Getting Started with Optic

MarkLogic 11

Publication date 2024-04-24

Table of Contents

1. What Is Optic?

Optic is the primary MarkLogic language for querying and updating your data. It is an API that uses a flexible pipeline to access any model in your database then to process it through one SQL-like operator after another. With Optic, you can do anything from routine, relational-database-like querying to complex quests through text, values, graphs, geospatial regions, and metadata with just a few lines of code without integrating other tools and systems.

2. Why Read This Guide?

Whether you are new to MarkLogic and multi-model databases or a veteran ready to accelerate your next MarkLogic project's development, this guide will get you going.

After reading it, you will be able to

- Query data by applying concepts similar to SQL statements like `SELECT`, `WHERE`, `ORDER BY`, `JOIN`, and `GROUP BY`.
- Query and update data in efficient, scalable ways to transcend the flat world of rows and columns, unleashing true multi-model access, freely mixing graph data, full text, and even geospatial coordinates.

Since the examples and concepts build upon one another, we recommend reading the guide in order rather than skipping sections.

3. In Which Languages Is Optic Available?

- Optic is available in these language bindings:
  - JavaScript
  - XQuery
- It can be accessed directly using the REST API.
- It can be accessed through a limited JavaScript syntax: Optic DSL (Domain-Specific Language).
- It is available in these client APIs:
  - Java
  - Node.js

NOTE
Optic code samples in this Guide are written in JavaScript, which relies on the `/MarkLogic/optic` library. To find the version of this library for your language of choice, refer to that language's MarkLogic documentation.

4. How Does Optic Work?

NOTE
The Optic API Processing Model article provides a deeper dive into how Optic works.

Optic accesses any MarkLogic data model—table, document, or graph—and represents it as one common model for processing the data: a row sequence, which you can think of as a table. From there, according to your query, Optic manipulates the data as if it were rows and columns with familiar concepts like joins, filters, sorts, and selects. Finally, it produces the results as a row sequence for your app to consume.

Here is a SQL query followed by its Optic equivalent:

```
SELECT ContributorUserName, UserReputation, UserLocation
FROM Samplestack.Contributors
WHERE UserReputation > 5000
ORDER BY UserReputation
LIMIT 25 OFFSET 0;
```

```
op.fromView('Samplestack', 'Contributors')                          // FROM
  .select(['ContributorUserName', 'UserReputation', 'UserLocation']) // SELECT
  .where(op.gt(op.col('UserReputation'), 5000))                     // WHERE
  .orderBy('UserReputation')                                        // ORDER BY
  .offsetLimit(0,25)                                                // LIMIT 25 OFFSET 0
  .result();
```

When you create an Optic query or update, you create a pipeline:

Data Accessor Function-->Operator Function(s)--> Executor Function (JavaScript & XQuery)

- With Data Accessor (or Data Access) Functions like `fromView()`, analogous to SQL's `FROM` statement, you tell Optic where to look for data. You use only one data accessor function per query or update (except in ones that by nature require two data sources: joins, unions, intersections, and excepts).
- With the Operator (or ModifyPlan) Function `select()`, analogous to SQL's `SELECT` statement, you select the columns you need from the data source.
- With operator functions like `orderBy()`, `where()`, and `offsetLimit()`, analogous to SQL's `ORDER BY`, `WHERE`, and `LIMIT` + `OFFSET` statements, you tell Optic how you want that data manipulated. You can have many operator functions in a query or update.
- For JavaScript and XQuery, with an Executor (or IteratePlan) Function like `result()`, you execute the pipeline and receive its results. You use only one executor function per query or update.

When the executor function executes the query or update, Optic optimizes the pipeline to minimize memory usage and maximize performance.

So, to satisfy the example query, upon hitting `result()`, Optic uses `fromView()` to project the required data from the correct documents into a row sequence. It then uses the most time- and memory-efficient way to manipulate the row sequence with `select()` reducing the number of columns, `orderBy()` determining the order of rows, `where()` reducing the rows to those containing specified values in specified columns, and `offsetLimit()` restricting the number of resulting rows. Finally, it uses `result()` to produce the processed rows. These rows are ready to be consumed by your application.

More details on how Optic works will unfold as you see how we have built the following queries and updates to explore and change some sample data.

NOTE

If your query or update requires calling built-in MarkLogic functions against every row in a column, turn those functions into Optic Expression Functions (formerly called Value Processing Functions) by using the `op.` prefix.

5. Building In-Memory Tables with Optic

Building an in-memory table for data that needs to be generated at runtime is a simple task with Optic.

We want to build an in-memory table with some Human Resource (HR) data: roles and levels for three positions.

An Optic example like this one builds a table in memory with two columns, `role` and `level`:

```
op.fromLiterals([
  { "role": "Software Engineer", "level": 1},
  { "role": "Team Lead", "level": 2},
  { "role": "Engineering Manager", "level": 3}
])
.result()
```

We used this example to create and retrieve the table:

- The Data Accessor Function `fromLiterals()` constructs a row sequence with provided data.
- The Executor Function `result()` executes the constructor and returns the results as a row sequence.

Here is the 3-row x 2-column result:

```
{
 "level":1,
 "role":"Software Engineer"
}
{
 "level":2,
 "role":"Team Lead"
}
{
 "level":3,
 "role":"Engineering Manager"
}
```

- The rows are in an unspecified order, which could change between executions. You can specify row order with the `orderBy()` operator function.
- The columns are in an unspecified order, which could change between executions. You can specify column order with the `select()` operator function.

6. Building Optic Queries

Producing rows and columns for SQL analytics tools is a popular Optic task: a good place to start, since it will feel most familiar to those coming from a relational database world.

We want to query HR data from documents containing employee data—employee documents—that are already in our database.

One of Optic's Data Accessor Functions, `fromView()`, together with a MarkLogic index-definition document called a TDE (for Template Driven Extraction), allows us to treat our data almost as if it were coming from and going into a relational database table.

The TDE specifies which document set(s) to extract data from. It also defines the view's row columns to pull the extracted data into.

Upon detecting a new TDE, MarkLogic reindexes to create an index for each view in the TDE, populated with the document data for each column defined.

For these examples, we built a TDE to pull data from our employee documents that we had placed in our employee collection, `http://example.com/content/employee`, into a view we called `Profile`: our employee profile.

Here are the parts of our TDE relevant to most of the queries in this section. Other relevant parts will be called out as needed:

```
// Employee TDE

  "template": {
    "description": "Employee Template",
    "context": "/",
    "collections": [
      "https://example.com/content/employee" // Specifying our document collection
    ],
    "rows": [
      {
        "schemaName": "Employee",             // Schema: Employee
        "viewName": "Profile",                // View: Profile
        "viewLayout": "sparse",
        "columns": [                          // Specifying our view's columns
          {
            "name": "GUID",                   // Column 1: GUID
            "scalarType": "string",
            "val": "GUID",                    // GUID's value comes from
            "nullable": true,                 //   document element, GUID
            "invalidValues": "ignore"
          },
// Columns 2 - 6 not shown
          {
            "name": "Surname",                // Column 7: Surname
            "scalarType": "string",
            "val": "Surname",
            "nullable": true,
            "invalidValues": "ignore"
          },
// Columns 8 - 9 not shown
          {
            "name": "State",                  // Column 10: State
            "scalarType": "string",
            "val": "State",
            "nullable": true,
            "invalidValues": "ignore"
          },
// Columns 11 - 20 not shown
          {
            "name": "Department",             // Column 21: Department
            "scalarType": "string",
            "val": "Department",
            "nullable": true,
            "invalidValues": "ignore"
          } // ,
// Columns 22 - 23 not shown
        ]
      }
    ]
  };
```

- We specified that only documents from our employee collection, `http://example.com/content/employee`, are relevant to this TDE.
- We defined one virtual row in this TDE:
  - `schemaName`: We named our schema `Employee`. Using a meaningful `schemaName` lets us create an association among any views from certain types of documents no matter which TDE they are in.

- `viewName`: We named our virtual row, or view, `Profile`, since it includes those properties from our documents that we want in our employee profile.
- `columns[]`: We defined our view's columns from our documents' available properties in the order we needed them for our `Employee Profile` (23 total):
  - The 4 columns relevant to most queries in this section are `GUID` (column 1), `Surname` (column 7), `State` (column 10), and `Department` (column 21).
  - Other columns will be called out as needed.
- We could also have created other views such as `Payroll`, `Benefits`, and `Reviews` associated with our schema, `Employee` , containing different subsets of our documents' properties.

We now have a view to use in `fromView()`:

    fromView('Employee', 'Profile').

So, analogous to the SQL line

    FROM Employee.Profile,

which accesses a particular table in a particular database, the data accessor function

    fromView('Employee', 'Profile')

lets Optic generate the correct row sequence to work with.

We can now build queries using our view.

NOTE
To create your own TDEs, see Template Driven Extraction (TDE) in the *Application Developer's Guide*.

6.1. With a Select Constraint

We want to retrieve the employee IDs and last names of our employees.

An Optic query like this one retrieves certain columns defined in a view:

```
op.fromView('Employee', 'Profile')
  .select(['GUID', 'Surname'])
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a row sequence containing the `GUID` and `Surname` columns, with a row for each employee in our employee document collection. We limited our output to the first 100 results to make sure our query was working as expected before unleashing it upon the entire collection:

- The Data Accessor Function `fromView()` pulls data indexed for the view `Profile` associated with the schema `Employee` into a row sequence of this view's columns.
- The Operator Function `select()` constrains the query to only the specified columns.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.
- Operator functions like `select()`, `offsetLimit()`, and `orderBy()` must come after any data accessor functions and before any executor function within a query.

- You can put operator functions in any order; however, each operator function works on the results of the previous operator function, so different orders define different queries.

Here are rows 1-5 of the 100-row x 2-column result:

```
{
  "Employee.Profile.GUID": "095d4e63-4a1f-4fc1-b694-b681e2aa3ee0",
  "Employee.Profile.Surname": "Morlan"
}
{
  "Employee.Profile.GUID": "f172c249-3f22-4ebb-a29f-aa2b88213d24",
  "Employee.Profile.Surname": "Crider"
}
{
  "Employee.Profile.GUID": "64f1827d-a2bb-40d0-8875-7fc1d03c311b",
  "Employee.Profile.Surname": "Williams"
}
{
  "Employee.Profile.GUID": "c38b7fba-349f-46ff-a210-e329d9f2dbf1",
  "Employee.Profile.Surname": "Inman"
}
{
  "Employee.Profile.GUID": "40c3def5-b544-4611-9a5b-445cb2c4d89b",
  "Employee.Profile.Surname": "Whitfield"
}
```

- This query returned the first 100 results as we specified in `offsetLimit()`.
- The rows are in an unspecified order, which could change between executions. You can specify row order with the `orderBy()` operator function.
- The columns are the ones we specified in `select()`.
- The columns are presented in the order they appear in `select()`.

6.2. With a Where Constraint

We want to retrieve data for all the employees living in California.

An Optic query like this one retrieves a row sequence containing all view columns if a specific column contains a specific value:

```
op.fromView('Employee', 'Profile')
  .where(op.eq(op.col('State'), 'CA'))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a row sequence of all 23 columns defined in our view with a row for each employee whose `State` column is CA. We limited our output to the first 100 results:

- The Data Accessor Function `fromView()` pulls data indexed for the view `Profile` associated with the schema `Employee` into a row sequence of this view's columns.
- The Operator Function `where()` restricts the rows returned to only those that satisfy the given Boolean expression.
- The Operator Function `eq()` is one of many Boolean Expression Functions. It returns TRUE if the result of its argument expressions is equal, FALSE otherwise.
- `col()` identifies the column in its argument.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here are row 1 and part of row 2 of the 100-row x 23-column result:

```
{
  "Employee.Profile.GUID": "64f1827d-a2bb-40d0-8875-7fc1d03c311b",
  "Employee.Profile.HiredDate": "2016-02-27",
  "Employee.Profile.Gender": "male",
  "Employee.Profile.Title": "Mr.",
  "Employee.Profile.GivenName": "Walter",
  "Employee.Profile.MiddleInitial": "S",
  "Employee.Profile.Surname": "Williams",
  "Employee.Profile.StreetAddress": "4201 Freed Drive",
  "Employee.Profile.City": "Stockton",
  "Employee.Profile.State": "CA",                                          // State: CA
  "Employee.Profile.ZipCode": "95202",
  "Employee.Profile.Country": "US",
  "Employee.Profile.EmailAddress": "WalterSWilliams@fleckens.hu",
  "Employee.Profile.TelephoneNumber": "209-766-7233",
  "Employee.Profile.TelephoneCountryCode": "1",
  "Employee.Profile.Birthday": "3/20/69",
  "Employee.Profile.NationalID": "623-98-7762",
  "Employee.Profile.Point": "37.900497,-121.38312",
  "Employee.Profile.BaseSalary": 86446,
  "Employee.Profile.Bonus": 8645,
  "Employee.Profile.Department": "R&D",
  "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
  "Employee.Profile.ManagerGUID": "695fdc37-42f1-4c19-9ba1-c4fe87454041"
}
{
  "Employee.Profile.GUID": "9fc9ac69-1dcb-46c1-9e0b-2f5ca11758d6",
  "Employee.Profile.HiredDate": "2014-06-22",
  "Employee.Profile.Gender": "male",
  "Employee.Profile.Title": "Mr.",
  "Employee.Profile.GivenName": "James",
  "Employee.Profile.MiddleInitial": "M",
  "Employee.Profile.Surname": "Dusek",
  "Employee.Profile.StreetAddress": "4947 Ella Street",
  "Employee.Profile.City": "Concord",
  "Employee.Profile.State": "CA"                                           // State: CA
// ... (other 13 columns)
}
```

- This query returned the first 100 results as we specified in `offsetLimit()`.
- The rows are in an unspecified order, which could change between executions. You can specify row order with the `orderBy()` operator function.
- The only rows returned have a value of CA in the `State` column, as specified by `where()`.
- The columns are presented in the order we defined them in our view.
  - To change the order for this query alone, put all the columns into the `select()` operator function in your desired order.
  - To permanently change the column order, edit your TDE to change their order in your view.
- To restrict the columns returned, use the `select()` operator function.

6.3. With an Ordered Result

We want to retrieve the data in alphabetical order according to employee last name.

An Optic query like this one retrieves a row sequence containing all view columns with rows in ascending order by the value of a specified column:

```
op.fromView('Employee', 'Profile')
  .orderBy(op.asc('Surname'))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a row sequence of all 23 columns defined in our view for each employee with rows in ascending order by `Surname`. We limited our output to the first 100 results:

- The Data Accessor Function `fromView()` pulls data indexed for the view `Profile` associated with the schema `Employee` into a row sequence of this view's columns.
- The Operator Function `orderBy()` sorts its input row sequence into the order specified.
- The Auxiliary Function `asc()` explicitly sorts rows in ascending order by the specified column's value. The default order is ascending. The Auxiliary Function `desc()` sorts in descending order.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here are row 1 and part of row 2 of the 100-row x 23-column result:

```
{
  "Employee.Profile.GUID": "523b80e5-b98b-46d4-b74f-a755e071e05b",
  "Employee.Profile.HiredDate": "2017-12-09",
  "Employee.Profile.Gender": "female",
  "Employee.Profile.Title": "Mrs.",
  "Employee.Profile.GivenName": "Jean",
  "Employee.Profile.MiddleInitial": "T",
  "Employee.Profile.Surname": "Abbey",                              // Surname: Abbey
  "Employee.Profile.StreetAddress": "1878 School Street",
  "Employee.Profile.City": "Wilton",
  "Employee.Profile.State": "CT",
  "Employee.Profile.ZipCode": "6897",
  "Employee.Profile.Country": "US",
  "Employee.Profile.EmailAddress": "JeanTAbbey@rhyta.com",
  "Employee.Profile.TelephoneNumber": "203-761-3316",
  "Employee.Profile.TelephoneCountryCode": "1",
  "Employee.Profile.Birthday": "2/15/62",
  "Employee.Profile.NationalID": "041-03-4433",
  "Employee.Profile.Point": "41.232025,-73.441277",
  "Employee.Profile.BaseSalary": 72334,
  "Employee.Profile.Bonus": 7233,
  "Employee.Profile.Department": "Marketing",
  "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
  "Employee.Profile.ManagerGUID": "d0862e5d-5be6-473b-a500-3e3a852b2bb8"
}
{
  "Employee.Profile.GUID": "71ea8757-e12e-45a9-bd42-a576b431812e",
  "Employee.Profile.HiredDate": "2012-08-19",
  "Employee.Profile.Gender": "female",
  "Employee.Profile.Title": "Mrs.",
  "Employee.Profile.GivenName": "Donna",
  "Employee.Profile.MiddleInitial": "J",
  "Employee.Profile.Surname": "Acevedo"                             // Surname: Acevedo
// ... (other 16 columns)
}
```

- This query returned the first 100 results as we specified in `offsetLimit()`.
- The rows are in alphabetical order according to the `Surname` column.
- The columns are presented in the order we defined them in our view.
  - To change the order for this query alone, put all the columns into the `select()` operator function in your desired order.
  - To permanently change the column order, edit your TDE to change their order in your view.
- To restrict the columns returned, use the `select()` operator function.

6.4. With a Date Constraint

We want to retrieve all the data for employees hired on or after January 1, 2022. We want the data in order from most to least recent hire.

When we built our view, we set the `scalarType` of the column `HiredDate` to `date` so we could treat it as a date value—

```
{
  "name": "HiredDate",        // Column 2
  "scalarType": "date",       // Date is type "date" in view
  "val": "HiredDate",
  "nullable": true,
  "invalidValues": "ignore"
}
```

—even though the `HiredDate` property in our document is a string:

```
{
  "HiredDate": "2012-09-15"  // Date is type "string" in document
}
```

An Optic query like this one retrieves a row sequence containing all view columns and any rows with a specified column containing a value equal to or greater than a specified value in descending order by that column:

```
op.fromView('Employee', 'Profile')
  .where(op.ge(op.col('HiredDate'), xs.date('2022-01-01')))
  .orderBy(op.desc('HiredDate'))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a row sequence with all 23 columns defined in our view for each employee with a `HiredDate` greater than or equal to `2022-01-01`, with rows in descending order by `HiredDate`. We limited our output to the first 100 results:

- The Data Accessor Function `fromView()` pulls data indexed for the view `Profile` associated with the schema `Employee` into a row sequence of this view's columns.
- The Operator Function `where()` restricts the rows returned to only those that satisfy the given Boolean expression.
- The Operator Function `ge()` is one of many Boolean Expression Functions. It returns `TRUE` if the result of its first argument expression is greater than or equal to the second, `FALSE` otherwise.
- `col()` identifies the column in its argument.
- `xs.date()` is the MarkLogic standard function for converting a string into a date value, the same data type as `HiredDate`.
- The Operator Function `orderBy()` sorts its input row sequence into the order specified.
- The Auxiliary Function `desc()` sorts rows in descending order by the specified column's value. The default order is ascending. The Auxiliary Function `asc()` explicitly sorts in ascending order.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here are row 1, row 2, and part of row 3 of the 85-row x 23-column result:

```
{
  "Employee.Profile.GUID": "87e92c81-f014-44e3-821e-89e25302ec33",
  "Employee.Profile.HiredDate": "2022-11-07",                    // HiredDate: 2022-11-07
  "Employee.Profile.Gender": "male",
  "Employee.Profile.Title": "Mr.",
  "Employee.Profile.GivenName": "Jack",
  "Employee.Profile.MiddleInitial": "V",
  "Employee.Profile.Surname": "Lee",
  "Employee.Profile.StreetAddress": "67 Smith Road",
  "Employee.Profile.City": "Cumming",
  "Employee.Profile.State": "GA",
  "Employee.Profile.ZipCode": "30130",
  "Employee.Profile.Country": "US",
  "Employee.Profile.EmailAddress": "JackVLee@jourrapide.com",
  "Employee.Profile.TelephoneNumber": "770-887-9223",
  "Employee.Profile.TelephoneCountryCode": "1",
  "Employee.Profile.Birthday": "9/22/90",
  "Employee.Profile.NationalID": "252-74-9314",
  "Employee.Profile.Point": "34.106709,-84.226639",
  "Employee.Profile.BaseSalary": 50814,
  "Employee.Profile.Bonus": 5081,
  "Employee.Profile.Department": "Sales",
  "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
  "Employee.Profile.ManagerGUID": "07bb968f-e95e-4c40-8889-d88ba6b369d0"
}
{
  "Employee.Profile.GUID": "33457955-6697-43d6-81f4-b1c12218fa01",
  "Employee.Profile.HiredDate": "2022-11-03",                    // HiredDate: 2022-11-03
  "Employee.Profile.Gender": "female",
  "Employee.Profile.Title": "Mrs.",
  "Employee.Profile.GivenName": "Patricia",
  "Employee.Profile.MiddleInitial": "M",
  "Employee.Profile.Surname": "Diaz",
  "Employee.Profile.StreetAddress": "4536 Norma Avenue",
  "Employee.Profile.City": "Marysville",
  "Employee.Profile.State": "OH",
  "Employee.Profile.ZipCode": "43040",
  "Employee.Profile.Country": "US",
  "Employee.Profile.EmailAddress": "PatriciaMDiaz@armyspy.com",
  "Employee.Profile.TelephoneNumber": "937-209-8542",
  "Employee.Profile.TelephoneCountryCode": "1",
  "Employee.Profile.Birthday": "8/7/62",
  "Employee.Profile.NationalID": "297-28-8599",
  "Employee.Profile.Point": "40.172447,-83.289566",
  "Employee.Profile.BaseSalary": 49297,
  "Employee.Profile.Bonus": 4930,
  "Employee.Profile.Department": "R&D",
  "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
  "Employee.Profile.ManagerGUID": "695fdc37-42f1-4c19-9ba1-c4fe87454041"
}
{
  "Employee.Profile.GUID": "8b7f98f0-60f2-4e40-bd01-bb0e2b30d99e",
  "Employee.Profile.HiredDate": "2022-11-01"                     // HiredDate: 2022-11-01
// ... (other 21 columns)
}
```

- This query returned 85 results, fewer than the 100 we specified in our `offsetLimit()`. Therefore, there were only 85 employees hired on or after January 1, 2022.
- The rows are in descending order according to the `HiredDate` column.

- The columns are presented in the order we defined them in our view.
  - To change the order for this query alone, put all the columns into the `select()` operator function in your desired order.
  - To permanently change the column order, edit your TDE to change their order in your view.
- To restrict the columns returned, use the `select()` operator function.

6.5. With a Grouped Result

We want to see what departments we have and how many employees are in each one.

