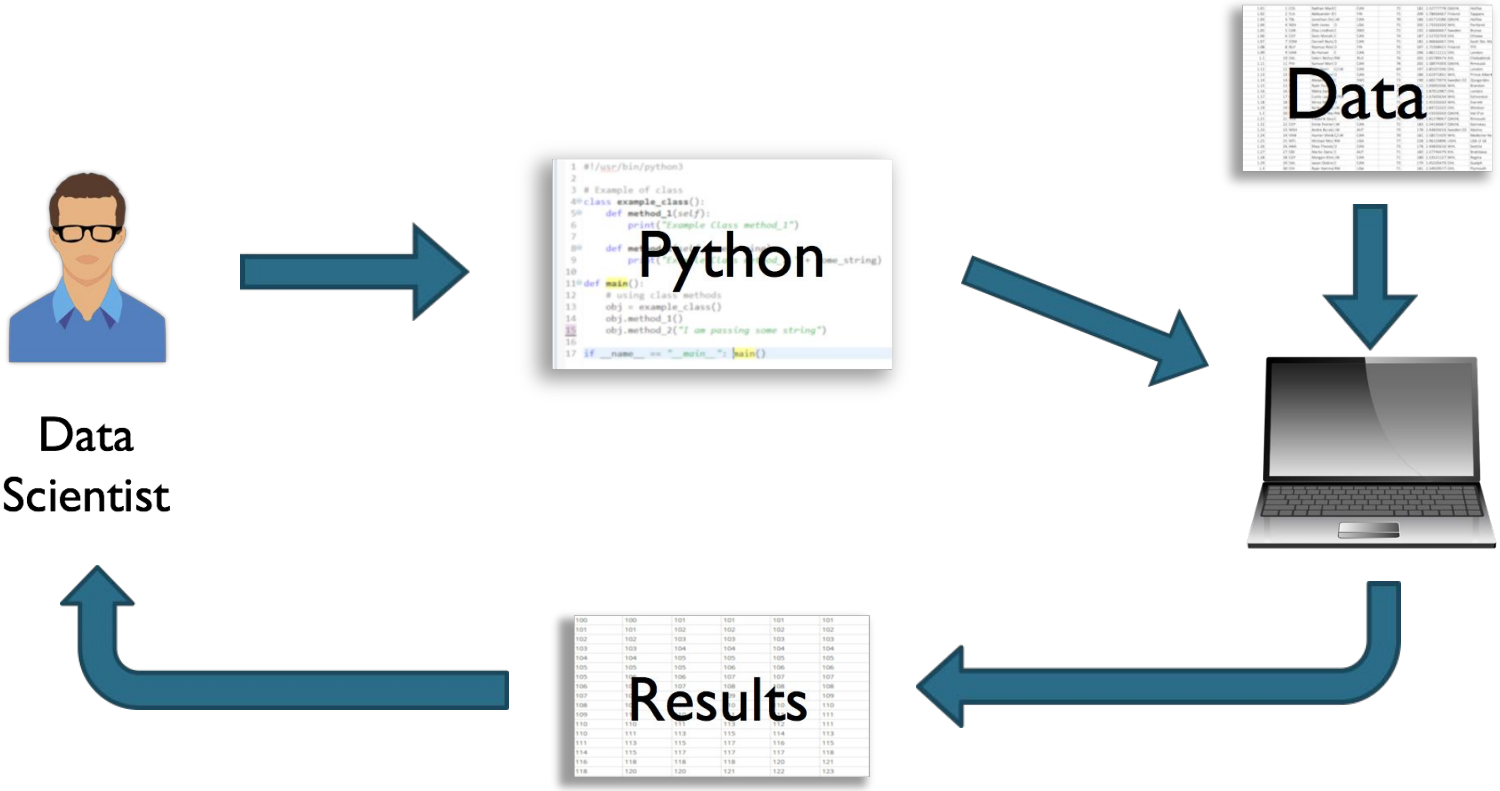
Create a report showing sales by product category for each year. It should also include a summary of sales in each category for all years and a grand total of all sales. The report rows should be ordered by category and by year within each category.
 	
