Big Data Analytics with Apache AsterixDB

Michael Carey
Computer Science Department
UC Irvine
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  – SQL++ vs. SQL
  – Basic aggregation and grouping (vs. SQL)
• Analytical features of SQL++
  – Grouping sets, rollups, and cubes (oh my 😊)
  – Window functions in SQL and SQL++
• Upcoming data science support (preview)
  – Python UDFs (including ScikitLearn)
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  – SQL++ vs. SQL
  – Basic aggregation and grouping (vs. SQL)
• Analytical features of SQL++
  – Grouping sets, rollups, and cubes (oh my 😊)
  – Window functions in SQL and SQL++
• Upcoming data science support (preview)
  – 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!
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
CREATE DATAVERSE ShopALot;
USE ShopALot;

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

A valid shopper object instance
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
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;
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)

```json
{
    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: "ZQLZ0"
      }
    ]
}
```
You Can Give It a Try ...!
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  – SQL++ vs. SQL
  – Basic aggregation and grouping (vs. SQL)
• Analytical features of SQL++
  – Grouping sets, rollups, and cubes (oh my 😊)
  – Window functions in SQL and SQL++
• Upcoming data science support (preview)
  – Python UDFs (including ScikitLearn)
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  – SQL++ vs. SQL
  – Basic aggregation and grouping (vs. SQL)
• Analytical features of SQL++
  – Grouping sets, rollups, and cubes (oh my 😊)
  – Window functions in SQL and SQL++
• Upcoming data science support (preview)
  – 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"
  }
]
```
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"
  }
]
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;
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
}]

Just Like SQL...
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]
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" }
    ]
  }
},
...
SELECT VALUE product_id
FROM StockedBy
WHERE store_id = "C4N2L";

[ "T1P2J",
  "TJHLQ",
  "MUFUS"
]
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";
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";
Quiz Time!

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

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];
```
SQL Pitfalls and the Value of VALUE

```sql
SELECT *
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++ “best guesses” that Orders is a field of Orders
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);

Standard SQL would apply “flat world” row/column coercion magic
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);

SQL++ SELECT statements always return collections (not scalars)
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 o2);

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

We know the subquery returns just one value, so we extract it this way
SELECT * 
FROM Orders 
WHERE total_price = 
  (SELECT MAX(total_price) FROM Orders);

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

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

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

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

[ "KMK3F",
  "OE4HV",
  "XCPVZ"
]
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 EVERY i IN o.items
    SATISFIES i.selling_price >= 70.00
    AND ARRAY_COUNT(o.items) > 0;

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

```json
{
  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: "P4XLS5"
    },
    {
      item_id: "SAB4K",
      qty: 2,
      selling_price: 4.94,
      product_id: "ZQLZO"
    }
  ]
}
```
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?
Have I “Missed” Anything?