An Optic query like this one counts how many view rows contain each unique value in a specified column. It returns a row sequence containing a row for each unique value in that column and the count of rows where that column equals that unique value in descending order by that count:

```
op.fromView('Employee', 'Profile')
  .groupBy('Department', [op.count('DepartmentCount', 'Department')])
  .orderBy(op.desc('DepartmentCount'))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a row sequence containing a `Department` column and a `DepartmentCount` column for each unique department with rows in descending order by `DepartmentCount`:

- The Data Accessor Function `fromView()` pulls data indexed for the view `Profile` associated with the schema `Employee` into a row sequence of this view's columns.
- The Operator Function `groupBy()` specifies `Department` as the column to group by and an Aggregate Function to apply to each resulting group.
- The Operator Function `count()` is one of many Aggregate Functions. It counts the number of rows with a certain `Department` and stores this count in `DepartmentCount`.
- The Operator Function `orderBy()` sorts its input row sequence into the order specified.
- The Auxiliary Function `desc()` sorts rows in descending order by the specified column's value. The default order is ascending. The Auxiliary Function `asc()` explicitly sorts in ascending order.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here is the 5-row x 2-column result:

```
{
  "Department": "R&D",
  "DepartmentCount": 408
}
{
  "Department": "Sales",
  "DepartmentCount": 206
}
{
  "Department": "Engineering",
  "DepartmentCount": 183
}
{
  "Department": "Marketing",
  "DepartmentCount": 174
}
{
  "Department": "Training",
  "DepartmentCount": 29
}
```

- This query returned 5 rows, fewer than the 100 we specified in our `offsetLimit()`. Therefore, there were only 5 different departments.
- Each row represents a unique department.
- The rows are ordered from the largest to the smallest department, as determined by `DepartmentCount`'s value.
- The columns are the ones `groupBy()` created, displayed in the order they appear in that function.

6.6. With a Join against Another View

We want to join data from our employee documents with data from documents containing department data—department documents—also already in our database.

So, we created a second view for our department documents in a separate TDE:

```
{
  "template": {
    "description": "Department Template",
    "context": "/",
    "collections": [
      "https://example.com/content/department" // Specifying our document collection
    ],
    "rows": [
      {
        "schemaName": "Department",            // Schema: Department
        "viewName": "Profile",                 // View: Profile
        "viewLayout": "sparse",
        "columns": [                           // Specifying our view's columns
          {
            "name": "Department",              // Column 1: Department
            "scalarType": "string",
            "val": "Department",
            "nullable": true,
            "invalidValues": "ignore"
          },
          {
            "name": "LineOfBusiness",
            "scalarType": "string",
            "val": "LineOfBusiness",
            "nullable": true,
            "invalidValues": "ignore"
          },
          {
            "name": "Description",
            "scalarType": "string",
            "val": "Description",
            "nullable": true,
            "invalidValues": "ignore"
          }
        ]
      }
    ]
  }
}
```

- We specified that only documents from our department collection, `http://example.com/content/department`, are relevant to this template.
- We named our schema `Department`.
- We also named this view `Profile`. We can use the same view name in different schemas.
- This view has only 3 columns: `Department` (also a column in our employee view), `LineOfBusiness`, and `Description`.

With two views, we can now do a join.

An Optic query like this one performs an inner join of two views on a matching column, with one view acting as the “left table”, the other as the “right table”:

```
const employees = op.fromView('Employee', 'Profile');
const department = op.fromView('Department', 'Profile');

employees
  .joinInner(department, op.on(employees.col('Department'), department.col('Department')))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a row sequence with all columns from both views for employee documents where there is a department document with a matching `Department` column:

- The Data Accessor Function `fromView` (`‘Employee’, ‘Profile’`) pulls data indexed for the view `Profile` associated with the schema `Employee` into a row sequence of this view’s columns.
- The Data Accessor Function `fromView` (`‘Department’, ‘Profile’`) pulls data indexed for the view `Profile` associated with the schema `Department` into a row sequence of this view’s columns.
- When you use more than one Data Accessor Function in a query, it is best practice to use a variable to represent each one to keep the query simple and readable. This practice also lets you use Expression Functions like `employees.col('Department')` to reference a view's columns.
- The Operator Function `joinInner` returns all rows from both views if the specified columns match, considering the view specified first in the query to be the "left table" and the view specified second to be the "right table."
- The Auxiliary Function `on()` lets you specify a join condition for join-type operator functions like `joinInner()`.
- `col()` identifies the column in its argument.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here is row 1 of the 100-row x 26-column result:

```
{
  "Employee.Profile.GUID": "095d4e63-4a1f-4fc1-b694-b681e2aa3ee0", // EC1
  "Department.Profile.Department": "Engineering",                  //  DC1 = EC21
  "Employee.Profile.HiredDate": "2019-06-12",                      // EC2
  "Department.Profile.LineOfBusiness": "Information Technology",   //  DC2
  "Employee.Profile.Gender": "male",                               // EC3
  "Department.Profile.Description": "Engineering involves ...",    //  DC3
  "Employee.Profile.Title": "Mr.",                                 // EC4
  "Employee.Profile.GivenName": "Elmer",                           // EC5, etc.
  "Employee.Profile.MiddleInitial": "H",
  "Employee.Profile.Surname": "Morlan",
  "Employee.Profile.StreetAddress": "3084 Bolman Court",
  "Employee.Profile.City": "Springfield",
  "Employee.Profile.State": "IL",
  "Employee.Profile.ZipCode": "62701",
  "Employee.Profile.Country": "US",
  "Employee.Profile.EmailAddress": "ElmerHMorlan@rhyta.com",
  "Employee.Profile.TelephoneNumber": "217-301-0206",
  "Employee.Profile.TelephoneCountryCode": "1",
  "Employee.Profile.Birthday": "7/31/37",
  "Employee.Profile.NationalID": "333-82-1925",
  "Employee.Profile.Point": "39.747955,-89.709328",
  "Employee.Profile.BaseSalary": 47744,
  "Employee.Profile.Bonus": 4774,
  "Employee.Profile.Department": "Engineering",                    // EC21 = DC1
  "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
  "Employee.Profile.ManagerGUID": "3ad0ffbc-3ade-4897-902b-718417a721f5"
}
```

- This query returned the first 100 results as we specified in `offsetLimit()`.
- The rows are in an unspecified order, which could change between executions. You can specify row order with the `orderBy()` operator function.
- The 26 columns are the 23 from our "left table," the `Employee Profile` view (`EC#`) plus the 3 from our "right table," the `Department Profile` view (`DC#`).
- The columns are presented in the order we defined them in our view.
  - To change the order for this query alone, put all the columns into the `select()` operator function in your desired order.
  - To permanently change the column order, edit your TDE to change their order in your view.
- To restrict the columns returned, use the `select()` operator function.

7. Building Queries with Search Capabilities

Now that you have seen how to write simple Optic queries for routine exploration of data, you are ready to see how to enhance them with search capabilities.

Optic supports the use of the MarkLogic core search API, Core Text Search (CTS), as a filter. With CTS queries, you can easily add powerful and specific document-level searches to your Optic queries. See the *Search Developer's Guide* for more detailed information than we provide here.

CTS queries apply to the data indexed from the documents themselves rather than just to the data indexed from columns extracted from those documents. So, for example, using a CTS query with the Data Accessor Function `fromView()` could return a row sequence which itself contains no matching data if the original document's metadata contained the match. To further understand how indexing affects CTS searches, see Indexing in MarkLogic in the *Concepts Guide*.

NOTE
Optic searches are unfiltered.

7.1. To Find a Phrase within a View

We want to find all full-time employees. We know that "Full-Time" or "full time"—or something like that—is somewhere in our document structure.

An Optic query like this one finds a case-, punctuation-, and whitespace-insensitive phrase that could be anywhere in any document in the view's context—not just in the view's column data:

```
op.fromView('Employee', 'Profile')
  .where(cts.wordQuery('full time'))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a row sequence with all our employee-view columns from any employee-collection document containing some form of the phrase "full time" anywhere within it, limited to 100 results:

- The Data Accessor Function `fromView()` pulls data indexed for the view `Profile` associated with the schema `Employee` into a row sequence of this view's columns.
- The Operator Function `where()` restricts the rows returned to only those that satisfy the given Boolean expression.
- The CTS (Core Text Search) Function `cts.wordQuery()` finds the word or phrase provided in the first parameter:
  - With the phrase containing no uppercase letters and with no other parameters, the search defaults to finding variations with any case letters.
  - With the phrase containing no punctuation between the words and with no other parameters, the search defaults to also finding variations with punctuation between words.
  - With the phrase containing a single space between the words and with no other parameters, the search defaults to also finding variations with more than one space between the words.
  - Other `cts.wordQuery()` parameters provide other possibilities.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.

- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here is row 1 of the 100-row x 23-column result:

```
{
  "Employee.Profile.GUID": "095d4e63-4a1f-4fc1-b694-b681e2aa3ee0",
  "Employee.Profile.HiredDate": "2019-06-12",
  "Employee.Profile.Gender": "male",
  "Employee.Profile.Title": "Mr.",
  "Employee.Profile.GivenName": "Elmer",
  "Employee.Profile.MiddleInitial": "H",
  "Employee.Profile.Surname": "Morlan",
  "Employee.Profile.StreetAddress": "3084 Bolman Court",
  "Employee.Profile.City": "Springfield",
  "Employee.Profile.State": "IL",
  "Employee.Profile.ZipCode": "62701",
  "Employee.Profile.Country": "US",
  "Employee.Profile.EmailAddress": "ElmerHMorlan@rhyta.com",
  "Employee.Profile.TelephoneNumber": "217-301-0206",
  "Employee.Profile.TelephoneCountryCode": "1",
  "Employee.Profile.Birthday": "7/31/37",
  "Employee.Profile.NationalID": "333-82-1925",
  "Employee.Profile.Point": "39.747955,-89.709328",
  "Employee.Profile.BaseSalary": 47744,
  "Employee.Profile.Bonus": 4774,
  "Employee.Profile.Department": "Engineering",
  "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",        // Found!
  "Employee.Profile.ManagerGUID": "3ad0ffbc-3ade-4897-902b-718417a721f5"
}
```

- This query returned the first 100 results as we specified in `offsetLimit()`.
- This query also could have found “Full-Time”, "Full Time", "FULL-TIME", "Full - time", and other variations.
- This query would NOT have found "fulltime" (or other case variations) because "fulltime" is indexed as one word while "full" and "time", separated by space and/or punctuation, are each indexed as separate words.
- Only one result will be returned per document no matter how many times the phrase occurs within a particular document.

7.2. To Find Documents in a Collection

In a MarkLogic multi-model database, your data does not have to be neat and highly structured to explore it. So, we could have explored our HR data using raw documents without creating a view.

The Data Accessor Function `fromSearchDocs()` is the Optic equivalent of `cts.search()`. It returns documents based on the CTS query provided along with their URI and score.

An Optic query like this one retrieves all of the data from 100 unique documents in a specified collection:

```
op.fromSearchDocs(cts.collectionQuery('https://example.com/content/employee'))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a 3-column row sequence of all data from each employee-collection document, along with each document's URI and score, limited to 100 results:

- The Data Accessor Function `fromSearchDocs()` pulls data from documents matching the `cts.collectionQuery()` parameter and narrowed down by other parameters into a row sequence with a unique row of these 3 columns for each matching document:
  - `uri`: Contains the document URI.
  - `doc`: Contains the document itself.
  - `score`: Contains the document's search score, a measure of how relevant this result is with respect to other results. The higher the score, the higher the relevance.
- The CTS Function `cts.collectionQuery()` restricts `fromSearchDocs()` to returning only data that matches the `cts.collectionQuery()`. In this case, `fromSearchDocs()` will only access documents in the specified collection.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here is row 1 of the 100-row x 3-column result:

```
{
 "uri": "/data/employees/c1f3450c-cf3b-4622-8df7-a2f0818ada72.json",
 "doc": {
  "GUID": "c1f3450c-cf3b-4622-8df7-a2f0818ada72",
  "Gender": "male",
  "Title": "Mr.",
  "GivenName": "Ralph",
  "MiddleInitial": "J",
  "Surname": "Garcia",
  "StreetAddress": "209 Stratford Park",
  "City": "Crane",
  "State": "IN",
  "ZipCode": "47522",
  "Country": "US",
  "EmailAddress": "RalphJGarcia@teleworm.us",
  "TelephoneNumber": "812-854-1074",
  "TelephoneCountryCode": "1",
  "Birthday": "1/2/78",
  "NationalID": "310-48-6699",
  "BaseSalary": "78730",
  "Bonus": "7873",
  "Department": "R&D",
  "Status": "Active - Regular Exempt (Full-time)",
  "ManagerGUID": "695fdc37-42f1-4c19-9ba1-c4fe87454041",
  "point": {
   "lat": 38.823173,
   "long": -86.881793
  },
  "HiredDate": "2012-01-03"
 },
 "score": 0
}
```

- This query returned the first 100 results as we specified in `offsetLimit()`.
- Note that the properties in the document returned are in a different order than the columns in our view that we created for this collection of documents. Building a view lets you put any of the data that you want from documents in a collection into any order you want without affecting the documents themselves.

