# DocumentDB 中的 SQL 查询和 SQL 语法

Microsoft Azure DocumentDB 通过将 SQL（结构化查询语言）用作 JSON 查询语言来支持查询文档。DocumentDB 是真正无架构的。凭借其对数据库引擎内 JSON 数据模型的直接承诺，它可以提供 JSON 文档的自动索引，而无需显式架构或创建辅助索引。

在设计 DocumentDB 的查询语言时，我们有两个目标：

* 我们希望支持 SQL，而不是发明一种新的 JSON 查询语言。SQL 是最常见和最常用的查询语言之一。DocumentDB SQL 提供了一种正式的编程模型，用于对 JSON 文档进行丰富查询。
* 由于 JSON 文档数据库能够在数据库引擎中直接执行 JavaScript，我们希望将 JavaScript 的编程模型用作我们的查询语言的基础。DocumentDB SQL 植根于 JavaScript 的类型系统、表达式计算和函数调用中。而这反过来为关系投影、跨 JSON 文档的分层导航、自联接、空间查询以及调用完全采用 JavaScript 编写的用户定义的函数 (UDF) 和其他功能提供了自然编程模型。

我们相信这些功能是减少应用程序和数据库之间冲突的关键，并且对于开发人员的工作效率来说是至关重要的。

我们建议从观看下面的视频开始，在此视频中，Aravind Ramachandran 会演示 DocumentDB 的查询功能，并且你可以首先访问我们的 [Query Playground（查询板块）](http://www.documentdb.com/sql/demo)，在其中尝试 DocumentDB 并对我们的数据集运行 SQL 查询。

[AZURE.VIDEO dataexposedqueryingdocumentdb]

然后，返回到本文中，我们将从 SQL 查询教程开始，指导你完成一些简单的 JSON 文档和 SQL 命令。

## DocumentDB 中的 SQL 命令入门

为了解 DocumentDB SQL 在工作时的情况，让我们以一些简单的 JSON 文档开始，并对它完成一些简单的查询。考虑以下两个关于两个家庭的 JSON 文档。请注意，使用 DocumentDB，我们不需要显式创建任何架构或辅助索引。我们只需将 JSON 文档插入到 DocumentDB 集合中并随后进行查询。这里，我们有一个包含 Andersen 家庭、父母、子女（以及他们的宠物）、地址和注册信息的简单 JSON 文档。该文档拥有字符串、数字、布尔、数组和嵌套属性。

**文档**

{  
 "id": "AndersenFamily",  
 "lastName": "Andersen",  
 "parents": [  
 { "firstName": "Thomas" },  
 { "firstName": "Mary Kay"}  
 ],  
 "children": [  
 {  
 "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 }  
 ],  
 "address": { "state": "WA", "county": "King", "city": "seattle" },  
 "creationDate": 1431620472,  
 "isRegistered": true  
}

下面是另一个有着细微差异的文档 – 其中使用了 givenName 和 familyName，取代了 firstName 和 lastName。

**文档**

{  
 "id": "WakefieldFamily",  
 "parents": [  
 { "familyName": "Wakefield", "givenName": "Robin" },  
 { "familyName": "Miller", "givenName": "Ben" }  
 ],  
 "children": [  
 {  
 "familyName": "Merriam",   
 "givenName": "Jesse",   
 "gender": "female", "grade": 1,  
 "pets": [  
 { "givenName": "Goofy" },  
 { "givenName": "Shadow" }  
 ]  
 },  
 {   
 "familyName": "Miller",   
 "givenName": "Lisa",   
 "gender": "female",   
 "grade": 8 }  
 ],  
 "address": { "state": "NY", "county": "Manhattan", "city": "NY" },  
 "creationDate": 1431620462,  
 "isRegistered": false  
}

现在，让我们尝试对此数据执行一些查询来了解 DocumentDB SQL 的一些主要方面。例如，下面的查询将返回其中的 ID 字段与 AndersenFamily 匹配的文档。由于它是 SELECT \*，因此该查询的输出为完整的 JSON 文档：

**查询**

SELECT \*   
FROM Families f   
WHERE f.id = "AndersenFamily"

**结果**

[{  
 "id": "AndersenFamily",  
 "lastName": "Andersen",  
 "parents": [  
 { "firstName": "Thomas" },  
 { "firstName": "Mary Kay"}  
 ],  
 "children": [  
 {  
 "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 }  
 ],  
 "address": { "state": "WA", "county": "King", "city": "seattle" },  
 "creationDate": 1431620472,  
 "isRegistered": true  
}]

现在，考虑我们需要将 JSON 输出的格式重新设置为一种不同的形式。当地址的城市名称与省/自治区名称相同时，此查询使用两个选定的字段 Name 和 City 表示新的 JSON 对象。在这种情况下，“NY, NY”匹配。

