MongoSQL Reference

General

SQL Syntax in MongoDB Shell vs. 3rd Party SQL Based Tools (going through ODBC/JDBC driver)

When executing a SQL statement within the MongoDB Shell environment, one must include the proper wrapping syntax that includes the aggregate command and proper quoting (see Quoting for more details). If executing a SQL statement by way of a 3rd party tool that would be connected using one of our Atlas SQL drivers (ODBC or JDBC), the MongoDB aggregate command is not necessary and in some cases the quotes are not needed either. In many cases (though it depends on the tool) the connectors and drivers add these code wrappers when doing the SQL to MQL translation.

Tool	SQL Syntax	Additional Info

MongoDB Shell	`db.sql(`Select username from Customers where username='fmiller' limit 2`)`
	
	`db.aggregate([{$sql: {statement: `SELECT * from Sales where storeLocation='Seattle' limit 2`, format:"jdbc", dialect:"mongosql"}}])`
		db.sql is the shortened form of the $sql aggregate.

DBeaver & Tableau	`Select username from Customers where username='fmiller' limit 2`	

Quoting - In some areas, Atlas SQL can be pretty forgiving with quotes and which types can be used. And many times if coming in from a BI Tool, you only need to worry about quotes when identifying a string literal (ex. A string within the where clause). But if you are in Mongo shell, you need to add quotes to the whole statement. And if you need quotes in other areas of your statement (column names or string filters) this is a general guide for quote usage and the types of quotes to use.

The quote marks you use in SQL Statements matter.

`` entire statement *tick marks*

“” column names *double quotes*

‘’ strings *single quotes*

Using Quotes with Atlas SQL in Mongo Shell

db.sql("Select username from customers where username='fmiller' limit 2")

```
db.sql(`Select username from customers where username='fmiller' limit 2`)
```

```
db.sql(`Select "username" from customers where username='fmiller' limit 2`)
```

Using Quotes with Atlas SQL in DBeaver or Tableau

Quoting Syntax Examples

Select purchaseMethod **from** Sales

Select purchaseMethod **from** Sales **where** purchaseMethod ='Online'

Select "purchaseMethod" **from** Sales **where** purchaseMethod ='Online'

Select "Sales"."customer" **AS** "customer","Sales"."customer"."age" **AS** "customerAge", "Sales"."purchaseMethod" **AS** "purchaseMethod" **From** "Sales" **Where** "purchaseMethod"='Online'

Type	Construct/Operators	Syntax/Example Examples are displayed for Mongo Shell, these can be applied to other tools by removing the db.sql(``)	Syntax/Example (connecting through a driver and/or connector)	Additional Notes
Object				

Identifiers				
	Collection/Table	db.sql(`select * from Sales Limit 2`)	select * from Sales Limit 2	Case Sensitive: Sales is the name of the virtual collection in the Atlas Federated Database (ADF). All MongoDB namespaces are case sensitive, the SQL syntax is not.
	Field/Column	db.sql(`select saleDate, items, customer.age from Sales Limit 2`)	select saleDate, items, customer.age from Sales Limit 2	saleDate is a top level field, items is an array, and customer.age is a part of a nested document. Customer Age can be displayed as a column/top level field by using dot notation. Because the items array is not unwound, it would display the whole array in Mongo Shell and in DBeaver or Tableau it would display as one column stringified
	String Literals	Single quotes: db.sql(`select * from Sales Where customer.gender = 'M' Limit 2`)	select * from Sales Where customer.gender = 'M' Limit 2	Notice the M, is in single quotes. While the whole SQL stmt is surrounded in ticks.
Query Syntax				

	SELECT	db.sql(`select * from Sales Limit 2`) db.sql(`select purchaseMethod,customer, items from Sales Limit 2`)	select * from Sales Limit 2 select purchaseMethod,customer, items from Sales Limit 2	While the commonly used syntax "Select * from Table" is supported, the syntax to combine * plus column names is not, the query "Select *, FieldA from Table" will product an error
	DISTINCT	Currently not supported Not compatible with documents because of the way field ordering and comparing would work		Count Distinct is working, Select Distinct is not supported
		db.sql(`SELECT COUNT(DISTINCT purchaseMethod) FROM Sales`)	SELECT COUNT(DISTINCT purchaseMethod) FROM Sales	Count Distinct is working, Select Distinct is not supported