SELECT category, year, sales
FROM Orders AS o
UNNEST o.items AS i
JOIN Products AS p
  ON i.product_id = p.product_id
LET year =
  GET_YEAR(DATETIME(o.time_placed))
GROUP BY ROLLUP(p.category, year)
LET sales =
  ROUND(SUM(i.qty * i.selling_price))
ORDER BY category, year;
	Create a report showing monthly sales and their running totals of products in "Beverages" category in California in 2020
	
	SELECT month, sales,
	  SUM(sales) OVER(ORDER BY month)
	  AS running_total
	FROM Orders AS o
	UNNEST o.items AS i
	JOIN Products AS p
	  ON i.product_id = p.product_id
	JOIN Stores AS s
	  ON o.store_id = s.store_id
	LET year = GET_YEAR(DATETIME(o.time_placed)),
	  month = GET_MONTH(DATETIME(o.time_placed))
	WHERE year = 2020
	  AND s.address.state = "CA"
	  AND p.category="Beverages"
	GROUP BY month
	LET sales = ROUND(SUM(i.qty * i.selling_price));

What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
- Analytical features of SQL++ (w/hands-on exercises)
  - Aggregation and grouping (vs. SQL)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

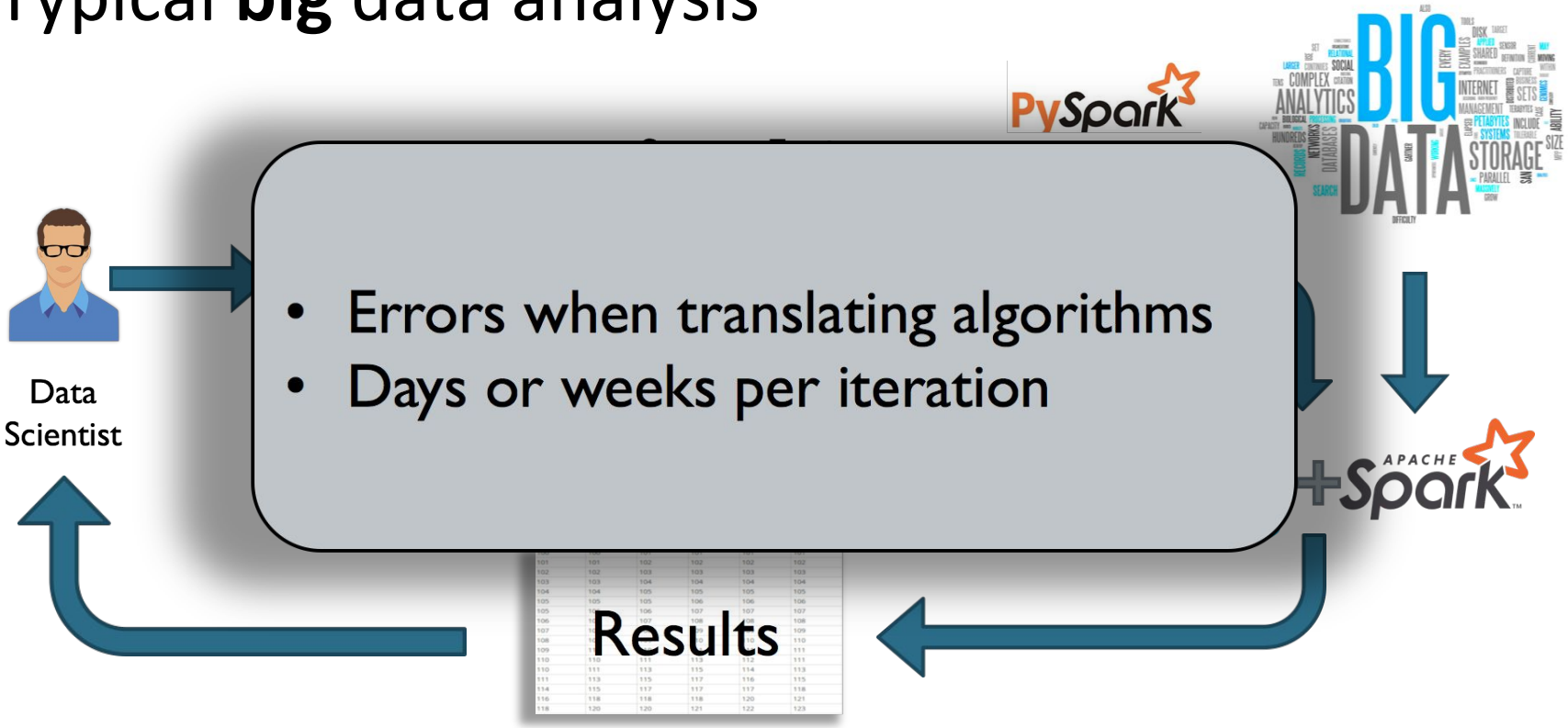
AsterixDB and ML-Based Analytics

Typical **small** data analysis



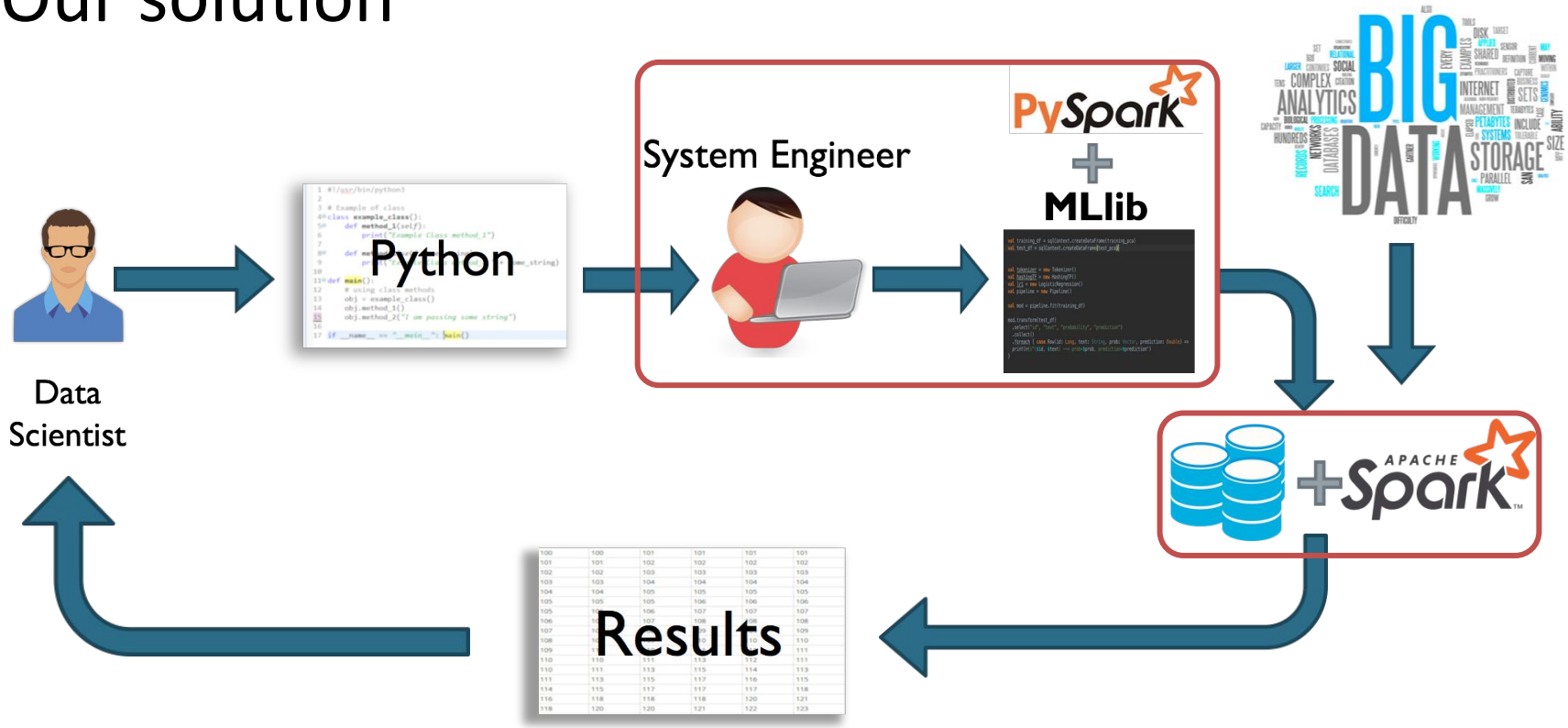
AsterixDB and ML-Based Analytics

