

# N1QL Queries

ANJU MUNOTH

# N1QL Language Structure

- ▶ N1QL language structure--composed of statements, expressions, and comments.

## Statements

- ▶ N1QL statements are categorized into the following groups:
  - ❖ Data Definition Language (DDL) statements to create indexes, modify indexes, and drop indexes.
  - ❖ Data Manipulation Language (DML) statements to select, insert, update, delete, and upsert data into JSON documents

# Queries

- ▶ A N1QL query is a string parsed by the query service.
- ▶ N1QL query language is based on SQL, but designed for structured and flexible JSON documents.
- ▶ As with SQL, can have nested sub-queries.
- ▶ N1QL queries run on JSON documents, and can query over multiple documents by using the JOIN clause.
- ▶ Because data in N1QL can be nested, there are operators and functions that let you navigate through nested arrays.
- ▶ Because data can be irregular, you can specify conditions in the WHERE clause to retrieve data.
- ▶ Can use standard GROUP BY, ORDER BY, LIMIT, and OFFSET clauses as well as a rich set of functions to transform the results as needed.

# Results

- ▶ Result for each query is a set of JSON documents.
- ▶ Returned document set is not required to be uniform, though it can be.
- ▶ SELECT statement that specifies a fixed set of attribute (column) names results in a uniform set of documents and a SELECT statement that specifies the wild card (\*) results in a non-uniform result set.
- ▶ Only guarantee is that every returned document meets the query criteria.

# Paths

- ▶ Paths provide a method for finding data in document structures without having to retrieve the entire document or handle it within an application.
- ▶ Any document data can be requested and returned to an application.
- ▶ When only relevant information is returned to an application, querying bandwidth is reduced.
- ▶ One of the main differences between JSON and flat rows is that JSON supports a nested structure, allowing documents to contain other documents, also known as sub-documents.
- ▶ N1QL provides paths to support nested data.

# PATHS

- ▶ Paths use dot notation syntax to identify the logical location of an attribute within a document.
- ▶ A path is used with arrays or nested objects to get to attributes within the data structure.
- ▶ For example, to get the street from a customer order, use the path **orders.billTo.street**.
  - ▶ This path refers to the value for street in the billTo object.
- ▶ Array syntax in the path can also be used to get to information.
- ▶ For example, the path **orders.items[0].productId** evaluates to the productId value for the first array element under the order item, items.

# Parameterized Queries

- ▶ N1QL allows the use of placeholders to declare dynamic query parameters. For example:

```
SELECT airportname FROM `travel-sample` WHERE city=$1
```

- ▶ The \$1 is a positional placeholder.
- ▶ When the query is constructed, it may receive arguments, with each argument being used as the placeholder value in the query.
- ▶ Thus, \$1 refers to the first argument, \$2 to the second, and so on.

# Parameterized Queries

- ▶ Placeholders may also be named.
- ▶ Useful when there are many query parameters and ensuring that they are all in the correct order may be cumbersome.
- ▶ Name query placeholders take the form of \$name.

```
SELECT airportname FROM `travel-sample`
```

```
WHERE country=$country
```

```
AND geo.alt > $altitude      AND (geo.lat BETWEEN $min_lat AND $max_lat) AND (geo.lon BETWEEN  
$min_lon AND $max_lon);
```

- ▶ Can set positional and named parameter values using cbq or the N1QL REST API when you run the query.



# Expressions

aggregate

arithmetic

collection

comparison

conditional

construction

function call

identifier

literal-value

logical

nested  
expression

string

subquery

# Comments

## Block Comments

- `/* [ [text] | [newline] ]+ */`
- Starts with `/*` and ends with `*/`.
- Query engine ignores the start and end markers `/* */`, and any text between them.
- May start on a new line, or in the middle of a line after other N1QL statements.
- May contain line breaks.

## Line Comments

- `-- [text]`
- Starts with two hyphens `--`.
- Ignores the two hyphens, and any text following them up to the end of the line.
- May start on a new line, or in the middle of a line after other N1QL statements.
- May not contain line breaks.

# Nested Path Expressions

- ▶ In N1QL, nested paths indicate an expression to access nested sub-documents within a JSON document or expression.
- ▶ For example, the latitude of a location in the type="airport" documents in the `travel-sample` are in the geo sub-document and can be addressed as the nested path `travel-sample.geo.lat`:

```
SELECT airportname, city, geo, round(geo.lat) as latitude  
FROM `travel-sample` t1  
WHERE t1.type = "airport" LIMIT 1;
```

# Array Operations

# ANY with one matching element

- ▶ Retrieve the details of KL flight schedules from Albuquerque (ABQ) to Atlanta (ATL) if any of the flights are after 23:40.

```
SELECT * FROM `travel-sample` WHERE type="route"  
AND airline="KL"  
AND sourceairport="ABQ"  
AND destinationairport="ATL"  
AND ANY departure IN schedule SATISFIES departure.utc > "23:40" END;
```

# ANY with atleast one matching elements

- ▶ ANY tests whether any element in an array matches a specified condition.
- ▶ (If the array is empty, then no element in the array is deemed to match the condition.)
- ▶ If the array is non-empty and at least one element in the array matches the specified condition, then the operator returns TRUE; otherwise, it returns FALSE

# EVERY

- ▶ EVERY tests whether every element in an array matches a specified condition. (If the array is empty, then every element in the array is deemed to match the condition.)
- ▶ Return Values
- ▶ If the array is empty, or if the array is non-empty and every element in the array matches the specified condition, then the operator returns TRUE; otherwise, it returns FALSE.

# EVERY with all matching elements

- Retrieve the details of KL flight schedules from Albuquerque (ABQ) to Atlanta (ATL) if all of the flights are after 00:35.

```
SELECT * FROM `travel-sample` WHERE type="route"
```

```
AND airline="KL"
```

```
AND sourceairport="ABQ"
```

```
AND destinationairport="ATL"
```

```
AND EVERY departure IN schedule SATISFIES departure.utc > "00:35" END;
```



# ANY AND EVERY

- ▶ ANY AND EVERY tests whether every element in an array matches a specified condition. (If the array is empty, then no element in the array is deemed to match the condition.)
- ▶ Return Values
- ▶ If the array is non-empty and every element in the array matches the specified condition, then the operator returns TRUE; otherwise, it returns FALSE.