	CASE	db.sql(`SELECT CASE WHEN customer.age <=20 THEN '20 years old or younger' WHEN customer.age >20 AND customer.age <=30 THEN '21-30 year olds' WHEN customer.age >30 AND customer.age <=40 THEN '31-40 year olds' WHEN customer.age >40 AND customer.age <=50 THEN '41-50 year olds' WHEN customer.age >50 AND customer.age <=60 THEN '51-60 year olds' WHEN customer.age >60 AND customer.age <=70 THEN '61-70 year olds' WHEN customer.age >70 THEN '70 years and older' ELSE 'Other' END AS ageRange, customer.age, customer.gender, customer.email FROM Sales`)	SELECT CASE WHEN customer.age <=20 THEN '20 years old or younger' WHEN customer.age >20 AND customer.age <=30 THEN '21-30 year olds' WHEN customer.age >30 AND customer.age <=40 THEN '31-40 year olds' WHEN customer.age >40 AND customer.age <=50 THEN '41-50 year olds' WHEN customer.age >50 AND customer.age <=60 THEN '51-60 year olds' WHEN customer.age >60 AND customer.age <=70 THEN '61-70 year olds' WHEN customer.age >70 THEN '70 years and older' ELSE 'Other' END AS ageRange, customer.age, customer.gender, customer.email FROM Sales	This shows a typical CASE expression using dot notation to select a field that is within a nested document.
	FROM	db.sql(`select * from Sales Limit 2`)	select * from Sales Limit 2	

	JOIN		SELECT b.ProductSold, CAST(b.`_id` as string) ID, (b.Price * b.Quantity) totalAmount FROM (SELECT * FROM Sales a WHERE customer.gender='F') a Inner Join Transactions b on (Cast(a.`_id` as string)=Cast(b.`_id` as string)) select b.ProductSold,b.`_id` as ID, (b.Price * b.Quantity) as SaleTotal,a.couponUsed from Sales a inner join Transactions b on Cast(a.`_id` as string)=Cast(b.`_id` as string) limit 2 select b.ProductSold,b.`_id` as ID, (b.Price * b.Quantity) as SaleTotal,a.couponUsed from Sales a inner join Transactions b on Cast(a.`_id` as string)=Cast(b.`_id` as string) where Cast(a.`_id`as string)='5bd761dcae323e45a93ccff4'	It's best to filter or limit the data as much as possible to ensure speed with query execution The first example shows how to form the SQL query to create a derived table that joins after the where clause filter MongoSQL supports: INNER JOIN, (CROSS) JOIN, LEFT OUTER JOIN, and RIGHT OUTER JOIN
	UNION ALL		SELECT * FROM Sales UNION ALL SELECT * FROM Transactions	UNION is not supported, UNION ALL is supported The main difference between UNION and UNION ALL is that: UNION: only keeps unique records. UNION ALL: keeps all records,

				including duplicates
	NESTED SELECTS		SELECT .. FROM (SELECT ..) as subSelect	Mongosql requires nested selects to have an alias. This is not a SQL-92 requirement.
	JOINED SUBQUERIES		SELECT ... FROM (SELECT ....) as sub1 JOIN (SELECT ....) as sub2 ON	
	WHERE	db.sql(`SELECT * from Sales WHERE customer.gender='M'`) db.sql(`SELECT * FROM Sales WHERE customer.age>20`)	SELECT * from Sales WHERE customer.gender='M' SELECT * FROM Sales WHERE customer.age>20	
	LIKE		SELECT purchaseMethod FROM Sales WHERE purchaseMethod LIKE 'In%'	
	ESCAPE		SELECT customer FROM Sales WHERE customer.email LIKE '%_%' ESCAPE '_'	The ESCAPE clause is supported in the LIKE operator to indicate the escape character. Escape characters are used in the pattern string to indicate that any wildcard character that occurs after the escape character in the pattern string should be treated as a regular character.