**查询**

SELECT {"Name":f.id, "City":f.address.city} AS Family   
FROM Families f   
WHERE f.address.city = f.address.state

**结果**

[{  
 "Family": {  
 "Name": "WakefieldFamily",   
 "City": "NY"  
 }  
}]

下一个查询返回其 ID与按居住城市排序的 WakefieldFamily 匹配的家庭中所有子女的给定名称。

**查询**

SELECT c.givenName   
FROM Families f   
JOIN c IN f.children   
WHERE f.id = 'WakefieldFamily'  
ORDER BY f.address.city ASC

**结果**

[  
 { "givenName": "Jesse" },   
 { "givenName": "Lisa"}  
]

我们希望通过我们目前已看到的示例让你注意到 DocumentDB 查询语言一些值得注意的方面：

* 由于 DocumentDB SQL 适用于 JSON 值，因此它可以处理三种形式的实体，而不是行和列。因此，该语言可让你在任意深度引用树的节点，如 Node1.Node2.Node3…..Nodem，这类似于引用 <table>.<column> 的两个部分引用的关系 SQL。
* 结构化查询语言适用于无架构的数据。因此，需要动态绑定类型系统。相同的表达式在不同文档上可能会产生不同的类型。查询的结果是一个有效的 JSON 值，但不保证它为固定的架构。
* DocumentDB 仅支持严格的 JSON 文档。这意味着类型系统和表达式仅限于处理 JSON 类型。有关更多详细信息，请参阅 [JSON specification（JSON 规范）](http://www.json.org/)。
* DocumentDB 集合是 JSON 文档的一个无构架容器。集合中，文档内和跨文档的数据实体的关系是按包含关系隐式捕获的，而不是按主键和外键关系。考虑到稍后将在本文中讨论文档内联接，因此这是一个值得注意的重要方面。

## DocumentDB 索引

在我们开始了解 DocumentDB SQL 语法前，值得先探索一下 DocumentDB 中的索引设计。

数据库索引的目的是在提供良好的吞吐量和低延迟的同时，以最小的资源消耗（如 CPU 和输入/输出）提供各种形式的查询。通常，为查询数据库选择正确的索引需要大量的计划和试验。此方法对数据不符合严格的架构并且快速发展的无架构数据库来说是一个挑战。

因此，在我们设计 DocumentDB 索引子系统时，我们设定了以下目标：

* 在无需架构的情况下索引文档：索引子系统不需要任何架构信息或对文档的架构做出任何假设。
* 支持高效、层次丰富的关系型查询：索引高效地支持 DocumentDB 查询语言，包括支持分层和关系投影。
* 在持续大量写入时支持一致的查询：对于使用一致的查询的高写入吞吐量工作负荷，在持续大量写入时可逐步、高效地联机更新索引。一致的索引更新对在用户配置文档服务的一致性级别进行查询来说是至关重要的。
* 支持多租户：在为跨租户的资源调控给定基于保留的模型的情况下，可以在为每个副本分配的系统资源（CPU、内存和每秒的输入/输出操作）的预算内执行索引更新。
* 存储效率：就成本效益而言，在磁盘上存储索引的开销是有限的，并且是可预测的。这一点非常重要，因为 DocumentDB 允许开发人员在索引开销与查询性能之间做出基于成本的权衡。

有关演示如何为集合配置索引策略的示例，请参阅 MSDN 上的 [DocumentDB samples（DocumentDB 示例）](https://github.com/Azure/azure-documentdb-net)。现在，让我们开始详细了解 DocumentDB SQL 语法。

## DocumentDB SQL 查询的基础知识

每个查询按 ANSI-SQL 标准由 SELECT 子句和可选的 FROM 和 WHERE 子句组成。通常，对于每个查询，已枚举 FROM 子句中的源。然后将 WHERE 子句中的筛选器应用到源以检索 JSON 文档的子集。最后，使用 SELECT 子句以投影选择列表中请求的 JSON 值。

SELECT [TOP <top\_expression>] <select\_list>   
[FROM <from\_specification>]   
[WHERE <filter\_condition>]  
[ORDER BY <sort\_specification]

## FROM 子句

FROM <from\_specification> 子句是可选的，除非稍后在查询中对源进行筛选或投影。此子句的目的在于指定必须对其执行查询的数据源。通常情况下，整个集合作为源，但可以改为指定集合的子集作为源。

一个类似 SELECT \* FROM Families 的查询指示整个家庭集合是要枚举的源。特殊标识符 ROOT 可以用来表示集合，而不使用集合名称来表示。以下列表包含每个查询需要强制执行的规则：