# ANY AND EVERY

This example tests the ANY AND EVERY operator with an empty array.

```
SELECT ANY AND EVERY v IN [] SATISFIES v = "abc" END AS universal;
```

In this case, the operator returns false.

# Range Transformations

- ▶ Range transformations (ARRAY, FIRST, and OBJECT) enable you to map and filter elements and attributes from an input array.
- ▶ Use the IN and WITHIN operators to range through the array.

- ▶ Syntax

```
range-xform ::= ( ( ARRAY | FIRST ) | OBJECT name-expr ':' ) var-expr  
              FOR range [ WHEN cond ] END
```

# ARRAY

- ▶ ARRAY operator generates a new array, using values in the input array.
- ▶ Return Values
- ▶ The operator returns a new array, which contains one element for each element in the input array.
- ▶ If the WHEN clause is specified, only elements in the input array which satisfy the WHEN clause are considered.
- ▶ The value of each element in the output array is the output of the var-expr argument for one element in the input array.
- ▶ If the input array is empty, or no elements in the input array satisfy the WHEN clause, the operator returns an empty array.

# ARRAY with array of objects

- List the details of KL flights from Albuquerque to Atlanta on Fridays.

```
SELECT ARRAY v FOR v IN schedule WHEN v.day = 5 END AS fri_flights
FROM `travel-sample` WHERE type = "route"
AND airline="KL"
AND sourceairport="ABQ"
AND destinationairport="ATL";
```

# ARRAY with multiple range terms

- List the details of KL flights from Albuquerque to Atlanta on Fridays after 7pm only.

```
SELECT ARRAY v
```

```
  FOR v IN schedule, w IN schedule WHEN v.utc > "19:00" AND w.day = 5 END
```

```
  AS fri_evening_flights
```

```
FROM `travel-sample` WHERE type="route"
```

```
  AND airline="KL"
```

```
  AND sourceairport="ABQ"
```

```
  AND destinationairport="ATL";
```

# ARRAY with multiple range terms

- List the details of KL flights from Albuquerque to Atlanta on Fridays after 7pm only.

```
SELECT ARRAY v
```

```
  FOR v IN schedule WHEN v.utc > "19:00" AND v.day = 5 END
```

```
  AS fri_evening_flights
```

```
FROM `travel-sample` WHERE type="route"
```

```
  AND airline="KL"
```

```
  AND sourceairport="ABQ"
```

```
  AND destinationairport="ATL";
```

# ARRAY with position variable

- List the first two KL flights from Albuquerque to Atlanta. This example uses the position variable *i* to return just the first two elements in the input array.

```
SELECT ARRAY v FOR i:v IN schedule WHEN i < 2 END AS two_flights  
FROM `travel-sample` WHERE type="route"  
AND airline="KL"  
AND sourceairport="ABQ"  
AND destinationairport="ATL";
```



# FIRST

- ▶ FIRST operator generates a new value, using a single value in the input array.
- ▶ Return Values
- ▶ The operator returns the output of the var-expr argument for the first element in the input array.
- ▶ If the WHEN clause is specified, only elements in the input array which satisfy the WHEN clause are considered.
- ▶ If the input array is empty, or no elements in the input array satisfy the WHEN clause, the operator returns MISSING.

# FIRST

- List the first KL flight from Albuquerque to Atlanta after 7pm.

```
SELECT FIRST v FOR v IN schedule WHEN v.utc > "19:00" END AS first_flight  
FROM `travel-sample` WHERE type="route"  
AND airline="KL"  
AND sourceairport="ABQ"  
AND destinationairport="ATL";
```

# OBJECT

- ▶ OBJECT operator generates a new object, using values in the input array.
- ▶ Return Values
- ▶ The operator returns an object, which contains one attribute for each element in the input array.
- ▶ If the WHEN clause is specified, only elements in the input array which satisfy the WHEN clause are considered.
- ▶ The value of each attribute in the output object is the output of the var-expr argument for one element in the input array.
- ▶ The name of each attribute in the output object is specified by the name-expr argument.

# OBJECT

- ▶ This argument must be an expression that generates a unique name string for every value in the output object.
- ▶ If the expression does not generate a string, then the current attribute is not output.
- ▶ If the expression does not generate a unique name string for each value, then only the last attribute is output; all previous attributes are suppressed.
- ▶ The name-expr argument may reference the var argument or the name-var argument, or use any other expression that generates a unique value.
- ▶ If the input array is empty, or no elements in the input array satisfy the WHEN clause, the operator returns an empty object.

# OBJECT with array of objects

- List the details of KL flights from Albuquerque to Atlanta on Fridays. This example uses the UUID() function to generate a unique name for each attribute in the output object.

```
SELECT OBJECT UUID():v FOR v IN schedule WHEN v.day = 5 END AS fri_flights
FROM `travel-sample` WHERE type="route"
AND airline="KL"
AND sourceairport="ABQ"
AND destinationairport="ATL";
```

# OBJECT with position variable

- List the details of KL flights from Albuquerque to Atlanta on Fridays.

```
SELECT OBJECT "num_" || TOSTRING(i):v  
  FOR i:v IN schedule WHEN v.day = 5 END  
  AS fri_flights  
FROM `travel-sample` WHERE type="route"  
  AND airline="KL"  
  AND sourceairport="ABQ"  
  AND destinationairport="ATL";
```

# Membership and Existence

- ▶ Membership tests (IN and WITHIN) enable you to test whether a value exists within an array. Membership tests are efficient over arrays with a large number of elements — up to approximately 8000.
- ▶ Existence tests enable you to test whether an array contains any elements at all. There is one existence test: EXISTS.

# IN

- ▶ IN operator specifies the search depth to include only the current level of an array, and not to include any child or descendant arrays.
- ▶ Syntax
- ▶ `in-expr ::= search-expr [ NOT ] IN target-expr`
- ▶ IN operator evaluates to TRUE if the right-side value is an array and directly contains the left-side value.
- ▶ The NOT IN operator evaluates to TRUE if the right-side value is an array and does not directly contain the left-side value.



# IN with simple array

- Search for all airlines from the United Kingdom or France.

```
SELECT * FROM `travel-sample` AS t WHERE type = "airline"  
AND country IN ["United Kingdom", "France"];
```

# IN with array of objects

- ▶ Search for the author "Walton Wolf" in the hotel keyspace

```
SELECT * FROM `travel-sample` AS t WHERE type = "hotel" AND "Walton Wolf" IN t;
```

This results in an empty set because authors are not in the current level (the root level) of the hotel keyspace.