```sql
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

```sql
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"
}]
HW 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]
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  – SQL++ vs. SQL
  – Basic aggregation and grouping (vs. SQL)
• Analytical features of SQL++
  – Grouping sets, rollups, and cubes (oh my 😊)
  – Window functions in SQL and SQL++
• Upcoming data science support (preview)
  – Python UDFs (including ScikitLearn)
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 
    },
    ...
]
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;

<table>
<thead>
<tr>
<th>s.address.state</th>
<th>s</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>AK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s_{THLUS}</td>
<td>o_{4QR5P}</td>
</tr>
<tr>
<td></td>
<td>s_{THLUS}</td>
<td>o_{4WUE6}</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>AL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s_{0HKZ3}</td>
<td>o_{0QDFV}</td>
</tr>
<tr>
<td></td>
<td>s_{0HKZ3}</td>
<td>o_{0SVOR}</td>
</tr>
<tr>
<td></td>
<td>s_{0HKZ3}</td>
<td>o_{125PT}</td>
</tr>
<tr>
<td></td>
<td>s_{0HKZ3}</td>
<td>o_{2PJ4Y}</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

... + 45 more
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;

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")
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;

{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"
    }
  ]
},
...}
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++ 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;

[{
    "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" }
        ...
    ]
}]

This could be any query over the group! (Notice that FROM came first, BTW...)
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  
  }
  ...
]
HW 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.
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  — SQL++ vs. SQL
  — Basic aggregation and grouping (vs. SQL)
• Analytical features of SQL++
  — Grouping sets, rollups, and cubes (oh my 😊)
  — Window functions in SQL and SQL++
• Upcoming data science support (preview)
  — 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...
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  – SQL++ vs. SQL
  – Basic aggregation and grouping (vs. SQL)
• Analytical features of SQL++
  – Grouping sets, rollups, and cubes (oh my 😊)
  – Window functions in SQL and SQL++
• Upcoming data science support (preview)
  – 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 }
]
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 },
    
]
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  – SQL++ vs. SQL
  – Basic aggregation and grouping (vs. SQL)
• Analytical features of SQL++
  – Grouping sets, rollups, and cubes (oh my 😊)
  – Window functions in SQL and SQL++
• Upcoming data science support (preview)
  – Python UDFs (including ScikitLearn)
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;

[ 
  { "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 },
... ]
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;

[  { "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 },
   ...
]
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  
    },  
    ...
]}
Running Aggregates

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

```
[ { "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 } ]
```
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;

[ 
  { "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 ... }, ...
]
# HW 3: Advanced Analytics

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
</tr>
</thead>
</table>
| 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. <br>Hint: use datasets: Orders, Products
| Create a report showing monthly sales and their running totals of products in the "Beverages" category in California in 2020 <br>Hint: use datasets: Orders, Products, Stores |
| { "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": ... }  
... |
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  – SQL++ vs. SQL
• Analytical features of SQL++
  – Aggregation and grouping (vs. SQL)
  – Grouping sets, rollups, and cubes (oh my 😊)
  – Window functions in SQL and SQL++
• Upcoming data science support (preview)
  – Python UDFs (including ScikitLearn)
AsterixDB and ML-Based Analytics

Typical small data analysis

Data Scientist → Python → Results → Data
AsterixDB and ML-Based Analytics

Typical **big** data analysis

- Errors when translating algorithms
- Days or weeks per iteration

Results
AsterixDB and ML-Based Analytics

Our solution

Data Scientist -> Python -> System Engineer -> PySpark + MLlib -> Apache Spark -> Results
AsterixDB and ML-Based Analytics

Our solution

Data Scientist → Python → UDF → AsterixDB

Results
Outline

• Quick overview of Apache AsterixDB
• SQL++ for basic JSON querying and manipulation
  – SQL++ vs. SQL
• Analytical features of SQL++
  – Aggregation and grouping (vs. SQL)
  – Grouping sets, rollups, and cubes (oh my 😊)
  – Window functions in SQL and SQL++
• Upcoming data science support (preview)
  – Python UDFs (including ScikitLearn)
AsterixDB Python UDF Example

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

<table>
<thead>
<tr>
<th>tweet_id</th>
<th>airline_sentiment</th>
<th>airline_sentiment_confidence</th>
<th>negativerason</th>
<th>negativerason_confidence</th>
<th>airline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>neutral</td>
<td>1.0000</td>
<td>NaN</td>
<td>NaN</td>
<td>Virgin America</td>
</tr>
<tr>
<td>1</td>
<td>positive</td>
<td>0.3486</td>
<td>NaN</td>
<td>0.0000</td>
<td>Virgin America</td>
</tr>
<tr>
<td>2</td>
<td>neutral</td>
<td>0.6837</td>
<td>NaN</td>
<td>NaN</td>
<td>Virgin America</td>
</tr>
<tr>
<td>3</td>
<td>negative</td>
<td>1.0000</td>
<td>Bad Flight</td>
<td>0.7033</td>
<td>Virgin America</td>
</tr>
<tr>
<td>4</td>
<td>negative</td>
<td>1.0000</td>
<td>Can’t Tell</td>
<td>1.0000</td>
<td>Virgin America</td>
</tr>
</tbody>
</table>

3 sentiments: Positive, Neutral, Negative

Sentiment Classifier with Scikit-Learn