* 集合可以使用别名，如 SELECT f.id FROM Families AS f 或只需为 SELECT f.id FROM Families f。这里的 f 等效于 Families。AS 是可选的关键字，用于为标识符取别名。
* 请注意，一旦有了别名，则无法绑定原始的源。例如，由于再也无法解析标识符“Families”，因此 SELECT Families.id FROM Families f 在语法上无效。
* 所有需要引用的属性都必须是完全限定的。在没有遵循严格架构的情况下，会强制性地执行这一点以避免任何不确定的绑定。因此，由于未绑定 id 属性，因此 SELECT id FROM Families f 在语法上无效。

### 子文档

也可以将源缩小为更小的子集。例如，要在每个文档中仅枚举子树，则子根可能变成源，如下例所示。

**查询**

SELECT \*   
FROM Families.children

**结果**

[  
 [  
 {  
 "firstName": "Henriette Thaulow",  
 "gender": "female",  
 "grade": 5,  
 "pets": [  
 {  
 "givenName": "Fluffy"  
 }  
 ]  
 }  
 ],  
 [  
 {  
 "familyName": "Merriam",  
 "givenName": "Jesse",  
 "gender": "female",  
 "grade": 1  
 },  
 {  
 "familyName": "Miller",  
 "givenName": "Lisa",  
 "gender": "female",  
 "grade": 8  
 }  
 ]  
]

虽然上面的示例中使用数组作为源，但也可以使用对象作为源，如下例所示。在源中可以找到的任何有效的 JSON 值（不是未定义的）都将被视为包含在查询的结果中。如果一些家庭没有 address.state 值，则会将他们排除在查询结果之外。

**查询**

SELECT \*   
FROM Families.address.state

**结果**

[  
 "WA",   
 "NY"  
]

## WHERE 子句

WHERE 子句（**WHERE <filter\_condition>**）是可选的。它指定由源提供的 JSON 文档必须满足的条件，以便作为结果的一部分包含在内。任何 JSON 文档必须将指定的条件评估为“true”以作为结果。WHERE 子句由索引层使用，以确定可以作为结果的一部分的源文档的绝对最小子集。

以下查询请求包含值为 AndersenFamily 的名称属性的文档。任何其他不具有名称属性或值与 AndersenFamily 不匹配的文档则被排除在外。

**查询**

SELECT f.address  
FROM Families f   
WHERE f.id = "AndersenFamily"

**结果**

[{  
 "address": {  
 "state": "WA",   
 "county": "King",   
 "city": "seattle"  
 }  
}]

上面的示例演示了一个简单的等式查询。DocumentDB SQL 还支持各种标量表达式。最常使用的是二进制和一元表达式。来自源 JSON 对象的属性引用也是有效的表达式。

当前支持以下二进制运算符，它们可在查询中使用，如下例所示：

算术

+,-,\*,/,%

位

|、&、^、<<、>>、>>>（补零右移）

逻辑

AND、OR、NOT

比较

=、!=、<、>、<=、>=、<>

String

||（连接）

让我们查看一些使用二进制运算符的查询。

SELECT \*   
FROM Families.children[0] c  
WHERE c.grade % 2 = 1 -- matching grades == 5, 1  
  
SELECT \*   
FROM Families.children[0] c  
WHERE c.grade ^ 4 = 1 -- matching grades == 5  
  
SELECT \*  
FROM Families.children[0] c  
WHERE c.grade >= 5 -- matching grades == 5

也支持一元运算符 +、-、~ 和 NOT，它们可在查询中使用，如下例所示：

SELECT \*  
FROM Families.children[0] c  
WHERE NOT(c.grade = 5) -- matching grades == 1  
  
SELECT \*  
FROM Families.children[0] c  
WHERE (-c.grade = -5) -- matching grades == 5

除了二进制和一元运算符以外，还允许使用属性引用。例如，SELECT \* FROM Families f WHERE f.isRegistered 返回包含 isRegistered 属性的文档，其中的属性值等于 JSON true 值。任何其他值（false、null、Undefined、<number>、<string>、<object>、<array> 等等）都会导致源文档被排除在结果之外。

### 等式和比较运算符

下表显示了 DocumentDB SQL 中任意两个 JSON 类型之间等式比较的结果。

<tr>  
 <td valign="top">  
 <strong>Op</strong>  
 </td>  
 <td valign="top">  
 <strong>Undefined</strong>  
 </td>  
 <td valign="top">  
 <strong>Null</strong>  
 </td>  
 <td valign="top">  
 <strong>布尔</strong>  
 </td>  
 <td valign="top">  
 <strong>Number</strong>  
 </td>  
 <td valign="top">  
 <strong>String</strong>  
 </td>  
 <td valign="top">  
 <strong>对象</strong>  
 </td>  
 <td valign="top">  
 <strong>Array</strong>  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Undefined<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Null<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>确定</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Boolean<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>确定</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Number<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>确定</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>String<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>确定</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Object<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>确定</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Array<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>确定</strong>  
 </td>  
 </tr>