The authors are listed inside the reviews array (a child element) and would need the `WITHIN` keyword to search all child elements along with the root level.

# WITHIN

- ▶ WITHIN operator specifies the search depth to include the current level of an array, and all of its child and descendant arrays.
- ▶ Syntax
- ▶ `within-expr ::= search-expr [ NOT ] WITHIN target-expr`
- ▶ WITHIN operator evaluates to TRUE if the right-side value is an array and contains the left-side value as a child or descendant, that is, directly or indirectly.
- ▶ The NOT WITHIN operator evaluates to TRUE if the right-side value is an array and no child or descendant contains the left-side value.

# WITHIN

- ▶ Search all elements for the author "Walton Wolf" in the hotel documents.

```
SELECT * FROM `travel-sample` AS t WHERE type = "hotel" AND "Walton Wolf" WITHIN t;
```

# EXISTS

- ▶ EXISTS operator enables you to test whether an array has any elements, or is empty.
- ▶ This operator may be used in a SELECT, INSERT, UPDATE, or DELETE statement in combination with a subquery. The condition is met if the subquery returns at least one result.
- ▶ Syntax
- ▶ `exists-expr ::= EXISTS expr`
- ▶ If the expression is an array which contains at least one element, the operator evaluates to TRUE; otherwise, it evaluates to FALSE.

# EXISTS

- ▶ Of the 274 cities with a hotel, search for all cities that have hotels with reviews

```
SELECT DISTINCT h.city  
FROM `travel-sample` AS h WHERE type="hotel"  
AND EXISTS h.reviews;
```

# Conditional Operators

- ▶ Case expressions evaluate conditional logic in an expression.
- ▶ **Simple Case Expressions**
- ▶ **Searched Case Expressions**

# Simple case expressions

- ▶ Simple case expressions allow for conditional matching within an expression. The evaluation process is as follows:
  - The first WHEN expression is evaluated. If it is equal to the search expression, the result of this expression is the THEN expression.
  - If it is not equal, subsequent WHEN clauses are evaluated in the same manner.
  - If none of the WHEN expressions are equal to the search expression, then the result of the CASE expression is the ELSE expression.
  - If no ELSE expression was provided, the result is NULL.

**CASE expression ( WHEN expression THEN expression)**

**[ ( WHEN expression THEN expression) ]\***

**[ ELSE expression ] END**



# Searched Case Expressions

- ▶ Searched case expressions allow for conditional logic within an expression. The evaluation process is as follows:
- ▶ The first WHEN expression is evaluated.
- ▶ If TRUE, the result of this expression is the THEN expression.
- ▶ If not TRUE, subsequent WHEN clauses are evaluated in the same manner.
- ▶ If none of the WHEN clauses evaluate to TRUE, then the result of the expression is the ELSE expression.
- ▶ If no ELSE expression was provided, the result is NULL.

**CASE ( WHEN condition THEN expression )**

**[( WHEN condition THEN expression ) ]\***

**[ ELSE expression ] END**

# Example

```
SELECT  
  CASE WHEN `shipped-on`  
    IS NOT NULL THEN `shipped-on`  
    ELSE "not-shipped-yet"  
  END  
  AS shipped  
FROM orders
```

# Group by queries

# List the cities with the landmarks with the highest latitude

```
SELECT country, state, MAX(ROUND(geo.lat)) AS Max_Latitude  
FROM `travel-sample`  
WHERE country IS NOT MISSING  
AND type = "landmark"  
GROUP BY country, state  
ORDER BY Max_Latitude DESC;
```

Use the MAX() aggregate to find the highest landmark latitude in each state, group the results by country and state, and then sort in reverse order by the highest latitudes per state.

# List the states with their total number of landmarks and the lowest latitude of any landmark

```
SELECT COUNT(country) AS Total_Landmarks, state, MIN(ROUND(geo.lat)) AS Min_Latitude
FROM `travel-sample`
WHERE country IN ["France", "United States", "United Kingdom"]
AND type = "landmark"
GROUP BY state
ORDER BY Min_Latitude;
```

Use the COUNT() operator to find the total number of landmarks and use the MIN() operator to find the lowest landmark latitude in each state, group the results by state, and then sort in order by the lowest latitudes per state.

# List the number of landmarks by latitude and the state it's in

```
SELECT COUNT(country) Num_Landmarks, MIN(state) State_Name, ROUND(geo.lat) Latitude
FROM `travel-sample`
WHERE country IS NOT MISSING
AND type = "landmark"
GROUP BY ROUND(geo.lat)
ORDER BY ROUND(geo.lat);
```

Use COUNT(country) for the total number of landmarks at each latitude. At a particular latitude, the state will be the same; but an aggregate function on it is needed, so MIN() or MAX() is used to return the original value.

# Aggregate functions

```
SELECT MIN(ROUND(geo.lat)) AS Min_Lat,  
       SUM(geo.alt) AS Sum_Alt,  
       COUNT(city) AS Count_City,  
       MAX(ROUND(geo.lon)) AS Max_Lon  
FROM `travel-sample`  
WHERE geo.lat IS NOT MISSING  
AND type = "airport"  
GROUP BY (ROUND(geo.lat))  
ORDER BY (ROUND(geo.lat)) DESC;
```

# Aggregate functions

```
SELECT SUM(DISTINCT ROUND(geo.lat)) AS Sum_Lat  
FROM `travel-sample`  
WHERE geo.lat IS NOT MISSING  
AND type = "airport"  
GROUP BY ROUND(geo.lat);
```




# Aggregate functions

```
SELECT COUNT(DISTINCT ROUND(geo.lat)) AS Count_Lat,  
       SUM(DISTINCT ROUND(geo.lon)) AS Sum_Lon  
FROM `travel-sample`  
WHERE geo.lat IS NOT MISSING  
AND type = "airport"  
GROUP BY ROUND(geo.lat), ROUND(geo.lon);
```

# Aggregate functions

```
SELECT MIN(country) AS Min_Country,  
       COUNT(DISTINCT 1) AS Constant_Value,  
       MIN(ROUND(geo.lon)) AS Min_Logitude  
FROM `travel-sample`  
WHERE country IS NOT MISSING  
AND type = "airport"  
GROUP BY geo.lat;
```