7.3. To Find a Phrase within Documents in a Collection

So, using `fromSearchDocs()`, we could have found our full-time employees by diving straight into our documents.

An Optic query like this one finds a case-, punctuation-, and whitespace-insensitive phrase anywhere within a document of a specified collection:

```
op.fromSearchDocs(
  cts.andQuery([
    cts.collectionQuery('https://example.com/content/employee'),
    cts.wordQuery('full time')
  ]))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a 3-column row sequence of all data from each employee-collection document containing some form of the phrase "full time", along with each document's URI and score, limited to 100 results:

- The Data Accessor Function `fromSearchDocs()` pulls data from documents matching the `cts.collectionQuery()` parameter and narrowed down by other parameters into a row sequence with a unique row of these 3 columns for each matching document:
  - `uri`: Contains the document URI.
  - `doc`: Contains the document itself.
  - `score`: Contains the document’s search score, a measure of how relevant this result is with respect to other results. The higher the score, the higher the relevance.
- The CTS Function `cts.andQuery()` returns the intersection of documents matching each of its CTS-type parameters:
  - Its first parameter, `cts.collectionQuery()`, finds all the data from documents in the specified collection: our employee collection.
  - Its second parameter, `cts.wordQuery()`, finds the word or phrase provided: `full time`.
  - So, `cts.andQuery()` returns all the data from documents from our employee collection that contain the phrase `full time`.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here is row 1 of the 100-row x 3-column result:

```
{
 "uri": "/data/employees/5899d871-1261-4057-ab3e-7fea1577ba61.json",
 "doc": {
  "GUID": "5899d871-1261-4057-ab3e-7fea1577ba61",
  "Gender": "male",
  "Title": "Mr.",
  "GivenName": "Scott",
  "MiddleInitial": "M",
  "Surname": "Schaaf",
  "StreetAddress": "3586 Paradise Lane",
  "City": "Pomona",
  "State": "CA",
  "ZipCode": "91766",
  "Country": "US",
  "EmailAddress": "ScottMSchaaf@rhyta.com",
  "TelephoneNumber": "909-629-3047",
  "TelephoneCountryCode": "1",
  "Birthday": "9/25/45",
  "NationalID": "561-42-6126",
  "BaseSalary": "79460",
  "Bonus": "7946",
  "Department": "Engineering",
  "Status": "Active - Regular Exempt (Full-time)",               // Found!
  "ManagerGUID": "3ad0ffbc-3ade-4897-902b-718417a721f5",
  "point": {
   "lat": 34.014225,
   "long": -117.843894
  },
  "HiredDate": "2021-11-19"
 },
 "score": 2048
}
```

- This query returned the first 100 results as we specified in `offsetLimit()`.
- Only one result will be returned per document no matter how many times the phrase occurs within a particular document.
- A common practice is to add `orderBy(op.desc(score))` to order by score from most to least relevant result.

7.4. To Find Phrases Near Each Other

We want to find only active, full-time employees. We know “active” and “full time” are near each other in the employee collection document `status` property. But a simple `AND`-type function would give us false matches if “active” were not near “full time”. We need to eliminate that issue.

An Optic query like this one finds two specified phrases in any order within 10 words of each other anywhere within a document in a specified collection. It is similar to the previous query except that it uses `cts.nearQuery()` with the two phrases as separate parameters instead of `cts.wordQuery()`, with its one phrase as a single parameter:

```
op.fromSearchDocs(
   cts.andQuery([
     cts.collectionQuery('https://example.com/content/employee'),
     cts.nearQuery(['active', 'full time'])
  ]))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a 3-column row sequence of all data from each employee-collection document containing some form of the phrase "full time" 10 words or fewer from the word "active", along with each document's URI and score, limited to 100 results:

- The Data Accessor Function `fromSearchDocs()` pulls data from documents matching the `cts.collectionQuery()` parameter and narrowed down by other parameters into a row sequence with a unique row of these 3 columns for each matching document:
  - `uri`: Contains the document URI.
  - `doc`: Contains the document itself.
  - `score`: Contains the document's search score, a measure of how relevant this result is with respect to other results. The higher the score, the higher the relevance.
- The CTS Function `cts.andQuery()` returns the intersection of matches that each of its parameter functions finds:
  - `cts.collectionQuery()` finds all the data from documents in the specified collection, our employee collection.
  - `cts.nearQuery()` finds the word or phrase provided in the first parameter within 10 words before or after the word or phrase provided in the second parameter:
    - Each phrase parameter has the same defaults and settings as it would in `cts.wordQuery()`.
    - By default, "near" is within 10 words, and the phrases can be found in either order.
    - Other `cts.nearQuery()` parameters provide other possibilities.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here is row 1 of the 3-column x 100-row result:

```
{
 "uri": "/data/employees/cb31aeaa-e708-4034-a77c-ceead02ca644.json",
 "doc": {
  "GUID": "cb31aeaa-e708-4034-a77c-ceead02ca644",
  "Gender": "female",
  "Title": "Mrs.",
  "GivenName": "Elaine",
  "MiddleInitial": "D",
  "Surname": "Perrone",
  "StreetAddress": "1223 Frederick Street",
  "City": "Sacramento",
  "State": "CA",
  "ZipCode": "94260",
  "Country": "US",
  "EmailAddress": "ElaineDPerrone@teleworm.us",
  "TelephoneNumber": "916-230-4803",
  "TelephoneCountryCode": "1",
  "Birthday": "4/1/44",
  "NationalID": "626-03-3604",
  "BaseSalary": "44042",
  "Bonus": "4404",
  "Department": "Engineering",
  "Status": "Active - Regular Exempt (Full-time)",                    // Found!
  "ManagerGUID": "3ad0ffbc-3ade-4897-902b-718417a721f5",
  "point": {
   "lat": 38.574274,
   "long": -121.374583
  },
  "HiredDate": "2018-09-12"
 },
 "score": 4096
}
```

- This query returned the first 100 results as we specified in `offsetLimit()`.
- Using the completely lowercase parameter `active` in `cts.nearQuery()` defaulted the search to case insensitive, so it also found the title-cased instance of "Active".

- With no parameter specifying order in the `cts.nearQuery()`, the search defaults to unordered, so it would have also matched if “Full-time” had come before “Active”.
- Only one result will be returned per document no matter how many times the phrase occurs within a particular document.

7.5. To Include an Operator in the Search String

We could have found our active, full-time employees with a different CTS Function: `cts.parse()`. The following query is similar to the previous query except that it uses `cts.parse()` with a single parameter consisting of both phrases separated by an operator instead of `cts.nearQuery()` with a parameter for each phrase:

```
op.fromSearchDocs(
   cts.andQuery([
      cts.collectionQuery('https://example.com/content/employee'),
      cts.parse('active NEAR full time')
   ]))
   .offsetLimit(0, 100)
   .result();
```

- The CTS Function `cts.parse()` allows you to quickly write complex search queries using a grammar that will be automatically translated into whatever `cts.query`-type function will do the job.
- Creating a search box field in your user-facing application is a common way to enable your users to leverage this grammar.
- `cts.parse(‘active NEAR full time’)` is translated into `cts.nearQuery([‘active’, ‘full time’])`.
- The full grammar for this function is explained in Creating a Query from Search Text with cts:parse in the *Search Developer's Guide*.

Here is row 1 of the 100-row x 3-column result:

```
{
 "uri": "/data/employees/b0d5a15e-b5ce-4138-9a59-f46e08119bc4.json",
 "doc": {
  "GUID": "b0d5a15e-b5ce-4138-9a59-f46e08119bc4",
  "Gender": "male",
  "Title": "Mr.",
  "GivenName": "Ralph",
  "MiddleInitial": "M",
  "Surname": "Deweese",
  "StreetAddress": "2031 O Conner Street",
  "City": "Gulfport",
  "State": "MS",
  "ZipCode": "39507",
  "Country": "US",
  "EmailAddress": "RalphMDeweese@rhyta.com",
  "TelephoneNumber": "228-850-3365",
  "TelephoneCountryCode": "1",
  "Birthday": "11/25/62",
  "NationalID": "428-08-3456",
  "BaseSalary": "75054",
  "Bonus": "7505",
  "Department": "Marketing",
  "Status": "Active - Regular Exempt (Full-time)",              // Found!
  "ManagerGUID": "d0862e5d-5be6-473b-a500-3e3a852b2bb8",
  "point": {
   "lat": 30.372732,
   "long": -88.999739
  },
  "HiredDate": "2020-03-18"
 },
 "score": 4096
}
```

- Only one result will be returned per document no matter how many times the phrase occurs within a particular document.

8. Building Queries with Geospatial Constraints

Geospatial search is a powerful feature of MarkLogic query capabilities that you will want to explore. See Geospatial Search Applications in the *Search Developer's Guide* for more detailed information than we provide here.

We want to find all our employees within a 25-mile radius of a customer located in a tri-state area. Using that customer's specific latitude and longitude and a geospatial query will yield better results than using either the employees' state or ZIP Code to find nearby employees.

So, when we built our employee profile view, we set the `scalarType` of the `Point` column to `point`. Then we set `Point`'s value to a single latitude, longitude coordinate point using `cts:point` to pull them from the individual `lat` and `long` properties of the document:

```
// Snippet from TDE Employee:
{
  "name": "Point",                              // Column 18: Point
  "scalarType": "point",                        // Data Type: point
  "val": "cts:point(point/lat, point/long)", // Values from point property array
  "coordinateSystem": "wgs84",                  // Coordinate System: WGS84
  "nullable": true,
  "invalidValues": "ignore"
}
```

```
// Snippet from employee document:

"point" {                                       // point property array
 "lat": 37.443029,
 "long": -121.720815
}
```

When we created our view column with the `scalarType point`, we also followed best practice of specifying its `coordinateSystem` even though `wgs84` is the default.

Now, we can use the `Point` column in any function requiring latitude and longitude as an x,y coordinate point.