对于其他比较运算符（如 >、>=、!=、< 和 <=），以下规则适用：

* 跨类型比较结果为 Undefined。
* 两个对象或两个数组之间比较结果为 Undefined。

如果筛选器中标量表达式的结果为 Undefined，则相应的文档不会包含在结果中，因为 Undefined 在逻辑上不等于“True”。

### BETWEEN 关键字

你也可以使用 BETWEEN 关键字来对一定范围内的值（如在 ANSI SQL 中）进行快速查询。可对字符串或数字使用 BETWEEN。

例如，此查询返回在其中第一个子女的年级为 1-5 之间（包括 1 和 5）的所有家庭文档。

SELECT \*  
FROM Families.children[0] c  
WHERE c.grade BETWEEN 1 AND 5

与在 ANSI-SQL 中不同，你也可以使用 FROM 子句中的 BETWEEN 子句，如以下示例所示。

SELECT (c.grade BETWEEN 0 AND 10)  
FROM Families.children[0] c

了更快地执行查询，请记得创建索引策略，该策略对在 BETWEEN 子句中筛选的任何数值属性/路径使用范围索引类型。

在 DocumentDB 与在 ANSI SQL 中使用 BETWEEN 的主要不同之处在于你可以对混合类型的属性执行快速范围查询 – 例如，你可以在某些文档中将“grade”设置为数字 (5)，并在其他文档中将其设置为字符串（“grade4”）。在这些情况下（如在 JavaScript 中），在两种不同类型之间进行比较的结果为“undefined”，将会跳过文档。

### 逻辑（AND、OR 和 NOT）运算符

逻辑运算符对布尔值进行运算。下表显示了这些运算符的逻辑真值表。

|  |  |  |  |
| --- | --- | --- | --- |
| 或者 | True | False | Undefined |
| True | True | True | True |
| False | True | False | Undefined |
| Undefined | True | Undefined | Undefined |

|  |  |  |  |
| --- | --- | --- | --- |
| AND | True | False | Undefined |
| True | True | False | Undefined |
| False | False | False | False |
| Undefined | Undefined | False | Undefined |

|  |  |
| --- | --- |
| NOT |  |
| True | False |
| False | True |
| Undefined | Undefined |

### IN 关键字

IN 关键字可用于检查指定的值是否与列表中的任意值匹配。例如，此查询返回 ID 为“WakefieldFamily”或“AndersenFamily”的所有家庭文档。

SELECT \*  
FROM Families   
WHERE Families.id IN ('AndersenFamily', 'WakefieldFamily')

此示例返回状态为任何指定值的所有文档。

SELECT \*  
FROM Families   
WHERE Families.address.state IN ("NY", "WA", "CA", "PA", "OH", "OR", "MI", "WI", "MN", "FL")

IN 等效于链接多个 OR 子句，但是由于可以使用单一索引处理它，因此 DocumentDB 支持对 IN 子句内指定的参数数量实现更高[限制](/documentation/articles/documentdb-limits)。

### 三元 (?) 和联合 (??) 运算符

三元和联合运算符可以用于生成条件表达式，类似于常用的编程语言（如 C# 和 JavaScript）。

当动态构建新的 JSON 属性时，使用三元 (?) 运算符会非常方便。例如，现在你可以写入查询以将类级别（初学者/中级/高级）分类到用户可读的表单中，如下面所示。

SELECT (c.grade < 5)? "elementary": "other" AS gradeLevel   
 FROM Families.children[0] c

你也可以将调用嵌套到运算符，如以下查询中所示。

SELECT (c.grade < 5)? "elementary": ((c.grade < 9)? "junior": "high") AS gradeLevel   
FROM Families.children[0] c

如同使用其他查询运算符一样，如果任何文档中缺少条件表达式的引用属性，或者如果正在进行比较的类型不同，那么这些文档将会被排除在查询结果之外。

联合 (??) 运算符可用于有效地检查文档中是否存在属性（也被称为已定义）。这在对半结构化数据或混合类型的数据执行查询时很有用。例如，此查询返回“lastName”（如果存在）或“surname”（如果不存在）。

SELECT f.lastName ?? f.surname AS familyName  
FROM Families f

### 带引号的属性访问器

你也可以使用带引号的属性运算符 [] 访问属性。例如，SELECT c.grade 和 SELECT c["grade"] 是等效的。此语法在需要转义包含空格和特殊字符的属性或正好将相同的名称作为 SQL 关键字或保留字共享的属性时很有用。

SELECT f["lastName"]  
FROM Families f  
WHERE f["id"] = "AndersenFamily"

## SELECT 子句

SELECT 子句 (**SELECT <select\_list>**) 是强制性的，用于指定要从查询中检索的值，就如在 ANSI-SQL 中一样。将在源文档顶端上筛选出来的子集传递给投影阶段，在其中检索指定的 JSON 值并为每个传递给它的输出构造新的 JSON 对象。