Typical **big** data analysis



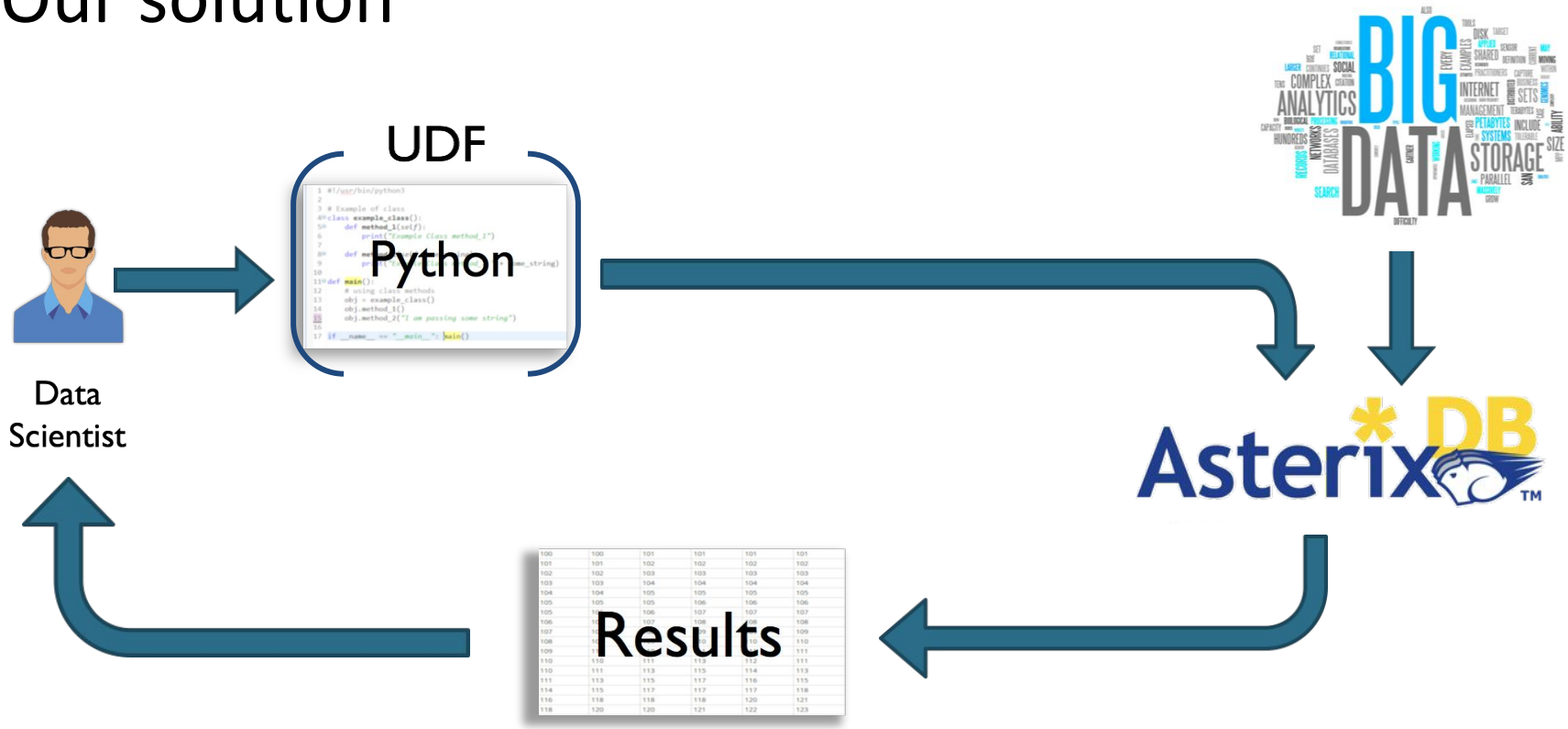
AsterixDB and ML-Based Analytics

Our solution



AsterixDB and ML-Based Analytics

Our solution



What to Expect Today

- Quick overview of Apache AsterixDB
- Connecting to AsterixDB instances in AWS
- SQL++ for basic JSON querying and manipulation
  - SQL++ vs. SQL (w/hands-on exercises)
- Analytical features of SQL++ (w/hands-on exercises)
  - Aggregation and grouping (vs. SQL)
  - Grouping sets, rollups, and cubes (oh my 😊)
  - Window functions in SQL and SQL++
- Upcoming data science support (demo)
  - Python UDFs (including ScikitLearn)