	GROUP BY	db.sql(`SELECT customerAge, COUNT(*) FROM Sales GROUP BY customer.age AS customerAge`) db.sql(`SELECT customerGender,customerAge, COUNT(*) FROM Sales GROUP BY customer.gender AS customerGender, customer.age AS customerAge`) db.sql(`SELECT ProductSold,Sum(Price) FROM Transactions GROUP BY ProductSold`)	SELECT customerAge, COUNT(*) FROM Sales GROUP BY customer.age AS customerAge SELECT customerGender,customerAge, COUNT(*) FROM Sales GROUP BY customer.gender AS customerGender, customer.age AS customerAge SELECT ProductSold,Sum(Price) FROM Transactions GROUP BY ProductSold	
	HAVING	db.sql(`SELECT customerGender,customerAge, COUNT(*) FROM Sales GROUP BY customer.gender AS customerGender, customer.age AS customerAge Having COUNT(*)>1`)	SELECT customerGender,customerAge, COUNT(*) FROM Sales GROUP BY customer.gender AS customerGender, customer.age AS customerAge Having COUNT(*)>1	
	ORDER BY	db.sql(`SELECT customerGender, COUNT(*) FROM Sales GROUP BY customer.gender AS customerGender ORDER BY customerGender`)	SELECT customerGender, COUNT(*) FROM Sales GROUP BY customer.gender AS customerGender ORDER BY customerGender	
	LIMIT	db.sql(`SELECT * FROM Sales LIMIT 3`)	SELECT * FROM Sales LIMIT 3	

	OFFSET	db.sql(`SELECT couponUsed FROM Sales OFFSET 2`)	SELECT couponUsed FROM Sales OFFSET 2	
	AS	db.sql("SELECT couponUsed AS Coupons FROM Sales OFFSET 2") db.sql(`SELECT customerAge, COUNT(*) FROM Sales GROUP BY customer.age AS customerAge`)	SELECT couponUsed AS Coupons FROM Sales OFFSET 2 SELECT customerAge, COUNT(*) FROM Sales GROUP BY customer.age AS customerAge	Alias assignments work as expected, though the syntax is a bit unexpected when using an aggregate with a subdocument namespace - see 2nd example.
Arithmetic Operators				
	=+, -, *, /, %	db.sql(`SELECT ProductSold,Price, Quantity,(Price*Quantity) as TotalCost FROM Transactions Limit 2`)	SELECT ProductSold,Price, Quantity,(Price*Quantity) as TotalCost FROM Transactions Limit 2	+ (Addition) → + - (Subtraction) → - * (Multiplication) → * / (Division) → / % (Modulus remainder) → MOD({val1},{val2}) These arithmetics operators all work as the syntax example shows, just replace the operator you need within this syntax example. Only % uses a function.

Comparison Operators				
	=, !=, <>, >, <, >=, <=	db.sql(`SELECT * FROM Sales WHERE customer.age>20`) db.sql(` SELECT * FROM Sales WHERE customer.gender='F'`)	SELECT * FROM Sales WHERE customer.age>20 SELECT * FROM Sales WHERE customer.gender='F'	= Equals != Not Equal <> Not Equal > Greater Than >= Greater Than & Equal < Less Than <= Less Than & Equal These comparison operators all work as the syntax example suggests, just replace the operator you need within this syntax example.
Logical/Boolean Operators				
	AND, NOT, OR	db.sql(`SELECT * FROM Sales WHERE customer.age>20 AND customer.gender='M'`) db.sql(`SELECT * FROM Sales WHERE customer.age=20 OR customer.gender='M'`) db.sql(`SELECT * FROM Sales WHERE	SELECT * FROM Sales WHERE customer.age>20 AND customer.gender='M' SELECT * FROM Sales WHERE customer.age=20 OR customer.gender='M' SELECT * FROM Sales WHERE customer.age>20 AND NOT customer.gender='M'	

		customer.age>20 AND NOT customer.gender='M'`)		
Aggregate Expressions				
	SUM()	db.sql(`SELECT ProductSold,SUM(Price) FROM Transactions GROUP BY ProductSold`)	SELECT ProductSold,SUM(Price) FROM Transactions GROUP BY ProductSold	
	AVG()	db.sql(`SELECT ProductSold,AVG(Price) FROM Transactions GROUP BY ProductSold`)	SELECT ProductSold,AVG(Price) FROM Transactions GROUP BY ProductSold	
	COUNT()	db.sql(`SELECT ProductSold,COUNT(Price) FROM Transactions GROUP BY ProductSold`)	SELECT ProductSold,COUNT(Price) FROM Transactions GROUP BY ProductSold	
	MIN()	db.sql(`SELECT ProductSold,MIN(Price) FROM Transactions GROUP BY ProductSold`)	SELECT ProductSold,MIN(Price) FROM Transactions GROUP BY ProductSold	
	MAX()	db.sql(`SELECT ProductSold,MAX(Price) FROM Transactions GROUP BY ProductSold`	SELECT ProductSold,MAX(Price) FROM Transactions GROUP BY ProductSold	
	COUNT(DISTINCT)	db.sql(`SELECT COUNT(DISTINCT purchaseMethod) FROM Sales`)	SELECT COUNT(DISTINCT purchaseMethod) FROM Sales	Does not work if the aggregated field is not comparable to itself. Document, Array, no schema set or polymorphic