下面的示例演示了典型的 SELECT 查询。

**查询**

SELECT f.address  
FROM Families f   
WHERE f.id = "AndersenFamily"

**结果**

[{  
 "address": {  
 "state": "WA",   
 "county": "King",   
 "city": "seattle"  
 }  
}]

### 嵌套属性

在下面的示例中，我们将投影两个嵌套的属性 f.address.state 和 f.address.city。

**查询**

SELECT f.address.state, f.address.city  
FROM Families f   
WHERE f.id = "AndersenFamily"

**结果**

[{  
 "state": "WA",   
 "city": "seattle"  
}]

投影也支持 JSON 表达式，如下例所示。

**查询**

SELECT { "state": f.address.state, "city": f.address.city, "name": f.id }  
FROM Families f   
WHERE f.id = "AndersenFamily"

**结果**

[{  
 "$1": {  
 "state": "WA",   
 "city": "seattle",   
 "name": "AndersenFamily"  
 }  
}]

让我们看看此处的 $1 角色。SELECT 子句需要创建 JSON 对象，并且由于没有提供任何密钥，因此我们使用以 $1 开头的隐式参数变量名。例如，此查询返回了两个隐式参数变量，标为 $1 和 $2。

**查询**

SELECT { "state": f.address.state, "city": f.address.city },   
 { "name": f.id }  
FROM Families f   
WHERE f.id = "AndersenFamily"

**结果**

[{  
 "$1": {  
 "state": "WA",   
 "city": "seattle"  
 },   
 "$2": {  
 "name": "AndersenFamily"  
 }  
}]

### 别名

现在让我们使用值的显示别名对上面的示例进行扩展。AS 是用于别名的关键字。请注意，在将第二个值投影为 NameInfo 时，它如显示的那样是可选的。

如果查询包含两个具有相同名称的属性，则必须使用别名以重命名其中一个属性或两个属性，以便可以在投影的结果中消除它们的歧义。

**查询**

SELECT   
 { "state": f.address.state, "city": f.address.city } AS AddressInfo,   
 { "name": f.id } NameInfo  
FROM Families f   
WHERE f.id = "AndersenFamily"

**结果**

[{  
 "AddressInfo": {  
 "state": "WA",   
 "city": "seattle"  
 },   
 "NameInfo": {  
 "name": "AndersenFamily"  
 }  
}]

### 标量表达式

除了属性引用之外，SELECT 子句还支持标量表达式，如常量、算术表达式和逻辑表达式等。例如，下面是一个简单的“Hello World”查询。

**查询**

SELECT "Hello World"

**结果**

[{  
 "$1": "Hello World"  
}]

以下是一个使用标量表达式的更复杂的示例。

**查询**

SELECT ((2 + 11 % 7)-2)/3

**结果**

[{  
 "$1": 1.33333  
}]

在下面的示例中，标量表达式的结果是布尔。

**查询**

SELECT f.address.city = f.address.state AS AreFromSameCityState  
FROM Families f

**结果**

[  
 {  
 "AreFromSameCityState": false  
 },   
 {  
 "AreFromSameCityState": true  
 }  
]

### 对象和数组创建

DocumentDB SQL 的另一个重要功能是数组/对象创建。请注意，在上一个示例中，我们已创建了一个新的 JSON 对象。同样，也可以构造数组，如下例所示。

**查询**

SELECT [f.address.city, f.address.state] AS CityState   
FROM Families f

**结果**

[  
 {  
 "CityState": [  
 "seattle",   
 "WA"  
 ]  
 },   
 {  
 "CityState": [  
 "NY",   
 "NY"  
 ]  
 }  
]

### VALUE 关键字

**VALUE** 关键字提供一种返回 JSON 值的方法。例如，下面所示的查询返回标量 "Hello World"，而不是 {$1: "Hello World"}。

**查询**

SELECT VALUE "Hello World"

**结果**

[  
 "Hello World"  
]

下面的查询在结果中返回不带 "address" 标签的 JSON 值。

**查询**

SELECT VALUE f.address  
FROM Families f

**结果**

[  
 {  
 "state": "WA",   
 "county": "King",   
 "city": "seattle"  
 },   
 {  
 "state": "NY",   
 "county": "Manhattan",   
 "city": "NY"  
 }  
]

下面的示例对此进行了扩展，以演示如何返回 JSON 基元值（JSON 树的叶级别）。

**查询**

SELECT VALUE f.address.state  
FROM Families f

**结果**

[  
 "WA",  
 "NY"  
]

### \* 运算符

支持使用特殊运算符 (\*) 按原样投影文档。在使用时，它必须仅为投影的字段。当类似 SELECT \* FROM Families f 的查询有效时，SELECT VALUE \* FROM Families f 和 SELECT \*, f.id FROM Families f 无效。