# List the cities that have more than 180 landmarks.

```
SELECT city AS City, COUNT(DISTINCT name) AS Landmark_Count  
FROM `travel-sample`  
WHERE city IS NOT MISSING  
AND type = "landmark"  
GROUP BY city  
HAVING COUNT(DISTINCT name) > 180;
```

# List cities that have more than half of all landmarks.

```
SELECT city AS City, COUNT(DISTINCT name) AS Landmark_Count  
FROM `travel-sample`  
WHERE city IS NOT MISSING  
AND type = "landmark"  
GROUP BY city  
LETting MinimumThingsToSee = COUNT(DISTINCT name)  
HAVING MinimumThingsToSee > 180;
```

# ARRAY\_AGG( [ ALL | DISTINCT ] expression)

- ▶ Return Value
- ▶ With the ALL quantifier, or no quantifier, returns an array of the non-MISSING values in the group, including NULL values.
- ▶ With the DISTINCT quantifier, returns an array of the distinct non-MISSING values in the group, including NULL values.

# ARRAY\_AGG( [ ALL | DISTINCT ] expression)

- List all values of the Cleanliness reviews given.

```
SELECT ARRAY_AGG(reviews[0].ratings.Cleanliness) AS Reviews FROM `travel-sample`;
```

- List all unique values of the Cleanliness reviews given.

```
SELECT ARRAY_AGG(DISTINCT reviews[0].ratings.Cleanliness) AS Reviews  
FROM `travel-sample`;
```

# AVG( [ ALL | DISTINCT ] expression)

- ▶ Return Value
- ▶ With the ALL quantifier, or no quantifier, returns the arithmetic mean (average) of all the number values in the group.
- ▶ With the DISTINCT quantifier, returns the arithmetic mean (average) of all the distinct number values in the group.
- ▶ Returns NULL if there are no number values in the group.

# Aggregate functions

ARRAY\_AGG( [ ALL | DISTINCT ] expression )  
AVG( [ ALL | DISTINCT ] expression )  
COUNT(\*)  
COUNT( [ ALL | DISTINCT ] expression )  
COUNTN( [ ALL | DISTINCT ] expression )  
MAX( [ ALL | DISTINCT ] expression )  
MEAN( [ ALL | DISTINCT ] expression )  
MEDIAN( [ ALL | DISTINCT ] expression )  
MIN( [ ALL | DISTINCT ] expression )

STDDEV( [ ALL | DISTINCT ] expression )  
STDDEV\_POP( [ ALL | DISTINCT ] expression )  
STDDEV\_SAMP( [ ALL | DISTINCT ]  
expression )  
SUM( [ ALL | DISTINCT ] expression )  
VARIANCE( [ ALL | DISTINCT ] expression )  
VARIANCE\_POP( [ ALL | DISTINCT ]  
expression )  
VARIANCE\_SAMP( [ ALL | DISTINCT ]  
expression )  
VAR\_POP( [ ALL | DISTINCT ] expression )  
VAR\_SAMP( [ ALL | DISTINCT ] expression )



# Array Functions

ARRAY_AGG(expr)	ARRAY_PREPEND(val1, val2, ... , expr)
ARRAY_APPEND(expr, val1, val2, ...)	ARRAY_PUT(expr, val1, val2, ...)
ARRAY_AVG(expr)	ARRAY_RANGE(start_num, end_num step_num ])
ARRAY_BINARY_SEARCH(expr, val, ...)	ARRAY_REMOVE(expr, val1, val2, ...)
ARRAY_CONCAT(expr1, expr2, ...)	ARRAY_REPEAT(val, rep_int)
ARRAY_CONTAINS(expr, val)	ARRAY_REPLACE(expr, val1, val2 max_int ])
ARRAY_COUNT(expr)	ARRAY_REVERSE(expr)
ARRAY_DISTINCT(expr)	ARRAY_SORT(expr)
ARRAY_EXCEPT(expr1, expr2)	ARRAY_STAR(expr)
ARRAY_FLATTEN(expr, depth)	ARRAY_SUM(expr)
ARRAY_IFNULL(expr)	ARRAY_SYMDIFF(expr1, expr2, ...)
ARRAY_INSERT(expr, pos, val1, val2, ...)	ARRAY_SYMDIFF1(expr1, expr2, ...)
ARRAY_INTERSECT(expr1, expr2, ...)	ARRAY_SYMDIFFN(expr1, expr2, ...)
ARRAY_LENGTH(expr)	ARRAY_MAX(expr)
ARRAY_POSITION(expr, val)	ARRAY_MIN(expr)
ARRAY_UNION(expr1, expr2, ...)	ARRAY_MOVE(expr, val1, val2)

# ARRAY\_STAR(expr)

- ▶ This function converts an array of expr objects into an object of arrays.
- ▶ expr
- ▶ [Required] The input array you want to convert into an object of arrays.
- ▶ Output Values
- ▶ An object of arrays.
- ▶ If the argument is MISSING, then it returns MISSING.
- ▶ If the argument is a non-array value, then it returns NULL.

Convert a given array of two documents each with five items into an object of five arrays each with two documents.

```
SELECT ARRAY_STAR( [
  {
    "address": "Capstone Road, ME7 3JE",
    "city": "Medway",
    "country": "United Kingdom",
    "name": "Medway Youth Hostel",
    "url": "http://www.yha.org.uk"
  },
  {
    "address": "6 rue aux Juifs",
    "city": "Giverny",
    "country": "France",
    "name": "The Robins",
    "url": "http://givernyguesthouse.com/robin.htm"
  }) AS array_star;
```

Output:

```
[
  {
    "array_star": {
      "address": [
        "Capstone Road, ME7 3JE",
        "6 rue aux Juifs"
      ],
      "city": [ "Medway", "Giverny" ],
      "country": ["United Kingdom", "France" ],
      "name": [
        "Medway Youth Hostel",
        "The Robins"
      ],
      "url": [
        "http://www.yha.org.uk",
        "http://givernyguesthouse.com/robin.htm"
      ] } } ]
```

# Array References

- ▶ Can use an asterisk (\*) as an array subscript which converts the array to an object of arrays.
- ▶ The following example returns an array of the ages of the given contact's children:
- ▶ **SELECT children[\*].age FROM contacts WHERE fname = "Dave"**
- ▶ An equivalent query can be written using the array\_star() function:  
**SELECT array\_star(children).age FROM contacts WHERE fname = "Dave"**