```python
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'))
```

sentiment.py

```python
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')
sentiment.pipeline = pickle.load(f)
f.close()

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

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;

SELECT sentiment, count(*) AS cnt
FROM PizzaReviews pr
LET sentiment = getSciKitSentiment(pr.text)
WHERE pr.name = “MOD Pizza”
GROUP BY sentiment;

{
    cnt: 898,
    sentiment: "negative"
}
{
    cnt: 1,
    sentiment: "neutral"
}
{
    cnt: 344,
    sentiment: "positive"
}
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

https://boss-workshop.github.io/boss-2020/

→ Questions? ←
For More: SQL++ Book (or Tutorial)

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

or

N1QL for Analytics Query Language Tutorial
**HW 1: Q1 - Q2 Answers**

<table>
<thead>
<tr>
<th>Q1: List the first names of users that have placed orders with a total price greater than $500. <em>Only return a list of strings, not a list of objects.</em></th>
<th>Q2: List the names and addresses of stores that have a stock of at least 45 products with “Wafer” in the name.</th>
</tr>
</thead>
</table>
| **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; |
**HW 1: Q3 - Q4 Answers**

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

```sql
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");
```
# HW 2: Q1 - Q2 Answers

<table>
<thead>
<tr>
<th>Q1: List the names of users that have placed exactly 14 orders.</th>
</tr>
</thead>
</table>
| **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;** |

<table>
<thead>
<tr>
<th>Q2: For the two most frequent store categories, list the category itself along with the number of stores containing that category.</th>
</tr>
</thead>
</table>
| **SELECT SC, COUNT(*) AS category_count**  
**FROM** ShopALot.Stores S, S.categories SC  
**GROUP BY** SC  
**ORDER BY** COUNT(*) **DESC**  
**LIMIT** 2; |
**HW 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;
```
## HW 3: Q1 - Q2 Answers

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>Create a report showing monthly sales and their running totals of products in &quot;Beverages&quot; category in California in 2020</td>
</tr>
<tr>
<td>SELECT category, year, sales</td>
<td>SELECT month, sales, SUM(sales) OVER(ORDER BY month) AS running_total</td>
</tr>
<tr>
<td>FROM Orders AS o</td>
<td>AS running_total</td>
</tr>
<tr>
<td>UNNEST o.items AS i</td>
<td>FROM Orders AS o</td>
</tr>
<tr>
<td>JOIN Products AS p</td>
<td>UNNEST o.items AS i</td>
</tr>
<tr>
<td>ON i.product_id = p.product_id</td>
<td>JOIN Products AS p</td>
</tr>
<tr>
<td>LET year = GET_YEAR(DATETIME(o.time_placed))</td>
<td>ON i.product_id = p.product_id</td>
</tr>
<tr>
<td>GROUP BY ROLLUP(p.category, year)</td>
<td>JOIN Stores AS s</td>
</tr>
<tr>
<td>LET sales = ROUND(SUM(i.qty * i.selling_price))</td>
<td>ON o.store_id = s.store_id</td>
</tr>
<tr>
<td>ORDER BY category, year;</td>
<td>LET year = GET_YEAR(DATETIME(o.time_placed)), month = GET_MONTH(DATETIME(o.time_placed)) WHERE year = 2020 AND s.address.state = &quot;CA&quot; AND p.category=&quot;Beverages&quot; GROUP BY month LET sales = ROUND(SUM(i.qty * i.selling_price));</td>
</tr>
</tbody>
</table>