**查询**

SELECT \*   
FROM Families f   
WHERE f.id = "AndersenFamily"

**结果**

[{  
 "id": "AndersenFamily",  
 "lastName": "Andersen",  
 "parents": [  
 { "firstName": "Thomas" },  
 { "firstName": "Mary Kay"}  
 ],  
 "children": [  
 {  
 "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 }  
 ],  
 "address": { "state": "WA", "county": "King", "city": "seattle" },  
 "creationDate": 1431620472,  
 "isRegistered": true  
}]

### TOP 运算符

TOP 关键字可用于限制来自查询中的值的数量。当 TOP 与 ORDER BY 子句配合使用时，结果集被限制为有序值的前 N 个数；否则，它会返回未定义排序的结果中的前 N 数。在 SELECT 语句中，最佳做法始终使用带有 TOP 子句的 ORDER BY 子句。这是可预测指示受 TOP 影响的行的唯一方法。

**查询**

SELECT TOP 1 \*   
FROM Families f

**结果**

[{  
 "id": "AndersenFamily",  
 "lastName": "Andersen",  
 "parents": [  
 { "firstName": "Thomas" },  
 { "firstName": "Mary Kay"}  
 ],  
 "children": [  
 {  
 "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 }  
 ],  
 "address": { "state": "WA", "county": "King", "city": "seattle" },  
 "creationDate": 1431620472,  
 "isRegistered": true  
}]

可将 TOP 与常量值（如以上所示）或使用参数化查询的变量值配合使用。有关更多详细信息，请参阅下面的参数化查询。

## ORDER BY 子句

如同在 ANSI-SQL 中一样，在查询时可以包含可选的 Order By 子句。该子句可以包括可选的 ASC/DESC 参数以指定检索结果必须遵守的顺序。有关 Order By 的详细信息，请参阅 [DocumentDB Order By Walkthrough（DocumentDB Order By 演练）](/documentation/articles/documentdb-orderby)。

例如，下面是按居住城市的名称检索家庭的查询。

**查询**

SELECT f.id, f.address.city  
FROM Families f   
ORDER BY f.address.city

**结果**

[  
 {  
 "id": "WakefieldFamily",  
 "city": "NY"  
 },  
 {  
 "id": "AndersenFamily",  
 "city": "Seattle"   
 }  
]

下面的查询按创建日期检索家庭，该创建日期存储为表示纪元时间的数字，即，自 1970 年 1 月 1 日起经过的时间（秒）。

**查询**

SELECT f.id, f.creationDate  
FROM Families f   
ORDER BY f.creationDate DESC

**结果**

[  
 {  
 "id": "WakefieldFamily",  
 "creationDate": 1431620462  
 },  
 {  
 "id": "AndersenFamily",  
 "creationDate": 1431620472   
 }  
]

## 高级数据库概念和 SQL 查询

### 迭代

在 DocumentDB SQL 中通过 **IN** 关键字添加了一个新的构造，以便为遍历 JSON 数组提供支持。FROM 源为迭代提供支持。让我们从下面的示例开始：

**查询**

SELECT \*   
FROM Families.children

**结果**

[  
 [  
 {  
 "firstName": "Henriette Thaulow",   
 "gender": "female",   
 "grade": 5,   
 "pets": [{ "givenName": "Fluffy"}]  
 }  
 ],   
 [  
 {  
 "familyName": "Merriam",   
 "givenName": "Jesse",   
 "gender": "female",   
 "grade": 1  
 },   
 {  
 "familyName": "Miller",   
 "givenName": "Lisa",   
 "gender": "female",   
 "grade": 8  
 }  
 ]  
]

现在，让我们来看看对集合中的子女执行遍历的另一个查询。请注意输出数组中的差异。此示例拆分 children 并将结果合并为一个数组。

**查询**

SELECT \*   
FROM c IN Families.children

**结果**

[  
 {  
 "firstName": "Henriette Thaulow",  
 "gender": "female",  
 "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 },  
 {  
 "familyName": "Merriam",  
 "givenName": "Jesse",  
 "gender": "female",  
 "grade": 1  
 },  
 {  
 "familyName": "Miller",  
 "givenName": "Lisa",  
 "gender": "female",  
 "grade": 8  
 }  
]

这可用于对数组的单个实体执行进一步筛选，如下例所示。

**查询**

SELECT c.givenName  
FROM c IN Families.children  
WHERE c.grade = 8

**结果**

[{  
 "givenName": "Lisa"  
}]

### 联接

在关系数据库中，跨表联接的要求是非常重要的。设计规范化的架构是一项逻辑要求。与此相反，DocumentDB 则处理无架构文档的非规范化数据模型。这在逻辑上等效于“自联接”。