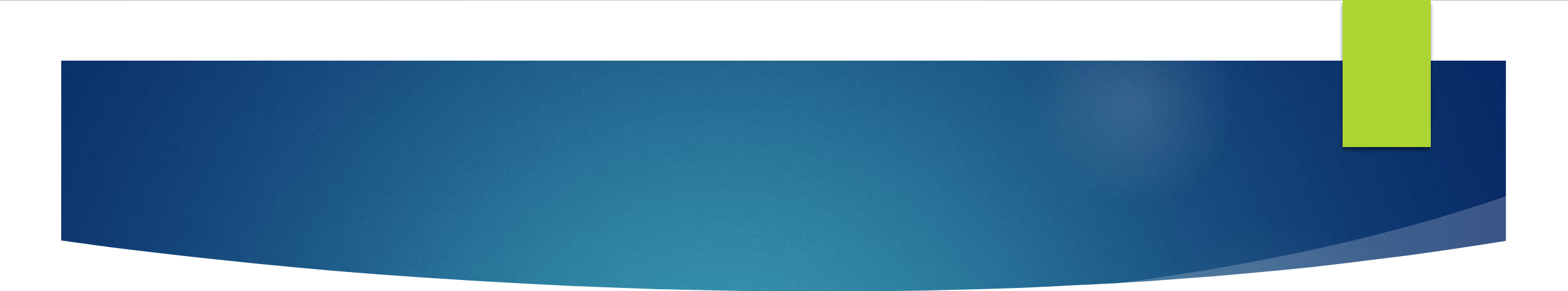
# Date Functions

- ▶ N1QL date functions return the system clock value or manipulate the datetime values, which are represented as a string or an integer.
- ▶ These functions are very useful for manipulating dates in datasets with various date formats and timezones.
- ▶ N1QL date functions accept dates in either Epoch/UNIX timestamp format or string date format.
- ▶ N1QL is then able to represent the passed date as a standardized date object internally.
- ▶ Functions whose name contains the word STR are designed to use string formats while MILLIS functions are designed to use Epoch/UNIX timestamps



CLOCK\_LOCAL([fmt])  
CLOCK\_MILLIS()  
CLOCK\_STR([fmt])  
CLOCK\_TZ(tz [, fmt])  
CLOCK\_UTC([fmt])  
DATE\_ADD\_MILLIS(date1, n, part)  
DATE\_ADD\_STR(date1, n, part)  
DATE\_DIFF\_MILLIS(date1, date2, part)  
DATE\_DIFF\_STR(date1, date2, part)  
DATE\_FORMAT\_STR(date1, fmt)  
DATE\_PART\_MILLIS(date1, part [, tz])  
DATE\_PART\_STR(date1, part)  
DATE\_RANGE\_MILLIS(date1, date2, part [,n])  
DATE\_RANGE\_STR(start\_date, end\_date,  
date\_interval [, quantity\_int ])  
WEEKDAY\_STR(date)  
WEEKDAY\_MILLIS(expr [, tz ])  
DURATION\_TO\_STR(duration)

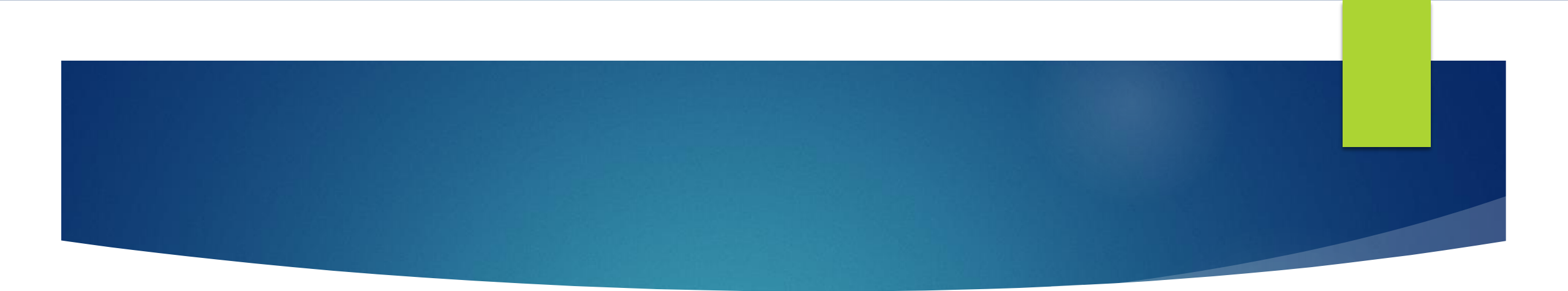
NOW\_LOCAL([fmt])  
NOW\_MILLIS()  
NOW\_TZ(tz [, fmt])  
NOW\_STR([fmt])  
NOW\_UTC([fmt])  
STR\_TO\_DURATION(duration)  
STR\_TO\_MILLIS(date1)  
STR\_TO\_UTC(date1)  
STR\_TO\_TZ(date1, tz)  
STR\_TO\_ZONE\_NAME(date1, tz)  
DATE\_TRUNC\_MILLIS(date1, part)  
DATE\_TRUNC\_STR(date1, part)  
MILLIS(date1)  
MILLIS\_TO\_LOCAL(date1 [, fmt])  
MILLIS\_TO\_STR(date1 [, fmt ])  
MILLIS\_TO\_TZ(date1, tz [, fmt])  
MILLIS\_TO\_UTC(date1 [, fmt])  
MILLIS\_TO\_ZONE\_NAME(date1, tz [, fmt])



```
SELECT CLOCK_LOCAL() as full_date,  
       CLOCK_LOCAL('invalid date') as invalid_date,  
       CLOCK_LOCAL('1111-11-11') as short_date;
```

```
SELECT CLOCK_MILLIS() AS CurrentTime;
```