				schema.
	SUM(DISTINCT)	db.sql(`SELECT ProductSold,SUM(DISTINCT Price) FROM Transactions GROUP BY ProductSold`)	SELECT ProductSold,SUM(DISTINCT Price) FROM Transactions GROUP BY ProductSold	
	GROUP_CONCAT()			Not Supported
Scalar Functions				
	CONCAT	db.sql(`SELECT purchaseMethod||' '||storeLocation as purchaseDetails,purchaseMethod, storeLocation from Sales`)	SELECT purchaseMethod||' '||storeLocation as purchaseDetails,purchaseMethod, storeLocation from Sales	|| operator for concatenating strings This example shows how to concatenate 2 strings, while adding a space between them.
	DATETRUNC	db.sql(`SELECT DATETRUNC(DAY,saleDate) from Sales`)	SELECT DATETRUNC(DAY,saleDate) from Sales	The DATETRUNC function shortens timestamps so they are easier to read. We support DATETRUNC(<date_part>, <timestamp>) and DATETRUNC(<date_part>, <timestamp>, '<start_of_week>') Supported date_parts are YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | WEEK |

				DAY_OF_YEAR | ISO_WEEK | ISO_WEEKDAY
	DATEADD	db.sql(`SELECT DATEADD(YEAR,1,saleDate), saleDate from Sales`)	SELECT DATEADD(YEAR,1,saleDate), saleDate from Sales	Supported date_parts are YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | WEEK | DAY_OF_YEAR | ISO_WEEK | ISO_WEEKDAY
	DATEDIFF	db.sql(`SELECT DATEDIFF(YEAR,CURRENT_TIMESTAMP,saleDate),saleDate from Sales`)	SELECT DATEDIFF(YEAR,CURRENT_TIMESTAMP,saleDate),saleDate from Sales	Supported date_parts are YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | WEEK | DAY_OF_YEAR | ISO_WEEK | ISO_WEEKDAY
	TO_STRING	db.sql(`SELECT CAST(saleDate as string), saleDate from Sales`)	SELECT CAST(saleDate as string), saleDate from Sales SELECT saleDate::STRING as salesDateStr, saleDate from Sales SELECT CAST(EXTRACT(YEAR FROM saleDate)as string), saleDate from Sales	CAST(<expr> AS STRING) or ::STRING
	TO/FROM EPOCH			This can be done with CAST. To Epoch == CAST(<timestamp> AS LONG) From Epoch == CAST(<epoch> AS

				TIMESTAMP)
	CASTING TO/FROM DATE, TIMESTAMP	db.sql(`SELECT CAST(EXTRACT(YEAR FROM saleDate)as integer), saleDate from Sales`) db.sql(`SELECT CAST(EXTRACT(YEAR FROM saleDate)as string), saleDate from Sales`) db.sql(`SELECT CAST('1975-01-23' as TIMESTAMP) as Birthdate, saleDate from Sales`)	SELECT CAST(EXTRACT(YEAR FROM saleDate)as integer), saleDate from Sales SELECT CAST(EXTRACT(YEAR FROM saleDate)as string), saleDate from Sales SELECT CAST('1975-01-23' as TIMESTAMP) as Birthdate, saleDate from Sales	MongoDB only supports TIMESTAMP type. supports: "cast from date/timestamp to/from other types"
	DAY OF WEEK	db.sql(`SELECT EXTRACT(ISO_WEEKDAY FROM saleDate), saleDate from Sales`)	SELECT EXTRACT(ISO_WEEKDAY FROM saleDate), saleDate from Sales	EXTRACT(ISO_WEEKDAY FROM <timestamp>)
	TIMEZONE CONVERSION			Not Supported, MongoDB stores in UTC
	CURRENT DATE TIME	db.sql(`SELECT CURRENT_TIMESTAMP from Sales`)	SELECT CURRENT_TIMESTAMP from Sales	
	EXTRACT	db.sql(`SELECT EXTRACT(YEAR FROM saleDate), saleDate from Sales`)	SELECT EXTRACT(YEAR FROM saleDate), saleDate from Sales	
	RANDOM()			Not Supported