语言支持的语法是 JOIN JOIN …JOIN 。总体而言，此语法返回一组 **N** 元组（带有 **N** 个值的元组）。每个元组拥有通过对它们相应的集遍历所有集合别名所产生的值。换言之，这是加入联接的集的完整叉积。

下面的示例演示了 JOIN 子句是如何工作的。在下面的示例中，由于源中每个文档和空集的叉积为空，因此结果为空。

**查询**

SELECT f.id  
FROM Families f  
JOIN f.NonExistent

**结果**

[{  
}]

在下面的示例中，联接位于文档根和 children 子根之间。这是两个 JSON 对象之间的叉积。子女是一个数组的事实在 JOIN 中无效，因为我们正在处理的是子女数组的单一根。因此，由于每个带有数组的文档的叉积仅生成一个文档，因此结果仅包含两个结果。

**查询**

SELECT f.id  
FROM Families f  
JOIN f.children

**结果**

[  
 {  
 "id": "AndersenFamily"  
 },   
 {  
 "id": "WakefieldFamily"  
 }  
]

下面的示例演示了更传统的联接：

**查询**

SELECT f.id  
FROM Families f  
JOIN c IN f.children

**结果**

[  
 {  
 "id": "AndersenFamily"  
 },   
 {  
 "id": "WakefieldFamily"  
 },   
 {  
 "id": "WakefieldFamily"  
 }  
]

首先要注意的是 **JOIN** 子句的 from\_source 是迭代器。因此，在此示例中的流程如下：

* 在数组中扩展每个子元素 **c**。
* 应用具有文档 **f** 的根的叉积，这些文档具有已在第一步中合并的每个子元素 **c**。
* 最后，单独投影根对象 **f** 名称属性。

第一个文档 (AndersenFamily) 仅包含一个子元素，因此结果集仅包含与此文档对应的单个对象。第二个文档 (WakefieldFamily) 包含两个子元素。因此，叉积会为每个子女生成单独的对象，从而产生两个对象，每个对象分别与此文档对应。请注意，这两个文档中的根字段将会是相同的，就如你在叉积中所预期的一样。

JOIN 真正实用的地方通过以其他方式难以投影的形式基于叉积生成元组。此外，就如我们将会在下面的示例中看见的那样，你可以对元组组合进行筛选，总体上由用户选择元组满足的条件。

**查询**

SELECT   
 f.id AS familyName,  
 c.givenName AS childGivenName,  
 c.firstName AS childFirstName,  
 p.givenName AS petName   
FROM Families f   
JOIN c IN f.children   
JOIN p IN c.pets

**结果**

[  
 {  
 "familyName": "AndersenFamily",   
 "childFirstName": "Henriette Thaulow",   
 "petName": "Fluffy"  
 },   
 {  
 "familyName": "WakefieldFamily",   
 "childGivenName": "Jesse",   
 "petName": "Goofy"  
 },   
 {  
 "familyName": "WakefieldFamily",   
 "childGivenName": "Jesse",   
 "petName": "Shadow"  
 }  
]

此示例是前面示例的自然扩展，且执行双联接。因此，可将叉积视为下面的伪代码。

for-each(Family f in Families)  
{   
 for-each(Child c in f.children)  
 {  
 for-each(Pet p in c.pets)  
 {  
 return (Tuple(f.id AS familyName,   
 c.givenName AS childGivenName,   
 c.firstName AS childFirstName,  
 p.givenName AS petName));  
 }  
 }  
}

AndersenFamily 有一个拥有一只宠物的孩子。因此，叉积从此家庭中生成一行 (1*1*1)。尽管 WakefieldFamily 有两个孩子，但只有一个孩子“Jesse”拥有宠物。Jesse 拥有 2 只宠物。因此叉积从此家庭中生成 1*1*2 = 2 行。

在接下来的示例中，可以根据 pet 进行额外的筛选。这排除了宠物名称不是“Shadow”的所有元组。请注意，我们能够从数组中生成元组，根据元组的任意元素进行筛选以及投影元素的任何组合。

**查询**

SELECT   
 f.id AS familyName,  
 c.givenName AS childGivenName,  
 c.firstName AS childFirstName,  
 p.givenName AS petName   
FROM Families f   
JOIN c IN f.children   
JOIN p IN c.pets  
WHERE p.givenName = "Shadow"

**结果**

[  
 {  
 "familyName": "WakefieldFamily",   
 "childGivenName": "Jesse",   
 "petName": "Shadow"  
 }  
]

## JavaScript 集成

DocumentDB 根据存储过程和触发器，为对集合直接执行基于 JavaScript 的应用程序逻辑提供编程模型。这允许以下两种情况：

* 借助直接在数据库引擎内深度集成 JavaScript 运行时，能够对集合中的文档执行高性能事务性 CRUD 操作和查询。
* 控制流、变量范围和分配的自然建模和将异常处理基元与数据库事务集成。有关 DocumentDB 对 JavaScript 集成的支持的更多详细信息，请参阅《JavaScript 服务器端可编程性》文档。