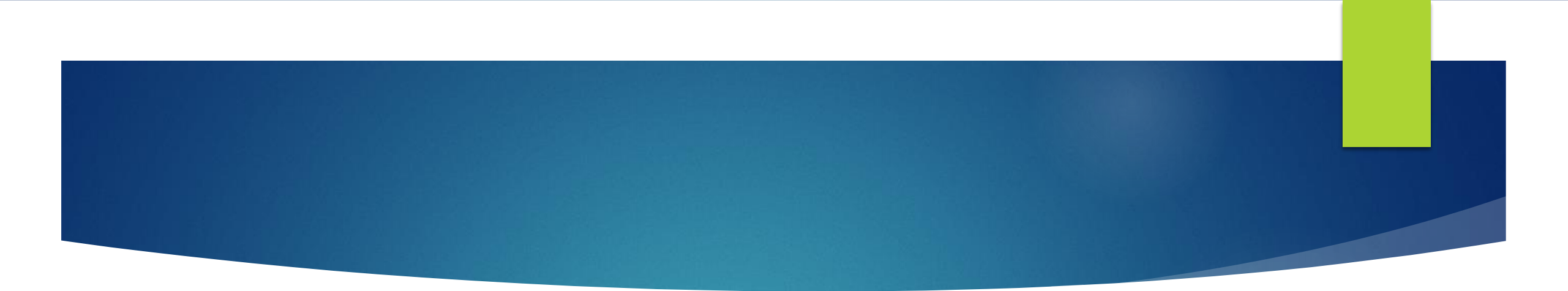
```
SELECT CLOCK_STR() as full_date,  
       CLOCK_STR('invalid date') as invalid_date,  
       CLOCK_STR('1111-11-11') as short_date;
```



```
SELECT CLOCK_TZ('UTC') as UTC_full_date,  
       CLOCK_TZ('UTC', '1111-11-11') as UTC_short_date,  
       CLOCK_TZ('invalid timezone') as invalid_timezone,  
       CLOCK_TZ('US/Eastern') as us_east,  
       CLOCK_TZ('US/Pacific') as us_west;
```

```
SELECT DATE_ADD_MILLIS(1463284740000, 3, 'day') as add_3_days,  
       DATE_ADD_MILLIS(1463284740000, 3, 'year') as add_3_years,  
       DATE_ADD_MILLIS(1463284740000, -3, 'day') as sub_3_days,  
       DATE_ADD_MILLIS(1463284740000, -3, 'year') as sub_3_years;
```





```
SELECT DATE_ADD_STR('2016-05-15 03:59:00Z', 3, 'day') as add_3_days,  
       DATE_ADD_STR('2016-05-15 03:59:00Z', 3, 'year') as add_3_years,  
       DATE_ADD_STR('2016-05-15 03:59:00Z', -3, 'day') as sub_3_days,  
       DATE_ADD_STR('2016-05-15 03:59:00Z', -3, 'year') as sub_3_years;
```

```
SELECT DATE_DIFF_STR('2016-05-18T03:59:00Z', '2016-05-15 03:59:00Z', 'day') as add_3_days,  
       DATE_DIFF_STR('2019-05-15T03:59:00Z', '2016-05-15 03:59:00Z', 'year') as add_3_years,  
       DATE_DIFF_STR('2016-05-12T03:59:00Z', '2016-05-15 03:59:00Z', 'day') as sub_3_days,  
       DATE_DIFF_STR('2013-05-15T03:59:00Z', '2016-05-15 03:59:00Z', 'year') as sub_3_years;
```

# List all hotel documents that were reviewed between two dates.

```
SELECT name, reviews[0].date  
FROM `travel-sample`  
WHERE type = "hotel"  
AND reviews[0].date BETWEEN "2013-01-01 00:00:00 +0100" AND "2014-01-01 00:00:00 +0100";
```

The same as:

```
SELECT name, reviews[0].date  
FROM `travel-sample`  
WHERE type = "hotel"  
AND reviews[0].date BETWEEN "2013-01-01 %" AND "2014-01-01 %";
```

When querying between two dates, you must specify the full date (with time and time zone) or use the wildcard character (%).

# Object Functions

OBJECT\_ADD()  
OBJECT\_CONCAT()  
OBJECT\_INNER\_PAIRS(expression)  
OBJECT\_INNER\_VALUES(expression)  
OBJECT\_LENGTH(expression)  
OBJECT\_NAMES(expression)  
OBJECT\_PAIRS(expression)  
OBJECT\_PUT()  
OBJECT\_RENAME(input\_obj, old\_field, new\_field)  
OBJECT\_REMOVE()  
OBJECT\_REPLACE(input\_obj, old\_value, new\_value)  
OBJECT\_UNWRAP(expression)  
OBJECT\_VALUES(expression)

# String Functions

CONCAT(string1, string2, ...)  
CONCAT2(separator, arg1, arg2, ...)  
CONTAINS(in\_str, search\_str)  
INITCAP(in\_str)  
LENGTH(in\_str)  
LOWER(in\_str)  
LTRIM(in\_str [, char])  
POSITION(in\_str, search\_str)  
REPEAT(in\_str, n)  
UPPER(in\_str)

REPLACE(in\_str, search\_str, replace [, n ])  
REVERSE(in\_str)  
RTRIM(in\_str [, char])  
SPLIT(in\_str [, in\_substr])  
SUBSTR(in\_str, start\_pos [, length])  
SUFFIXES(in\_str)  
TITLE(in\_str)  
TOKENS(in\_str, opt)  
TRIM(in\_str [, char])

# User-Defined Functions

- ▶ can call a user-defined function in any expression where you can call a built-in function.
- ▶ This is a Developer Preview feature(in version 6.6), intended for development purposes only. Do not use this feature in production. No Enterprise Support is provided for Developer Preview features.
- ▶ When you have created a user-defined function, you can call it in any expression, just like a built-in function. User-defined functions have the same syntax as built-in functions, with brackets () to contain any arguments.
- ▶ Name of the function may be an unqualified identifier, such as func1 or `func-1`, or a qualified identifier with a namespace, such as default:func1.
- ▶ Name of a user-defined function is case-sensitive, unlike that of a built-in function.
- ▶ Must call the user-defined function using the same case that was used when it was created.

# Arguments

- ▶ User-defined function has zero, one, or more arguments, separated by commas, just like a built-in function.
- ▶ Each argument is a N1QL expression required by the function.
- ▶ If the function was created with named parameters, you must supply all the arguments that were specified when the function was created.
- ▶ If the function was created without named parameters, it is variadic, and you can supply as many arguments as needed.

# Return Value

- ▶ The function returns one value, of any valid N1QL type.
- ▶ The result (and the data type of the result) depend on the expression or code that were used to define the function.
- ▶ If you supply the wrong number of arguments, or arguments with the wrong data type, the possible results differ, depending on whether the function was defined with or without any named parameters.
- ▶ If the function was defined with named parameters:
  - ▶ If you do not supply enough arguments, the function generates error 10104: Incorrect number of arguments.
  - ▶ If you supply too many arguments, the function generates error 10104: Incorrect number of arguments.
  - ▶ If any of the arguments have the wrong data type, the function may return unexpected results, depending on the function expression or code.