An Optic query like this one retrieves a row sequence containing specified columns and ordered from employee closest to farthest away from a specified geolocation:

```
const location = cts.point(39.7176,-75.9349) // In NE MD near PA & DE borders

op.fromView('Employee', 'Profile')
  .where(
    op.geo.within(
      op.col('Point'),
      geo.circlePolygon(cts.circle(25, location),0.5)
    )
  )
  .bind(op.as('Distance', op.geo.distance(op.col('Point'), location)))
  .select(['GUID', 'Surname', 'State', 'Point', 'Distance'])
  .orderBy(op.col('Distance'))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a row sequence for each employee within a 25-mile radius of our customer's location, ordered from closest to farthest away. Each row contains the `GUID`, `Surname`, `State`, and `Point` columns as well as `Distance`, a column calculated within the query to contain how far each employee is from our customer:

- The Data Accessor Function `fromView()` pulls data indexed for the view `Profile` associated with the schema `Employee` into a row sequence of this view's columns.
- The Expression Function `op.geo.within()` is one of many geospatial expression functions. It returns `TRUE` if the region specified in the first parameter is within the region specified by the second parameter:
  - Its first parameter provides the first region with `col(Point)`: `Point`'s coordinates.
  - Its second parameter provides the second region: a polygon approximating a geospatial circle that `geo.circlePolygon()` determines.
  - So, `op.geo.within()` returns `TRUE` for every `Point` within a 25-mile radius of our customer's `location`.
- The Operator Function `bind()` uses `as()` to define a new column and sets its value for each row in turn.
- The Expression Function `op.geo.distance()` calculates the radial distance between two points.
- The Operator Function `select()` constrains the query to only the specified columns.
- The Operator Function `orderBy()` sorts its input row sequence into the order specified.
- The Auxiliary Function `asc()` explicitly sorts rows in ascending order by the specified column's value. The default order is ascending. The Auxiliary Function `desc()` sorts in descending order.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here is the 4-row x 5-column result:

```
{
  "Employee.Profile.GUID": "b7d04aa0-8348-4794-b9a6-a2f2723f3ffe",
  "Employee.Profile.Surname": "English",
  "Employee.Profile.State": "DE",
  "Employee.Profile.Point": "39.761456,-75.803268",
  "Distance": 7.63593344994741},
}
{
  "Employee.Profile.GUID": "8de256d2-b6ba-4b3f-9b2e-2e0f65fa9025",
  "Employee.Profile.Surname": "Desantis",
  "Employee.Profile.State": "PA",
  "Employee.Profile.Point": "39.68557,-75.559578",
  "Distance": 20.1233662009832},
}
{
  "Employee.Profile.GUID": "1c208e16-5350-4e4e-83eb-24ccc72bdabd",
  "Employee.Profile.Surname": "McCray",
  "Employee.Profile.State": "MD",
  "Employee.Profile.Point": "39.622498,-76.300697",
  "Distance": 20.5768543237538},
}
{
  "Employee.Profile.GUID": "97f4c500-b158-4e92-9cdb-f69e24bc2c57",
  "Employee.Profile.Surname": "Barlow",
  "Employee.Profile.State": "PA",
  "Employee.Profile.Point": "39.738686,-76.396965",
  "Distance": 24.6574870655178}
}
```

- The columns are the ones we specified in `select()`.
- The columns are presented in the order they appear in `select()`.
- Employees from 3 different states are among the results.
- We could pass this result to a client application that plots points onto a map.

9. Building Results-with-Facets Queries

Faceting results is a powerful feature of MarkLogic search capabilities that you will want to explore. Facets enrich search user interfaces by effectively generating metadata about the query, helping your end users to explore data more efficiently. See Including Facets in Search Results in the *Search Developer's Guide* for more detailed information than we provide here.

Faceting results amounts to defining a query to evaluate for results then reevaluating it to discover how many results fall into specified categories.

We want to retrieve all the data for employees hired on or after January 1, 2022 as in With a Date Constraint. But we also want to see how these employees are distributed across departments and/or states.

An Optic query like this one creates an output object containing a result with two facets, `Departments` and `States`:

```
const op = require('/MarkLogic/optic');

// Reuse the same constrained query for multiple tasks:
const query = op.fromView('Employee', 'Profile')
              .where(op.ge(op.col('HiredDate'), xs.date('2022-01-01')))

// Fetch the view results:
const results = query.offsetLimit(0, 100).result();

// Fetch facets for quick aggregations:
const facets =
    query.facetBy([
      op.namedGroup('Departments', 'Department'),
      op.namedGroup('States', 'State')
    ])
    .offsetLimit(0, 100)
    .result()

// Build an Output Object:
const output = {
  'results': results,
  'facets': facets
}

output
```

We used this query to produce a results array containing up to 100 employees hired on or after January 1, 2022 and a facets object containing the department and state facet arrays.

There are several sections to this query:

- The part of the original query from With a Date Constraint that finds employees hired on or after a certain date, stored as `query`.
- The result of that query, limited to 100 results, stored as `results`.
- The generation of the facets, stored as `facets`, explained in more detail below.
- The building of the output object containing both the results and the facets, stored as `output`.

Here are the details of the faceting functions:

- The Operator Function `facetBy()` takes as many `namedGroup()` functions as needed, creating a facet for each one.

- The Operator Function `namedGroup()` defines an array in the first parameter to hold counts for each unique value encountered in the column specified in the second parameter.

There are several parts to the result:

- A 23-row x 85-column row sequence called `results` containing all 23 view columns for each of the 85 matching employees. Here are the first, second, and last rows:

```
{
  "results": [
    {
      "Employee.Profile.GUID": "86148084-f46f-46ba-a5b7-24748629a611",
      "Employee.Profile.HiredDate": "2022-06-07",              // HiredDate
      "Employee.Profile.Gender": "male",
      "Employee.Profile.Title": "Mr.",
      "Employee.Profile.GivenName": "Michael",
      "Employee.Profile.MiddleInitial": "S",
      "Employee.Profile.Surname": "Dandridge",
      "Employee.Profile.StreetAddress": "1012 Union Street",
      "Employee.Profile.City": "Seattle",
      "Employee.Profile.State": "WA",
      "Employee.Profile.ZipCode": "98107",
      "Employee.Profile.Country": "US",
      "Employee.Profile.EmailAddress": "MichaelSDandridge@superrito.com",
      "Employee.Profile.TelephoneNumber": "206-784-1532",
      "Employee.Profile.TelephoneCountryCode": "1",
      "Employee.Profile.Birthday": "2/11/61",
      "Employee.Profile.NationalID": "535-09-2679",
      "Employee.Profile.Point": "47.675446,-122.41084",
      "Employee.Profile.BaseSalary": 79072,
      "Employee.Profile.Bonus": 7907,
      "Employee.Profile.Department": "R&D",
      "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
      "Employee.Profile.ManagerGUID": "695fdc37-42f1-4c19-9ba1-c4fe87454041"
    },
    {
      "Employee.Profile.GUID": "82f2a62f-ff1f-4657-bbef-3ae2a032851f",
      "Employee.Profile.HiredDate": "2022-08-12",              // HiredDate
      "Employee.Profile.Gender": "female",
      "Employee.Profile.Title": "Mrs.",
      "Employee.Profile.GivenName": "Bettye",
      "Employee.Profile.MiddleInitial": "J",
      "Employee.Profile.Surname": "Henley",
      "Employee.Profile.StreetAddress": "3838 Braxton Street",
      "Employee.Profile.City": "Sheridan",
      "Employee.Profile.State": "IL",
      "Employee.Profile.ZipCode": "60551",
      "Employee.Profile.Country": "US",
      "Employee.Profile.EmailAddress": "BettyeJHenley@teleworm.us",
      "Employee.Profile.TelephoneNumber": "815-496-5820",
      "Employee.Profile.TelephoneCountryCode": "1",
      "Employee.Profile.Birthday": "7/30/59",
      "Employee.Profile.NationalID": "325-42-8057",
      "Employee.Profile.Point": "41.463001,-88.609024",
      "Employee.Profile.BaseSalary": 33752,
      "Employee.Profile.Bonus": 3375,
      "Employee.Profile.Department": "Sales",
      "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
      "Employee.Profile.ManagerGUID": "07bb968f-e95e-4c40-8889-d88ba6b369d0"
    },
// ... 82 other results ...
    {
      "Employee.Profile.GUID": "8b7f98f0-60f2-4e40-bd01-bb0e2b30d99e",
      "Employee.Profile.HiredDate": "2022-11-01",               // HiredDate
      "Employee.Profile.Gender": "female",
      "Employee.Profile.Title": "Mrs.",
      "Employee.Profile.GivenName": "Shirley",
      "Employee.Profile.MiddleInitial": "D",
      "Employee.Profile.Surname": "Heath",
      "Employee.Profile.StreetAddress": "3236 Zappia Drive",
      "Employee.Profile.City": "Lexington",
      "Employee.Profile.State": "KY",
      "Employee.Profile.ZipCode": "40507",
      "Employee.Profile.Country": "US",
      "Employee.Profile.EmailAddress": "ShirleyDHeath@superrito.com",
      "Employee.Profile.TelephoneNumber": "859-357-3731",
      "Employee.Profile.TelephoneCountryCode": "1",
      "Employee.Profile.Birthday": "9/17/38",
      "Employee.Profile.NationalID": "406-84-3068",
      "Employee.Profile.Point": "38.028934,-84.538956",
      "Employee.Profile.BaseSalary": 17920,
      "Employee.Profile.Bonus": 1792,
      "Employee.Profile.Department": "Sales",
      "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
      "Employee.Profile.ManagerGUID": "07bb968f-e95e-4c40-8889-d88ba6b369d0"
    }
  ],
```

- Two sets of facets:
  - The `Departments` facet array: Each object's first property contains a unique value encountered in the result's `Department` column. The second property contains the number of times that value was encountered. So, this facet effectively provides the number of employees in each department. Here are all 5 `Department` facets:

```
  "facets": {
    "Departments": [
      {
        "Department": "R&D",
        "count": 30
      },
      {
        "Department": "Sales",
        "count": 24
      },
      {
        "Department": "Engineering",
        "count": 15
      },
      {
        "Department": "Marketing",
        "count": 13
      },
      {
        "Department": "Training",
        "count": 3
      }
    ],
```

- The `States` facet array: Each object's first property contains a unique value encountered in the result's `State` column. The second property contains the number of times that value was encountered. So, this facet effectively provides the number of employees in each state. Here are the first 4 and last 1 of the 33 `State` facets:

  ```
      "States": [
        {
          "State": "WA",
          "count": 4
        },
        {
          "State": "IL",
          "count": 6
        },
        {
          "State": "MI",
          "count": 2
        },
        {
          "State": "OR",
          "count": 1
        },
  // ... 28 other results ...
        {
          "State": "KS",
          "count": 1
        }
      ]
    }
  }
  ```
- These facets allow your end user to not only find out how many new employees are in particular departments or particular states but also to find out how many new employees in particular departments are also in particular states and vice versa.
- You can return just the facets without the query results.

10. Building Semantic Queries

Semantic data in the form of triples that describe the edges of graphs is a powerful data model supported by MarkLogic that you will want to explore. See the *Semantic Graph Developer's Guide* for more detailed information than we provide here.

Briefly, triples allow you to encode interconnected “facts” in a subject-predicate-object form to express a domain of knowledge from which you can infer other “facts.” For example, from these two triples,

> `John` (subject) `Lives In` (predicate) `London` (object) and
>
> `London` (subject) `Is In` (predicate) `England` (object),

we can infer the “fact” that `John Lives In England` without having that “fact” explicitly stored anywhere in our database.

We can also use triples to standardize our data, drawing on publicly available vocabularies such as naming conventions or official abbreviations.

Triples are normally queried with a language called SPARQL.

Optic provides two Data Accessor Functions for triples queries:

- `fromTriples()` directly accesses the triples so that you do not need SPARQL to make simple triple pattern matches.
- `fromSPARQL()` lets you use SPARQL to write the more complex and expressive graph queries needed for searching nested taxonomy structures.

We want to find all our employees in the Northeast. Unfortunately, we only have state data in our employee documents. Fortunately, we do have documents containing semantic triples:

```
ex:CT          rdfs:isDefinedBy "CT" ;
               a                ex:State ;
               skos:broader     ex:Northeast ;
               skos:prefLabel   "Connecticut" .
```

Each of these 4 triples has its own IRI (Internationalized Resource Identifier). They use predefined vocabularies such as RDFS and SKOS shown here as well as others like RDF.

One of these triple facts is that a given state has an official, two-letter abbreviation—which our employee documents use to identify employee states. Another fact is that a given state is in a particular region—such as our needed region, Northeast. This means that we have the data we need to relate our employees' state data from one set of documents with their regions from another set of documents.

So, with this triples data, we can find all our employees in the Northeast in two steps:

The first step is to produce a row sequence of official codes for states in the Northeast.

An Optic query like this one returns up to 100 rows for triples matching the given patterns:

```
const ex    = op.prefixer('https://example.com/semantics/geo#');
const rdfs  = op.prefixer('http://www.w3.org/2000/01/rdf-schema#');
const skos  = op.prefixer('http://www.w3.org/2004/02/skos/core#');

const state = op.col('state')

op.fromTriples([
   op.pattern(state, skos('broader'), ex('Northeast')),
   op.pattern(state, rdfs('isDefinedBy'), op.col('code'))
])
.offsetLimit(0, 100)
.result();
```

We used this query to find all states whose broader definition is Northeast, then, for each found state, to find its official state code:

- We defined three prefixers:
  - `ex` is the base IRI for our triples.
  - `rdfs` is the base IRI for the RDFS vocabulary.
  - `skos` is the base IRI for the SKOS vocabulary.
- We defined two columns with `col()`. They will both appear in our result:
  - `col()` identifies the column in its argument.
  - Before the query, we defined `state`.
  - When it was needed within a query function parameter, we defined `code`.
- The Data Accessor Function `fromTriples()` returns a row for each triple matching the given pattern specified in the `pattern()` functions:
  - The first `pattern()` function finds triples with any subject if `broader` is the predicate and `Northeast` is the object.
  - The second `pattern()` function finds triples with any object if its subject matches one of the states found by the first `pattern()` and its predicate is `isDefinedBy`.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here are rows 1-4 of the 11-row x 2-column result:

```
{
  "state": "https://example.com/semantics/geo#CT",
  "code": "CT"
}
{
  "state": "https://example.com/semantics/geo#DE",
  "code": "DE"
}
{
  "state": "https://example.com/semantics/geo#MA",
  "code": "MA"
}
{
  "state": "https://example.com/semantics/geo#MD",
  "code": "MD"
}
```

- There is one row for each of the 11 Northeastern US states:
  - Its `state` column contains the IRI for the triples graph node.
  - Its `code` column contains the official state code.
- You could suppress the `state` column with the `select(code)` operator function.

- The rows are in an unspecified order, which could change between executions. You can specify row order with the `orderBy()` operator function.

We could have used this `fromSPARQL()` query to get the same results:

```
op.fromSPARQL(`
    PREFIX ex: <https://example.com/semantics/geo#>
    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

    SELECT ?code ?region FROM <https://example.com/semantics/geo> WHERE {
      ?state skos:broader* ?region .
      ?state rdfs:isDefinedBy ?code .
      FILTER (?region = ex:Northeast)
    }
`)
  .offsetLimit(0, 100)
  .result();
```

We would have used it instead of `fromTriples()` if the triples we were interested in were nested in a child structure, because SPARQL has the operator `*`. Used here on `skos:broader`, it would enable the query to search all descendants, not just children.

Either way, we have completed the first step toward finding all our employees in the Northeast. Our second step is to join this triples data with our existing employee data in a multi-model query. The next section describes two ways to accomplish this step.

11. Building Multi-model Queries

Combining data from different models with just a few lines of code is Optic's strength. Optic's Data Accessor Functions represent any model of data they pull from MarkLogic as one model—the row sequence. So, the "multi-model" Optic query effectively becomes a single-model join, union, intersection, or except.

Following are two multi-model query examples that each complete the task we started in Building Semantic Queries: finding all our employees in the Northeast.

11.1. Joining View and Triples Data

With the triples data we leveraged in the last example to retrieve Northeastern states and our employee view with its `State` column, we can find all our employees in the Northeast by building a multi-model query. We will use the now-familiar `fromView()` along with `fromTriples()`, which we introduced in the last section.

An Optic query like this one performs an inner join of a view and triples on a matching column, with the row sequence from the view acting as the "left table," and the row sequence from the triples acting as the "right table":

```
// SELECT FROM EMPLOYEE VIEW
const employees = op.fromView('Employee', 'Profile');

// SELECT WITH CATEGORY EXPANSION VIA TRIPLES
const ex    = op.prefixer('https://example.com/semantics/geo#');
const rdfs  = op.prefixer('http://www.w3.org/2000/01/rdf-schema#');
const skos  = op.prefixer('http://www.w3.org/2004/02/skos/core#');

const state = op.col('state')

const regionalStates =
op.fromTriples([
   op.pattern(state, skos('broader'), ex('Northeast')),
   op.pattern(state, rdfs('isDefinedBy'), op.col('code'))
])

// JOIN VARYING DATA MODELS TO CREATE A MULTI-MODEL QUERY
employees
  .joinInner(regionalStates, op.on(employees.col('State'), regionalStates.col('code')))
  .offsetLimit(0, 100)
  .result();
```

We used this query to retrieve a row sequence with all columns from our `Employee Profile` view and all columns from the triples row sequence we generated previously in Building Semantic Queries where there is an employee document whose `State` column matches a triples `code` column:

- The Data Accessor Function `fromView()` pulls data indexed for the view `Profile` associated with the schema `Employee` into a row sequence of this view's columns.
- The Data Accessor Function `fromTriples()` pulls data into a row sequence containing `state` and `code` as its columns based on the pattern provided as explained in the previous example.
- When you use more than one data accessor function in a query, it is best practice to use a variable to represent each one to keep the query simple and readable.
- The Operator Function `joinInner()` returns all rows from both the view and the triples if the specified columns match, considering the data accessor function specified first in the query to be the "left table" and the second to be the "right table."
- The Auxiliary Function `on()` lets you specify a join condition for join-type operator functions like `joinInner()`.

- `col()` identifies the column in its argument.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here is row 1 of the 100-row x 25-column result:

```
{
  "Employee.Profile.GUID": "f172c249-3f22-4ebb-a29f-aa2b88213d24", // EC1
  "state": "https://example.com/semantics/geo#NY",                 //  TC1
  "Employee.Profile.HiredDate": "2019-08-20",                      // EC2
  "code": "NY",                                                    //  TC2 = EC10
  "Employee.Profile.Gender": "female",                             // EC3
  "Employee.Profile.Title": "Ms.",                                 // EC4, etc.
  "Employee.Profile.GivenName": "Thelma",
  "Employee.Profile.MiddleInitial": "D",
  "Employee.Profile.Surname": "Crider",
  "Employee.Profile.StreetAddress": "2525 Shinn Street",
  "Employee.Profile.City": "New York",
  "Employee.Profile.State": "NY",                                  // EC10 = TC2
  "Employee.Profile.ZipCode": "10017",
  "Employee.Profile.Country": "US",
  "Employee.Profile.EmailAddress": "ThelmaDCrider@superrito.com",
  "Employee.Profile.TelephoneNumber": "212-731-8336",
  "Employee.Profile.TelephoneCountryCode": "1",
  "Employee.Profile.Birthday": "9/28/91",
  "Employee.Profile.NationalID": "115-03-1703",
  "Employee.Profile.Point": "40.709812,-73.908279",
  "Employee.Profile.BaseSalary": 82777,
  "Employee.Profile.Bonus": 8278,
  "Employee.Profile.Department": "Marketing",
  "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
  "Employee.Profile.ManagerGUID": "d0862e5d-5be6-473b-a500-3e3a852b2bb8"
}
```

- This query returned 100 rows for view documents with matching triples documents.
- The rows are in an unspecified order, which could change between executions. You can specify row order with the `orderBy()` operator function.
- The 25 columns are the 23 from our "left table," the `Employee Profile` view (EC#) plus the 2 from our "right table," the triples row sequence (TC#).

11.2. Joining SQL and SPARQL Data

We can also find all our employees in the Northeast with a multi-model inner-join query using `fromSQL()` as our "left table" instead of `fromView()` and `fromSPARQL()` as our "right table" instead of `fromTriples()`. (The `fromSPARQL()` part of this query is from the Building Semantic Queries example):

```
// Select from employee profile columns via SQL:
const sqlView =
      op.fromSQL(`
        SELECT *
        FROM Employee.Profile
      `);

// Select with category expansion via SPARQL:
const sparqlView =
      op.fromSPARQL(`
        PREFIX ex: <https://example.com/semantics/geo#>
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
        PREFIX skos: <http://www.w3.org/2004/02/skos/core#>

        SELECT ?code ?region FROM <https://example.com/semantics/geo> WHERE {
          ?state skos:broader ?region .
          ?state rdfs:isDefinedBy ?code .
          FILTER (?region = ex:Northeast)
        }
      `);

// Join two different data models to create a multi-model query:
sqlView
  .joinInner(sparqlView, op.on('State', 'code'))
  .offsetLimit(0, 100)
  .result()
```

We used this query to obtain results similar to Joining View and Triples Data:

- The Data Accessor Function `fromSQL()` pulls data into a row sequence containing the SELECT-specified columns from the FROM-specified view. In our query, we selected all the columns from our employee view.
- The Data Accessor Function `fromSPARQL()` pulls data into a row sequence containing the SELECT-specified columns from the FROM-specified named graph WHERE the FILTER-specified column matches a specified value. In our query, we selected the `code` and `region` columns where the `region` is `Northeast`.
- When you use more than one data accessor function in a query, it is best practice to use a variable to represent each one to keep the query simple and readable.
- The Operator Function `joinInner()` returns all rows from both `fromSQL`() and `fromSPARQL()` if the specified columns match, considering the data accessor function specified first in the query to be the "left table" and the second to be the "right table."
- The Auxiliary Function `on()` lets you specify a join condition for join-type operator functions like `joinInner()`.
- The Operator Function `offsetLimit()` restricts results returned. The first parameter specifies the number of results to skip; the second, the number of results to return. So, (0, 100) returns the first 100 results.
- The Executor Function `result()` executes the query and returns the results as a row sequence.

Here is row 1 of the 100-row x 25-column result:

```
{
  "Employee.Profile.GUID": "91ed05cb-a2ff-4bd5-9ea1-e8a5829bc665", // EC1
  "code": "MA",                                                     //  TC1 = EC10
  "region": "https://example.com/semantics/geo#Northeast",          //  TC2
  "Employee.Profile.HiredDate": "2013-02-28",                       // EC2
  "Employee.Profile.Gender": "female",                              // EC3
  "Employee.Profile.Title": "Mrs.",                                 // EC4, etc.
  "Employee.Profile.GivenName": "Priscilla",
  "Employee.Profile.MiddleInitial": "J",
  "Employee.Profile.Surname": "Torres",
  "Employee.Profile.StreetAddress": "4121 Cedar Lane",
  "Employee.Profile.City": "Cambridge",
  "Employee.Profile.State": "MA",                                   // EC10 = TC1
  "Employee.Profile.ZipCode": "2142",
  "Employee.Profile.Country": "US",
  "Employee.Profile.EmailAddress": "PriscillaJTorres@teleworm.us",
  "Employee.Profile.TelephoneNumber": "617-639-9170",
  "Employee.Profile.TelephoneCountryCode": "1",
  "Employee.Profile.Birthday": "1/11/45",
  "Employee.Profile.NationalID": "027-18-9604",
  "Employee.Profile.Point": "42.307606,-71.037903",
  "Employee.Profile.BaseSalary": 14480,
  "Employee.Profile.Bonus": 1448,
  "Employee.Profile.Department": "Engineering",
  "Employee.Profile.Status": "Active - Regular Exempt (Full-time)",
  "Employee.Profile.ManagerGUID": "3ad0ffbc-3ade-4897-902b-718417a721f5"
}
```

- This query returned 100 rows for view documents with matching triples documents.
- The rows are in an unspecified order, which could change between executions. You can specify row order with the `orderBy()` operator function.
- The 25 columns are the 23 from our "left table," the `Employee Profile` view (`EC#`) plus the 2 from our "right table," the triples row sequence (`TC#`).

You now know how to use Optic to query your data in many ways. Next, you will learn how to use Optic to update your data.

12. Building Document Updates

Optic also allows you to insert, update, or delete one or more documents through your client application. With this functionality, you can perform both queries and updates through a single API.

This section describes each action in depth with more HR examples.

NOTE
To make bulk insertions of content into a database either as one-off tasks or as part of an external orchestration of a workflow, use the MarkLogic Content Pump (mlcp).

To make as-needed insertions of or updates to documents in a database as part of an external client application's business logic, use the MarkLogic REST API.

12.1. To Insert Documents

You can insert multiple documents at a time--assigning URIs, providing the document data, assigning permissions, and adding the document to various collections among other actions--with a single tidy Optic statement.

Our company was acquired by a larger company. So, we need to add two new departments to our HR documents. Unlike the five original departments, these new departments are temporary and will contain employees that also remain members of their original departments:

- The Integration Department contains employees from our original company selected to work with the new company during transition.
- The Attrition Department contains employees considered potentially redundant as the integration moves forward.

To add these new department documents, we need to provide what are called document descriptors along with the document itself. In our case, we will include three: a unique `URI` for the document, any `collections` the document will belong to, and proper `permissions`.

An Optic update like this one creates documents with these properties:

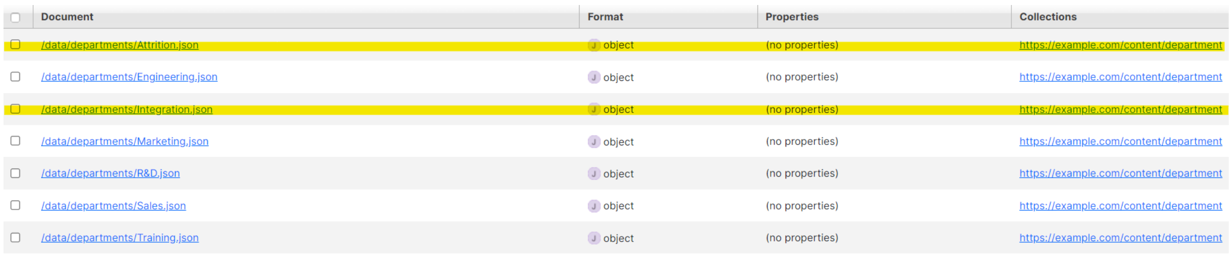
```
declareUpdate();
const op = require("/MarkLogic/optic");
const collections = ["https://example.com/content/department"];
const permissions = xdmp.defaultPermissions();

op.fromDocDescriptors([
   {
      uri: "/data/departments/Integration.json",
      doc: {
            "Department": "Integration",
            "LineOfBusiness": "Cross-Department Collaboration",
            "Description": "The Integration Department helps the new company determine
optimal allocation of resources and employees moving forward."
           },
      collections,
      permissions
   },
   {
      uri: "/data/departments/Attrition.json",
      doc: {
            "Department": "Attrition",
            "LineOfBusiness": "Cross-Department Collaboration",
            "Description": "The Attrition Department contains potentially redundant
employees identified as the integration process progresses."
           },
      collections,
      permissions
   }])
  .write()
  .execute();
```

We used this update to add our 2 new departments:

- The Data Accessor Function `fromDocDescriptors()` creates 2 4-column rows holding our new documents' URIs, data, collections, and permissions. We could also have provided these document descriptor columns:
  - `metadata`
  - `quality`
  - `temporalCollection`
- The Operator Function `write()` inserts each document, by default giving it the URI in the `uri` column created by the latest data accessor function.
- The Executor Function `execute()` executes the update without returning any rows.
- To test the update before writing the document, replace `write()` and `execute()` with `result()` to return the rows that `fromDocDescriptors()` has rendered.

The two new departments take their places among the others in our department collection:



	Document	Format	Properties	Collections
☐	/data/departments/Attrition.json	object	(no properties)	https://example.com/content/department
☐	/data/departments/Engineering.json	object	(no properties)	https://example.com/content/department
☐	/data/departments/Integration.json	object	(no properties)	https://example.com/content/department
☐	/data/departments/Marketing.json	object	(no properties)	https://example.com/content/department
☐	/data/departments/R&D.json	object	(no properties)	https://example.com/content/department
☐	/data/departments/Sales.json	object	(no properties)	https://example.com/content/department
☐	/data/departments/Training.json	object	(no properties)	https://example.com/content/department

- To return rows, use `result()` instead of `execute()`.
- You can insert XML documents equivalent to the JSON ones that we inserted above. This example shows the Integration Department document being inserted using `xdmp.unquote()` and the Attrition Department document being inserted using `NodeBuilder` functions:

```
declareUpdate();
const op = require("/MarkLogic/optic");
const collections = ["https://example.com/content/department"];
const permissions = xdmp.defaultPermissions();
const ns = "http://example.com/example"

const integrationDoc = xdmp.unquote(`
  <Department xmlns="http://example.com/example">
    <Name>Integration</Name>
    <LineOfBusiness>Cross-Department Collaboration</LineOfBusiness>
    <Description>The Integration Department helps the new company determine optimal
allocation of resources and employees moving forward.</Description>
  </Department>
`)

const attritionDoc = new NodeBuilder()
  .startElement("Department", ns)
    .addElement("Name", "Attrition")
    .addElement("LineOfBusiness", "Cross-Department Collaboration")
    .addElement("Description", "The Attrition Department contains potentially redundant
employees identified as the integration process progresses.")
  .endElement()
.toNode()

op.fromDocDescriptors([
   {
      uri: "/data/departments/Integration.xml",
      doc: integrationDoc,
      collections,
      permissions
   },
   {
      uri: "/data/departments/Attrition.xml",
      doc: attritionDoc,
      collections,
      permissions
   }])
  .write()
  .execute();
```

12.2. To Update Document Metadata

You can update a document's metadata without updating the document itself.

We received the list of employees to place in our new departments:

Integration:

- Engineering: Brandy Pace
- Marketing: Thomas Shaner
- R&D: Sage Sutton
- Sales: Eric Molina
- Training: Roseann Seals

Attrition:

- Engineering: Claude Ashe
- Marketing: Mary Lewin
- R&D: Richard Kemble
- Sales: Juana Levenson
- Training: Michael Troxel

To add these employees to their temporary departments, we will add them to collections representing those departments. That way, we do not have to change the employee documents themselves.

An Optic update like this one searches for the employee documents containing each last name then uses the found documents' URIs to update the collections for each document:

```
declareUpdate();
const op = require('/MarkLogic/optic');

const Integration = ['Pace', 'Shaner', 'Sutton', 'Molina', 'Seales'];
const Attrition = ['Ashe', 'Lewin', 'Kemble', 'Levenson', 'Troxel'];
const collection = 'https://example.com/content/employee';

const integrationDescriptors = op.fromDocUris(cts.andQuery([
  cts.collectionQuery(collection),
  cts.jsonPropertyValueQuery("Surname", Integration)
]))
.bind(op.as("collections", [collection, "https://example.com/content/department/
Integration"]))
.result();

const attritionDescriptors = op.fromDocUris(cts.andQuery([
  cts.collectionQuery(collection),
  cts.jsonPropertyValueQuery("Surname", Attrition)
]))
.bind(op.as("collections", [collection, "https://example.com/content/department/
Attrition"]))
.result();

op.fromDocDescriptors(integrationDescriptors.toArray().concat(attritionDescriptors.toArray
()))
  .write()
  .result();
```

We used this update to update the collections associated with the 10 employee documents, adding the new collection for the relevant new department.

This update has three main parts:

1. Creating a row sequence containing one column for the URIs of the 5 Integration employees' documents and another column for their 2 collections, employee and Integration:
   - The constant `Integration` is an array containing the surnames of the employees to be added to the Integration Department.
   - The constant `collection` contains the employee collection URI, which all 10 employees currently belong to and should continue belonging to after this update is executed.
   - The constant `integrationDescriptors` contains a 5-row x 2-column row sequence: one row for each employee to be added to Integration. Each row contains the column that `fromDocUris()` naturally produces and the additional column, `collections`, containing both the employee collection and the Integration collection:
     - The Data Accessor Function `fromDocUris()` produces a row for each document with the column `uri` containing document URIs:
       - The CTS Function `cts.andQuery()` returns the intersection of matches that each of its parameter functions finds:
         - `cts.collectionQuery()` finds all the data from documents in the specified collection, our employee collection.
         - `cts.jsonPropertyValueQuery()` finds all the data--including the URI--for documents whose `Surname` property matches each of the last names in the `Integration` array.
       - With these CTS functions as parameters, `fromDocUris()` produces 5 rows with the `uri` column containing the URIs found for the employee collection documents containing the 5 specified surnames of the employees to add to the Integration collection.

    - The Operator Function `bind()` uses `as()` to define an additional column, `collections`, to hold both the current collection, employee, and the new collection, Integration, for each row in turn.
    - The Executor Function `result()` executes the update and returns the 5-row x 2-column row sequence.
2. Creating a row sequence containing one column for the URIs of the 5 Attrition employees' documents and another column for their 2 collections, employee and Attrition. This part is a repeat of the first part with Attrition instead of Integration.
3. Writing these two row collections, converted to a single array, which adds the proper new collections to the proper employee documents:
    - The Data Accessor Function `fromDocDescriptors()` creates a 10-row x 2-column row sequence, one column holding the URIs of the 10 employee documents to update, and the other column holding the collections to update them with:
        - `toArray()` converts a row sequence into an array.
        - `concat()` concatenates a second parameter to the first of the same type.
        - `fromDocDescriptors()` then uses the array to produce a row sequence with just the `uri` and `collections` columns for the 10 employee documents whose collections need to be updated.
    - The Operator Function `write()` writes just the data provided in the specified columns--in this case, just `collections`--to the document specified by `uri`.
    - The Executor Function `result()` executes the update and returns the 10-row x 2-column row sequence.
    - Before executing this update, you can check that it will affect only the metadata you choose for only the documents you choose by commenting out `write()`. The resulting update returns the rows that `fromDocDescriptors()` accesses.

Here are row 5 and row 6 from the 10-row x 2-column result, one row to represent an employee document for each new collection:

```
{
    "collections": [
    "https://example.com/content/employee",
    "https://example.com/content/department/Integration"
  ],
  "uri": "/data/employees/58fdfd5b-2956-4c7f-a888-8875ff659752.json"
}
{
    "collections": [
    "https://example.com/content/employee",
    "https://example.com/content/department/Attrition"
  ],
  "uri": "/data/employees/67ba986d-af98-49ee-b731-5b230853ad53.json"
}
```

- To add metadata like collections and permissions, you must include all collections for the document in the `collections` column and all permissions for the document in the `permissions` column of `fromDocDescriptors()`. `write()` replaces the existing metadata rather than appending what is provided to what is already there.

12.3. To Update Document Content

You can update a document without needing to provide metadata that you are not updating.

The Integration Department has completed their collection of Marketing, Sales, and Training employees to let go.

To keep track of where they are, they are changing the Attrition Department description:

"The Attrition Department contains redundant employees from Marketing, Sales, and Training."

To update this description, we merely provided the URI and the entire document with the new description replacing the original description:

```
declareUpdate();
const op = require("/MarkLogic/optic");
op.fromDocDescriptors([
   {
      uri: "/data/departments/Attrition.json",
      doc: {
            "Department": "Attrition",
            "LineOfBusiness": "Cross-Department Collaboration",
            "Description": "The Attrition Department contains redundant employees from
Marketing, Sales, and Training."
           }
   }])
  .write()
  .result();
```

Here is the 1 row x 2 column result:

```
{
  "doc": {
     "Department": "Attrition",
     "LineOfBusiness": "Cross-Department Collaboration",
     "Description": "The Attrition Department contains redundant employees from
Marketing, Sales, and Training."
  },
  "uri": "/data/departments/Attrition.json"
}
```

- `write()` writes only the data from the columns that the data accessor provides. So, you do not need to provide `collections` or `permissions` or any other metadata column unless you are also updating that data.
- Before executing this update, you can check that it will affect only the data you choose for only the documents you choose by commenting out `write()`. The resulting update returns the rows that `fromDocDescriptors()` accesses.

12.4. To Update Parts of Document Content

[v11.2.0 and up]

You can update parts of document content such as changing a specific value or adding a new property. You can do this without needing to provide any of the other data or metadata that you are not updating.

We want to change the employee document `Status` property from `Active - Regular Exempt (Full-time)` to `Terminated` for each employee terminated so far.

We also want to add a property: `TerminationDate`.

An Optic update like this one finds all the documents in a collection then replaces one of the values and adds a new property and value to each document found:

```
declareUpdate();
const op = require('/MarkLogic/optic');
op.fromDocUris(cts.collectionQuery('https://example.com/content/department/Attrition'))
  .joinDocCols(null, op.fragmentIdCol('fragmentId'))
  .patch(
    op.col('doc'),
    op.patchBuilder('/')
      .replaceValue("Status", "Terminated")
      .insertAfter("HiredDate", xdmp.toJSON({"TerminationDate":
fn.currentDate()}).root.TerminationDate)
  )
  .write()
  .execute();
```

We used this update to replace the `Status` value and add the `TerminationDate` property--with the current date as its value--after the existing `HiredDate` property in all documents in the Attrition collection:

- The Data Accessor Function `fromDocUris()` produces a row for each document with the column `uri` containing the document URIs and the column `fragmentId` contaning the document root Fragment ID:
  - The CTS Function `cts.collectionQuery()` matches documents in the specified collection, our Attrition collection.
  - With this CTS function as a parameter, `fromDocUris()` produces a row for each matched document. Each row has a `uri` column containing the URIs for the documents that we want to update: the employee documents currently in the Attrition collection. Each row also has a `fragmentID` column containing the Fragment ID of each document.
- The Operator Function `joinDocCols()` retrieves the actual document.
  - `null` tells `joinDocCols()` to use default column names such as `doc` for document content.
  - `fragmentIdCol()` matches the value in `joinDocCols()`'s `fragmentId` column with the value in `fromDocUri()`'s `fragmentId` column.
- The Operator Function `patch()` applies the patch, once it is built, to the specified areas of each document in memory:
  - `col()` identifies the column in its argument.
  - The Operator Function `patchBuilder()` defines a patch consisting of one or more patch builder plan functions:
    - `/` is an XPath expression that selects the root node of the document.
    - The Patch Builder Plan Function `replaceValue()` defines the first change to be applied: Changing the employee status property to terminated:
      - `Status` is an XPath expression selecting the property to update.
      - `Terminated` is the new value for that property.
    - The Patch Builder Plan Function `insertAfter()` defines the second change to be applied: Adding an employee termination date property with `fn.currentDate()` (today's date) determining its value:
      - `HiredDate` is an XPath expression identifying the sibling property after which to add the new property.
      - `xdmp.toJSON()` converts the new native JavaScript Object `{ "Termination":"<current date>" }` into a MarkLogic DocumentObject.
      - `.root.TerminationDate` accesses the `TerminationDate` object node property to insert.
- The Operator Function `write()` inserts each document, by default giving it the URI in the `uri` column created by the latest data accessor function.
- The Executor Function `execute()` executes the update without returning any rows.
- Optic data accessor functions always pull the document root Fragment ID into a `fragmentId` column. However, operator functions do not process that column and executor functions do not produce that column as part of the result unless a preceding operator function explicitly defines that column.

- To test the update before writing the document, replace `write()` and `execute()` with `result()` to return the rows that `fromDocDescriptors()` has rendered.
- You cannot insert a child on an existing property whose value is `null`.
- You cannot add another attribute with the same name to an element.
- You cannot add a JSON property of the same name under the same parent property.
- A document cannot have more than one root. Therefore, you cannot insert a property before or after the root of a document.
- You cannot add an XML element or attribute node to a JSON document nor a JSON property or array node to an XML document.
- Each Patch Builder Function applies to all items matching the provided XPath expression:
  - If it matches more than one node in a document, then the patch applies to all instances.
  - If it matches no property in the document, no changes are applied.

The documents in the Attrition collection now have the old property `Status` updated to `Terminated` and the new property `TerminationDate` containing today's date.

- To return rows, use `result()` instead of `execute()`.

12.5. To Handle Document Update Errors

[v11.2.0 and up]

You can include an error-handling function to continue an update despite errors and to report those errors.

The Integration Department has completed their collection of R&D employees to let go. As before, we have added these employees to the Attrition collection and updated the Attrition Department description.

Also as before, we want to update some individual document properties with the `Status` of `Terminated` and the new property `TerminationDate` using the same update code.

But using the same code will not work: some of the employee documents in the collection already have the new property, so trying to add it again will cause an error, stopping the update process.

We want the process to continue, updating all possible documents and reporting errors for the document updates that fail.

An Optic update like this one completes the update, skipping the documents that fail and adding an error column to the response rows:

```
declareUpdate();
const op = require('/MarkLogic/optic');
op.fromDocUris(cts.collectionQuery('https://example.com/content/department/Attrition'))
  .joinDocCols(null, op.fragmentIdCol('fragmentId'))
  .patch(
    op.col('doc'),
    op.patchBuilder('/')
      .replaceValue('Status', "Terminated")
      .insertAfter('HiredDate', xdmp.toJSON({'TerminationDate':
fn.currentDate()}).root.TerminationDate)
  )
  .write()
  .onError('continue')
  .result();
```

We used this update as before. It is the same except that we added `onError()` and replaced `execute()` with `result()` so that we can see the error column.

- The Operator Function `onError()` with the parameter `continue` continues processing the update after encountering an error, adding the new column `sys.errors` to the resulting rows.

- You can use only one `onError()` function per update.
- `onError()` must be followed directly by either `execute()` or `result()`.
- `onError()` catches only runtime errors, not pre-runtime errors like these:
  - Trying to access a non-existent view.
  - Trying to access a non-existent column.
  - Trying to access a non-existent document.
  - Syntax errors on your SPARQL or SQL strings.
  - Optic Search errors.
- Use `onError('fail')` to fail as soon as an error is encountered. This option yields the same results as not using `onError()` at all.

Here is one of the rows with its error column populated because this Marketing employee document already had the `TerminationDate` property:

```
{
 "uri": "/data/employees/b0d5a15e-b5ce-4138-9a59-f46e08119bc4.json",
 "doc": {
  "GUID": "b0d5a15e-b5ce-4138-9a59-f46e08119bc4",
  "Gender": "male",
  "Title": "Mr.",
  "GivenName": "Ralph",
  "MiddleInitial": "M",
  "Surname": "Deweese",
  "StreetAddress": "2031 O Conner Street",
  "City": "Gulfport",
  "State": "MS",
  "ZipCode": "39507",
  "Country": "US",
  "EmailAddress": "RalphMDeweese@rhyta.com",
  "TelephoneNumber": "228-850-3365",
  "TelephoneCountryCode": "1",
  "Birthday": "11/25/62",
  "NationalID": "428-08-3456",
  "BaseSalary": "75054",
  "Bonus": "7505",
  "Department": "Marketing",
  "Status": "Terminated",
  "ManagerGUID": "d0862e5d-5be6-473b-a500-3e3a852b2bb8",
  "point": {
   "lat": 30.372732,
   "long": -88.999739
  },
  "HiredDate": "2020-03-18"
  "TerminationDate": "2024-03-2024"
 },
 "sys.errors": {
  "data": [
  ],
  "message": "Object nodes cannot have two children with the same name",
  "name": "XDMP-CHILDDUPNAME",
  "documentUri": "/data/employees/b0d5a15e-b5ce-4138-9a59-f46e08119bc4.json",
  "uri": null,
  "operatorId": "2687000105340244565"
 }
}
```

- Use the `'errosOnly'` option of `result()` (for example, `result(null, null, ['errorsOnly'])`) to return only rows containing documents whose update fail.
- To change the default error column name, `sys.errors`, to something else, like `MyErrors`, use `onError('continue',op.col('MyErrors'))`.

12.6. To Delete Documents

The Integration Department has completed their collection of Engineering employees to let go. Along with disbanding the department, management has decided to simply delete the records of the terminated employees.

We must now delete the Attrition Department along with the employee documents associated with that department.

An Optic update like this one deletes documents by their URIs:

```
declareUpdate();
const op = require("/MarkLogic/optic");
op.fromDocUris(cts.orQuery([
    cts.documentQuery("/data/departments/Attrition.json"),
    cts.collectionQuery('https://example.com/content/department/Attrition')
  ])
  )
  .remove()
  .result();
```

We used this update to delete the Attrition Department document as well as all employee documents in the Attrition collection:

- The Data Accessor Function `fromDocUris()` produces a row for each document with the column `uri` containing document URIs:
- - The CTS Function `cts.orQuery()` returns the union of matches that each of its parameter functions finds:
    - `cts.documentQuery()` finds the document with the specified URI, our Attrition Department.
    - `cts.collectionQuery()` finds all the data from documents in the specified collection, our Attrition collection.
  - With these CTS functions as parameters, `fromDocUris()` produces a row with the `uri` column containing the URI for each document that we want to delete: the Attrition department document and each employee document in the Attrition collection.
- The Operator Function `remove()` marks for system deletion the documents whose URIs match the ones in `fromDocUris()`'s `uri` column, leaving behind only the `uri` column.
- The Executor Function `result()` executes the update and returns the results as a row sequence.

Here are the first 6 rows of the 1-column result:

```
{
 "uri": "/data/departments/Attrition.json"
}
{
 "uri": "/data/employees/f7d055a5-9462-424c-8e9b-13db2878145f.json"
}
{
 "uri": "/data/employees/bd5e4f9c-14d5-4f88-89d0-4434fc9318f4.json"
}
{
 "uri": "/data/employees/2ef6c49b-ac42-4f83-b887-e31394843ef9.json"
}
{
 "uri": "/data/employees/67ba986d-af98-49ee-b731-5b230853ad53.json"
}
{
 "uri": "/data/employees/9a4d5d9a-e04e-41a7-a2ac-1bd64f6a2edd.json"
}
```

- These are the URIs of the deleted documents.

- Before executing this update, you can check that your `fromDocUris()` statement will return the URIs of the documents that you want to delete by commenting out `remove()`. The resulting update returns the rows that `fromDocUris()` accesses.
- Use `execute()` to execute the update without returning the resultant row sequence.
- With no documents left in it, the Attrition collection automatically vanishes.

You now know how to build Optic updates to insert, update, and delete documents.

13. What’s Next?

Having read this guide, you are now ready to load some of your own data into Query Console to write your own Optic queries and updates.

14. Technical support

Progress Software provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at `http://help.marklogic.com` to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the *Support Handbook* for instructions on registering support contacts and on working with the MarkLogic Server Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at `http://developer.marklogic.com`. For technical questions, we encourage you to ask your question on Stack Overflow.

15. Copyright