	SUBSTRING()	db.sql(`SELECT ProductSold, SUBSTRING (ProductSold,0,2)FROM Transactions`)	SELECT ProductSold, SUBSTRING (ProductSold,0,2)FROM Transactions	SUBSTRING(string, start position, length) MongoSQL uses 0-indexing, so start should be 0 to start at the first character, 1 to start at the second, and so on
	UPPER	db.sql(`SELECT ProductSold, UPPER(SUBSTRING (ProductSold,0,2))FROM Transactions`)	SELECT ProductSold, UPPER(SUBSTRING (ProductSold,0,2))FROM Transactions	UPPER(string)
	LOWER	db.sql(`SELECT ProductSold, LOWER(SUBSTRING (ProductSold,0,2))FROM Transactions`)	SELECT ProductSold, LOWER(SUBSTRING (ProductSold,0,2))FROM Transactions	LOWER(string)
	TRIM	db.sql(`SELECT TRIM(purchaseMethod) FROM Sales`) db.sql(`SELECT TRIM('In' FROM purchaseMethod) FROM Sales`)	SELECT TRIM(purchaseMethod) FROM Sales SELECT TRIM('In' FROM purchaseMethod) FROM Sales	removes leading and/or trailing spaces, or an optionally specified substring, from the argued string
	LEFT	db.sql(`SELECT ProductSold, SUBSTRING (ProductSold,0,2)FROM Transactions`)	SELECT ProductSold, SUBSTRING (ProductSold,0,2)FROM Transactions	Use Substring with a starting position of 0 or use: (CASE WHEN %2 <= 0 THEN NULL ELSE SUBSTRING(%1,0,%2) END)

	RIGHT	db.sql(`SELECT SUBSTRING (ProductSold,(CHAR_LENGTH( ProductSold)-2),2)FROM Transactions`)	SELECT SUBSTRING (ProductSold,(CHAR_LENGTH(ProductS old)-2),2)FROM Transactions	Use a combo of Substring and Char_length minus the length from the substring argument or use: (CASE WHEN %2 <= 0 THEN (CASE WHEN %1 IS NULL THEN NULL ELSE '' END) WHEN %2 >= CHAR_LENGTH(%1) THEN %1 ELSE SUBSTRING(%1 FROM (CHAR_LENGTH(%1) - %2) FOR %2) END)
	CHAR_LENGTH	db.sql(`SELECT CHAR_LENGTH(ProductSold) FROM Transactions`)	SELECT CHAR_LENGTH(ProductSold) FROM Transactions	
	POSITION	db.sql(`SELECT purchaseMethod, POSITION ('i' IN purchaseMethod) FROM Sales`)	SELECT purchaseMethod, POSITION ('i' IN purchaseMethod) FROM Sales	POSITION(substring IN string) returns the position of the first occurrence of substring in the string, or -1 if it does not occur
	SIMILAR TO			Not Supported

MongoDB commands:

Flatten & Unwind

How can I flatten all of the data within a nested object?

Initial view of data within Tableau

Tableau Table View: Select * from Sales where customer.age=59

_id	Coupon Used	Customer

_Id {"$oid": "5bd761dcae323e45a93ccff4"}	False	{"gender": "F", "age": 59, "satisfaction": 4}
_Id {"$oid": "5bd761dcae323e45a93ccff4"}	False	Customer {"gender": "M", "age": 59, "satisfaction": 5}
_Id {"$oid": "5bd761dcae323e45a93ccfee"}	True	Customer {"gender": "F", "age": 59, "email": "pan@cak.zm", "satisfaction": 5}

Flatten Command:

This selects and flattens all nested fields within the customer object and presents them as top level fields (aka columns):

Tableau Table View: Select * from FLATTEN(Sales) where customer_age=59

_id	Coupon Used	customer_age	customer_gender	customer_satisfaction
_Id {"$oid": "5bd761dcae323e45a93ccff4"}	False	59	F	4
_Id {"$oid": "5bd761dcae323e45a93ccff4"}	False	59	M	5
_Id {"$oid": "5bd761dcae323e45a93ccfee" }	True	59	F	5

How do I only flatten some fields, I don't need them all?

Initial view of data within Tableau

Tableau Table View: Select * from Sales where purchaseMethod ='Online'

_id	Coupon Used	Customer
_Id {"$oid": "5bd761dcae323e45a93ccff4"}	False	{"gender": "F", "age": 59, "satisfaction": 4}
_Id {"$oid": "5bd761dcae323e45a93ccff3"}	False	Customer {"gender": "M", "age": 59, "satisfaction": 5}

_Id {"$oid": "5bd761dcae323e45a93ccfee"}	True	Customer {"gender": "F", "age": 59, "email": "pan@cak.zm", "satisfaction": 5}

Use dot notation:

This selects customer name and email (which are nested within the customer object) and presents them as top level fields (aka columns):

Select CAST(_id as String) as ID_new, customer.age, customer.gender from Sales where purchaseMethod ='Online'

_id	Coupon Used	customer.age	customer.gender
5bd761dcae323e45a93ccff4	False	59	F
5bd761dcae323e45a93ccff3	False	59	M
5bd761dcae323e45a93ccfee	True	59	F

How can I unwind array data?