# Return Value

- ▶ If the function was defined without named parameters:
  - ▶ If you do not supply enough arguments, the function may return unexpected results, depending on the function expression or code.
  - ▶ If you supply too many arguments, the extra parameters are ignored.
  - ▶ If any of the arguments have the wrong data type, the function may return unexpected results, depending on the function expression or code.





create a function called to\_meters,  
which converts feet to meters.

► **CREATE FUNCTION to\_meters() { args[0] \* 0.3048 };**

```
SELECT airportname, ROUND(to_meters(geo.alt)) AS mamsl  
FROM `travel-sample`  
WHERE type = "airport"  
LIMIT 5;
```

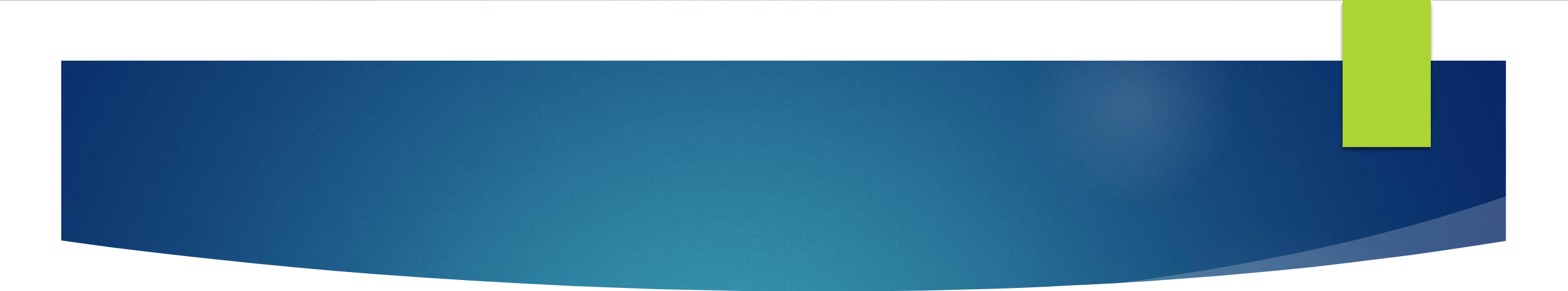
# Inline function with subquery

```
CREATE FUNCTION locations(vType) { (  
  SELECT id, name, address, city  
  FROM `travel-sample`  
  WHERE type = vType) };
```

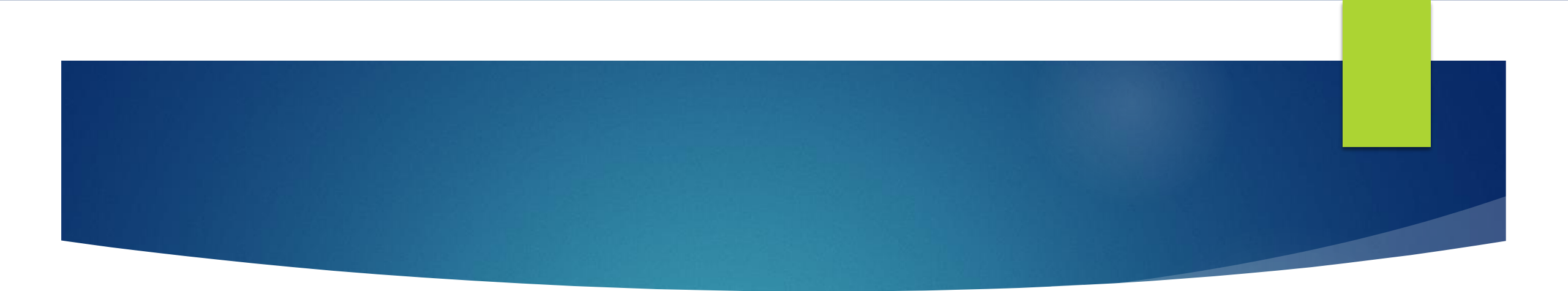
```
SELECT l.name, l.city  
FROM locations("landmark") as l  
WHERE l.city = "Gillingham";
```

# Subqueries

```
SELECT t1.city
FROM `travel-sample` t1
WHERE t1.type = "landmark" AND
      t1.city IN (SELECT RAW city
                  FROM `travel-sample`
                  WHERE type = "airport");
```



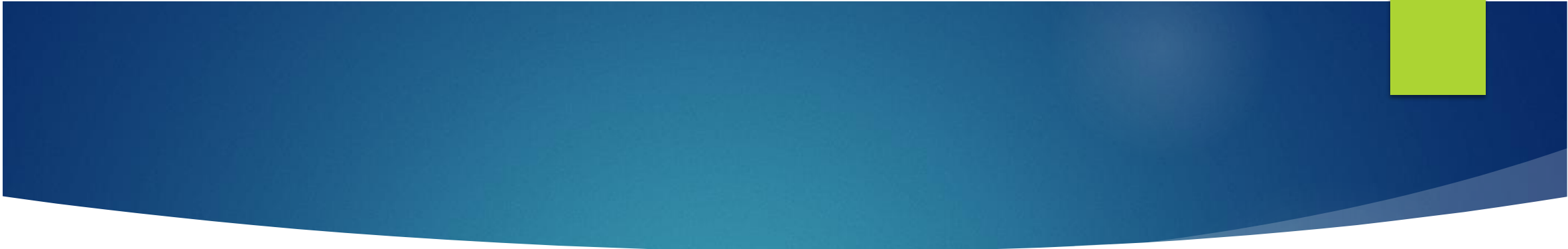
```
SELECT t1.country, array_agg(t1.city), sum(t1.city_cnt) as apnum
FROM (SELECT city, city_cnt, array_agg(airportname) as apnames, country
      FROM `travel-sample` WHERE type = "airport"
      GROUP BY city, country LETTING city_cnt = count(city) ) AS t1
WHERE t1.city_cnt > 5
GROUP BY t1.country;
```



```
SELECT array_max((SELECT count(city) as cnt FROM `travel-  
sample` WHERE type = "airport" GROUP BY city)[*].cnt);
```

subquery is wrapped in a parenthesis ( ) and it occurs as part of the expression `[*].cnt` which treats the result of the subquery as an array of JSON documents (with one attribute `cnt`) and extracts the `cnt` values from all the subquery result array elements.