AsterixDB Python UDF Demo

Training data: https://www.kaggle.com/crowdflower/twitter-airline-sentiment

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativereason	negativereason_confidence	airline
0	570306133677760513	neutral	1.0000	NaN	NaN	Virgin America
1	570301130888122368	positive	0.3486	NaN	0.0000	Virgin America
2	570301083672813571	neutral	0.6837	NaN	NaN	Virgin America
3	570301031407624196	negative	1.0000	Bad Flight	0.7033	Virgin America
4	570300817074462722	negative	1.0000	Can't Tell	1.0000	Virgin America

3 sentiments: Positive, Neutral, Negative

sentiment.py

```
import pickle
import os
class model(object):

    def __init__(self):

        pickle_path = os.path.join(os.path.dirname(__file__), 'sentiment_model')
        f = open(pickle_path,'rb')
        self.pipeline = pickle.load(f)
        f.close()

    def getSentiment(self, *args):
        return self.pipeline.predict(args[0])[0]
```

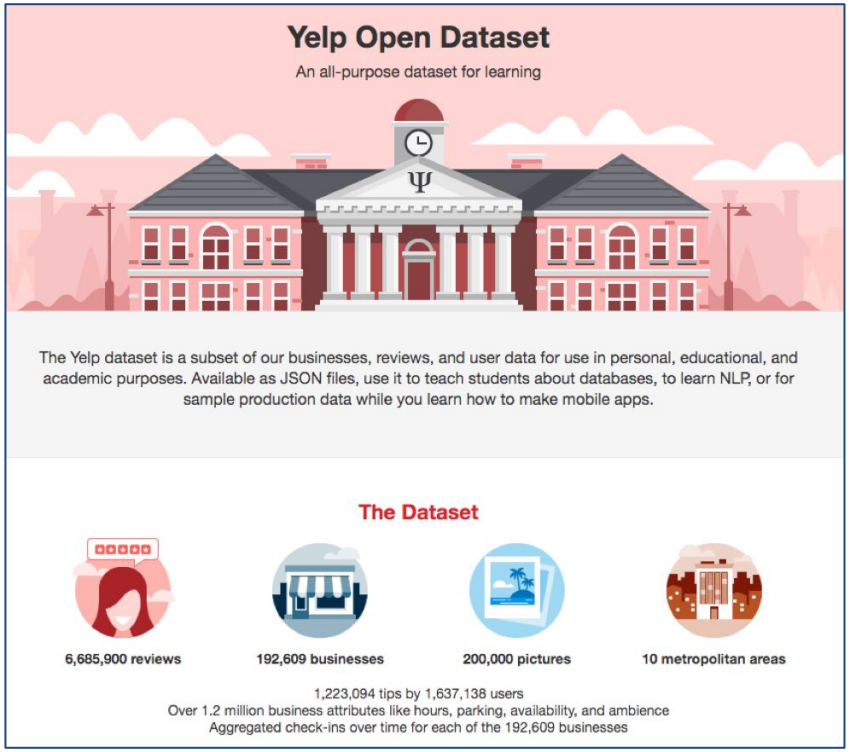
Sentiment Classifier with Scikit-Learn

```
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import LogisticRegression
import pickle

tweets = read_csv("Airline-Sentiment.csv")
X = tweets["text"]
y = tweets["sentiment"]

X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    random_state=111,
                                                    test_size=0.2)
model = Pipeline([
    ('vectorizer', CountVectorizer()),
    ('transformer', TfidfTransformer()),
    ('classifier', LogisticRegression(solver='sag',
                                      multi_class='multinomial'))
])

model.fit(X_train, y_train)
predictions = model.predict(X_test)

pickle.dump(model, open("sentiment_model", 'wb'))
```

AsterixDB Python UDF Demo

```
CREATE FUNCTION getSciKitSentiment(text)
    AS "sentiment", "model.getSentiment"
    AT sklearn;
```

```
CREATE TYPE businessType AS {
    business_id: string
};
CREATE TYPE reviewType AS {
    review_id: string,
    business_id: string,
    text: string
};

CREATE DATASET businesses(businessType)
    PRIMARY KEY business_id;
CREATE DATASET reviews(reviewType)
    PRIMARY KEY review_id;
```



Demo data

That's Basically It...!