The "items" column is an array data type

Tableau Table View: Select * from Sales

_id	Coupon Used	Customer	items
_Id {"$oid": "5bd761dcae323e45a93ccff4"}	False	{"gender": "F", "age": 59, "satisfaction": 4}	[{"name": "backpack", "price": {"$numberDecimal": "187.16"}, "quantity": 2}, {"name": "printer paper", "price": {"$numberDecimal": "20.61"}, "quantity": 10}, {"name": "notepad", "price": {"$numberDecimal": "23.75"}, "quantity": 5}]

This unwinds and flattens all of the "items" array and presents all nested object data as top level fields (columns):

Select * **from** FLATTEN(UNWIND(Sales **WITH PATH**=> Sales.items))

_id	Coupon Used	customer_age	customer_gender	customer_satisfaction	items_name	items_price	items_quantity
_Id {"$oid": "5bd761dcae323e45a93ccff4"}	False	59	F	4	backpack	187.16	2
_Id {"$oid": "5bd761dcae323e45a93ccff4"}	False	59	F	4	printer paper	20.61	10
_Id {"$oid": "5bd761dcae323e45a93ccff4"}	False	59	F	4	notepad	23.75	5

*This unwinds the "items" array but only selects some of the fields within the array (the quantity, price, and tags) this also shows how to use a where clause (note that you do not have to flatten the data when not returning *):*

Select CAST(_id as String),items.quantity, items.name, items.price from UNWIND(Sales WITH PATH=> Sales.items)

_id	items_name	items_price	items_quantity
5bd761dcae323e45a93ccff4	backpack	187.16	2
5bd761dcae323e45a93ccff4	printer paper	20.61	10
5bd761dcae323e45a93ccff4	notepad	23.75	5

This unwinds the "items" array but only selects some of the fields within the array (the quantity and price), selects some top level fields (_id and purchaseMethod) and also selects a nested object field (customer_age) plus this also shows how to use a where clause with an integer data type:

******Notice the Flatten command can be omitted, since we are not intending to Flatten every nested object as we are using dot notation to select specific fields (both nested objects and array types)***

Select CAST(_id as String),purchaseMethod, customer.age, items.quantity, items.price from UNWIND(Sales WITH PATH=> Sales.items) Where items.quantity = 2

_id	purchaseMethod	customer.age	items.price	items.quantity
5bd761dcae323e45a93ccff4	Online	59	187.16	2
5bd761dcae323e45a93ccff4	Online	59	22.61	2
5bd761dcae323e45a93ccff4	Store	59	24.75	2

MongoSQL Translation

The components within the Atlas SQL Interface allow for SQL to be translated to MQL. This translation allows for SQL tools and users to interact with Atlas data. These translations into MQL may not render the most efficient query. While we expose how the SQL is translated, this may not be the best substitute for the most efficient MQL. The Query Planner Extended can help users to see how SQL is translated into MQL to then troubleshoot efficiency and performance.

***this can be run in shell*

db.aggregate([{$sql: {statement: `SELECT * from Sales where storeLocation='Seattle' limit 2`, format:"jdbc", dialect:"mongosql"}}]).explain('queryPlannerExtended')

```
{ ok: 1,
 stats: { size: '4.127714157104492 MiB', numberOfPartitions: 1 },
 plan:
  { kind: 'region',
    region: 'AWS/us-east-1',
    node:
     { kind: 'data',
       size: '4.127714157104492 MiB',
       numberOfPartitions: 1,
       partitions:
        [ { source: 'TestFree',
            provider: 'atlas',
```

```
        size: '4.127714157104492 MiB',
        database: 'sample_supplies',
        collection: 'sales',
        pipeline:
         [ { '$match':
              { '$expr':
                 { '$cond':
                    { if: { '$lte': [ '$storeLocation', null ] },
                      then: null,
                      else: { '$eq': [ '$storeLocation', 'Seattle' ] } } } } },
           { '$limit': 2 },
           { '$project': { _id: 0, Sales: '$$ROOT' } } ] } } }
```