Finally, the `array_max()` function is applied to this to get the maximum number of airports in any of the cities

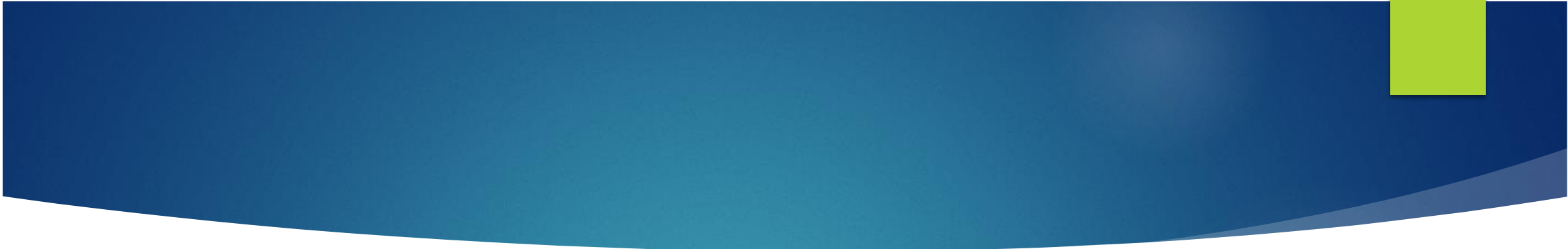
- 
- Find the landmark that are named with corresponding city name.

```
SELECT t1.city, t1.name FROM `travel-sample` t1 WHERE t1.type  
= "landmark" AND t1.city IN SPLIT((SELECT RAW t2.name FROM t1  
AS t2)[0]);
```

- ▶ `SELECT count(*) FROM `travel-sample` t WHERE (SELECT RAW t.geo.alt FROM t t1)[0] > 6000 ;`
- ▶ `SELECT count(*) FROM `travel-sample` t WHERE (SELECT RAW alt FROM t.geo.alt)[0] > 6000;`
- ▶ `SELECT count(*) FROM `travel-sample` t LET x = t.geo WHERE (SELECT RAW y.alt FROM x y)[0] > 6000;`
- ▶ `SELECT count(*) FROM `travel-sample` t WHERE (SELECT RAW geo.alt FROM t.geo)[0] > 6000;`

- ▶ `SELECT array_length((SELECT RAW t1.geo.alt FROM `travel-sample` t1)) FROM `travel-sample` LIMIT 4;`
  - ▶ `SELECT array_length((SELECT RAW t1.geo.alt FROM t t1)) FROM `travel-sample` t;`
  - ▶ `SELECT array_length((SELECT RAW t1.geo.alt FROM `travel-sample`)) FROM `travel-sample`;`
- ```
[ { "code": 4020, "msg": "Duplicate subquery alias travel-sample", "query_from_user": "SELECT array_length((SELECT RAW t1.geo.alt \nFROM `travel-sample` ))\nFROM `travel-sample`;" } ]
```



- 
- ▶ `SELECT * FROM `travel-sample` t1 WHERE t1.type = "landmark" AND t1.city IN (SELECT RAW city FROM `beer-sample` WHERE type = "brewery");`
  - ▶ `SELECT x.alt FROM (SELECT geo from `travel-sample` WHERE type = "airport")[*].geo AS x LIMIT 2;`

# Find airports that are at altitudes more than 4000ft

- ▶ `SELECT t1.city, t1.geo.alt FROM `travel-sample` t1 WHERE t1.type = "airport" AND (SELECT RAW t2.alt FROM `travel-sample`.geo t2)[0] > 4000;`
- ▶ `[ { "code": 3000, "msg": "Ambiguous reference to field travel-sample.", "query_from_user": "SELECT t1.city, t1.geo.alt\nFROM `travel-sample` t1\nWHERE t1.type = \"airport\" AND \n(SELECT RAW t2.alt \n FROM `travel-sample`.geo t2)[0] > 4000;" } ]`

# Find airports that are at altitudes more than 4000ft

► `SELECT t1.city, t1.geo.alt FROM `travel-sample` t1 WHERE  
t1.type = "airport" AND (SELECT RAW t2.alt FROM t1.geo  
t2)[0] > 4000;`



Find the top 10 hotels and number of reviewers which have Overall rating at least 4 and rated by minimum 6 people.

► `SELECT name, cnt_reviewers FROM `travel-sample` AS t LET  
cnt_reviewers = (SELECT raw count(*) FROM t.reviews AS s  
WHERE s.ratings.Overall >= 4)[0] WHERE type = "hotel" and  
cnt_reviewers >= 6 ORDER BY cnt_reviewers DESC LIMIT 10;`

# find the top 3 overall rated hotels.

- ▶ `SELECT name, (SELECT raw avg(s.ratings.Overall) FROM t.reviews as s)[0] AS overall_avg_rating FROM `travel-sample` AS t WHERE type = "hotel" ORDER BY overall_avg_rating DESC LIMIT 3;`
- ▶ example of aggregating using correlated subquery expression in projection

# Inner Join

- List the source airports and airlines that fly into SFO, where only the non-null route documents join with matching airline documents.

```
SELECT route.airlineid, airline.name, route.sourceairport, route.destinationairport
FROM `travel-sample` route
INNER JOIN `travel-sample` airline
ON route.airlineid = META(airline).id
WHERE route.type = "route"
AND route.destinationairport = "SFO"
ORDER BY route.sourceairport;
```

# Left Outer Join

- ▶ Left Outer Join of U.S. airports in the same city as a landmark
- ▶ List the airports and landmarks in the same city, ordered by the airports.

```
SELECT DISTINCT MIN(aport.airportname) AS Airport__Name,
```

```
                MIN(lmark.name) AS Landmark_Name,
```

```
                MIN(aport.tz) AS Landmark_Time
```

```
FROM `travel-sample` aport
```

```
LEFT JOIN `travel-sample` lmark
```

```
  ON aport.city = lmark.city
```

```
  AND lmark.country = "United States"
```

```
  AND lmark.type = "landmark"
```

```
WHERE aport.type = "airport"
```

```
GROUP BY lmark.name
```

```
ORDER BY lmark.name;
```

# Examples