- ***Apache AsterixDB*** Big Data Management System
- Apply MPP parallelism to NoSQL analytics with SQL++!
- Available for applications, teaching, research, ...
- Committers from all over the globe (quite literally)
- We'd be happy to help you get started, if interested!

http://asterixdb.apache.org

→ Questions? ←

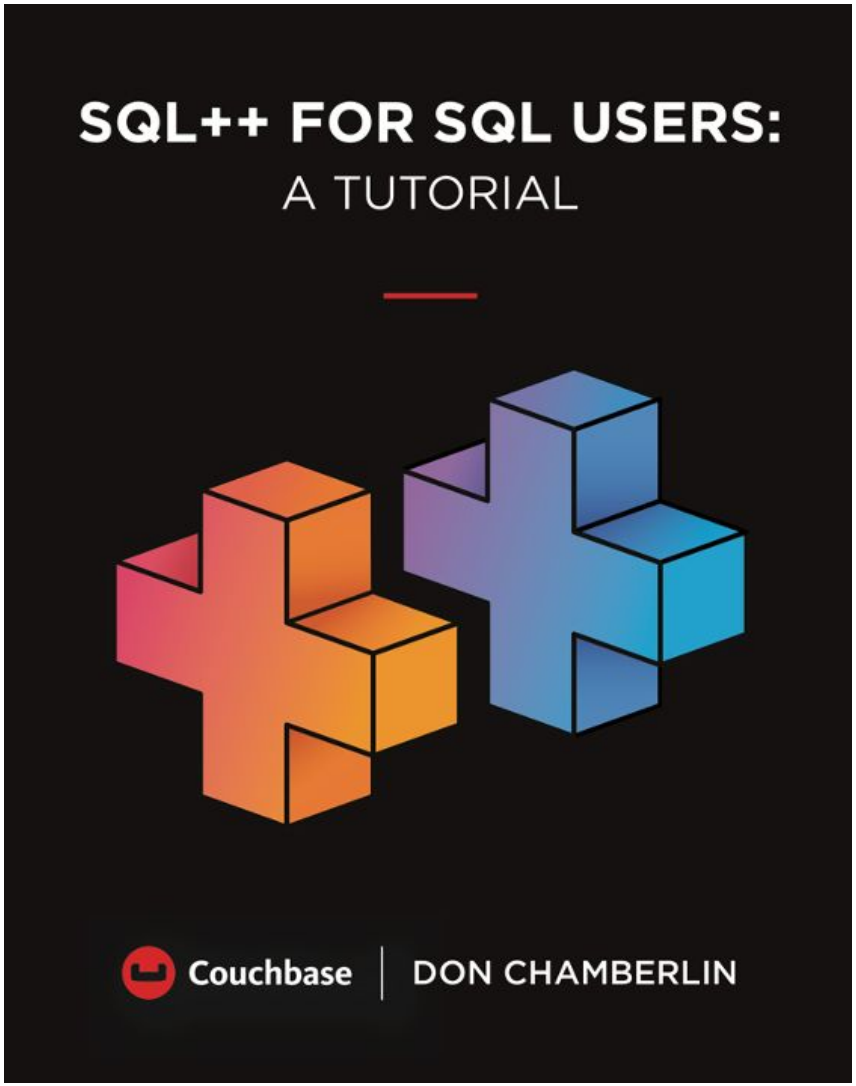
THE APACHE SOFTWARE FOUNDATION

For More: SQL++ Book (or Tutorial)

D. Chamberlin
SQL++ for SQL Users: A Tutorial

or

[N1QL for Analytics Query Language Tutorial]




AsterixDB
more engine less trunk