### 用户定义的函数 (UDF)

除了本文中已定义的类型外，DocumentDB SQL 也对用户定义的函数 (UDF) 提供支持。具体而言，支持标量 UDF，开发人员可在其中传入零个或许多参数并返回单个参数结果。检查每个参数是否是合法的 JSON 值。

扩展 DocumentDB SQL 语法以支持使用这些用户定义的函数的自定义应用程序逻辑。可使用 DocumentDB 注册 UDF，然后作为 SQL 查询的一部分引用这些函数。事实上，UDF 经过精心设计，可由查询调用。作为此选择的必然结果，UDF 不能访问其他 JavaScript 类型（存储过程和触发器）所拥有的上下文对象。由于查询以只读方式执行，因此它们可以在主要或次要副本上运行。因此，UDF 设计为在次要副本上运行，这与其他 JavaScript 类型不同。

以下是如何在 DocumentDB 数据库中（特别是在文档集合下）注册 UDF 的示例。

UserDefinedFunction regexMatchUdf = new UserDefinedFunction  
 {  
 Id = "REGEX\_MATCH",  
 Body = @"function (input, pattern) {   
 return input.match(pattern) !== null;  
 };",  
 };  
   
 UserDefinedFunction createdUdf = client.CreateUserDefinedFunctionAsync(  
 UriFactory.CreateDocumentCollectionUri("testdb", "families"),   
 regexMatchUdf).Result;

之前的示例创建了名称为 REGEX\_MATCH 的 UDF。它接受两个 JSON 字符串值 input 和 pattern，并且使用 JavaScript 的 string.match() 函数检查第一个值是否与第二个值中指定的模式匹配。

现在，我们可以在投影中的查询中使用此 UDF。在从查询内调用时，必须用区分大小写的前缀“udf.”限定 UDF。

[AZURE.NOTE] 在 2015 年 3 月 17 日之前，DocumentDB 支持无需“udf.”前缀的 UDF 调用，例如 SELECT REGEX\_MATCH()。已弃用此调用模式。

**查询**

SELECT udf.REGEX\_MATCH(Families.address.city, ".\*eattle")  
FROM Families

**结果**

[  
 {  
 "$1": true  
 },   
 {  
 "$1": false  
 }  
]

也可在 UDF 中使用筛选器，这同样要使用“udf.”前缀进行限定，如下例所示：

**查询**

SELECT Families.id, Families.address.city  
FROM Families  
WHERE udf.REGEX\_MATCH(Families.address.city, ".\*eattle")

**结果**

[{  
 "id": "AndersenFamily",  
 "city": "Seattle"  
}]

从本质上来说，UDF 是有效的标量表达式且可在投影和筛选器中使用。

要扩展 UDF 的功能，让我们看看有关使用条件逻辑的一个示例：

UserDefinedFunction seaLevelUdf = new UserDefinedFunction()  
 {  
 Id = "SEALEVEL",  
 Body = @"function(city) {  
 switch (city) {  
 case 'seattle':  
 return 520;  
 case 'NY':  
 return 410;  
 case 'Chicago':  
 return 673;  
 default:  
 return -1;  
 }"  
 };  
  
 UserDefinedFunction createdUdf = await client.CreateUserDefinedFunctionAsync(  
 UriFactory.CreateDocumentCollectionUri("testdb", "families"),   
 seaLevelUdf);

以下是使用 UDF 的一个示例。

**查询**

SELECT f.address.city, udf.SEALEVEL(f.address.city) AS seaLevel  
FROM Families f

**结果**

[  
 {  
 "city": "seattle",   
 "seaLevel": 520  
 },   
 {  
 "city": "NY",   
 "seaLevel": 410  
 }  
]

如之前的示例所的，UDF 使用 DocumentDB SQL 集成 JavaScript 语言的功能以通过丰富的可编程接口执行复杂的过程，并在内置 JavaScript 运行时功能的帮助下，执行条件逻辑。

DocumentDB SQL 在处理 UDF 当前阶段（WHERE 子句或 SELECT 子句），为源中每个文档的 UDF 提供参数。将结果无缝地纳入总体执行管道中。如果由 UDF 参数引用的属性在 JSON 值中不可用，则参数将被视为未定义，因此会完全跳过 UDF 调用。同样，如果未定义 UDF的结果，则它不会包含在结果中。

总而言之，UDF 是作为查询的一部分处理复杂业务逻辑重要的工具。

### 运算符评估

DocumentDB 是一个 JSON 数据库，与 JavaScript 运算符以及其评估语义具有许多相似之处。就 JSON 支持而言，当 DocumentDB 尝试保留 JavaScript 语义时，操作评估在某些实例中有所偏移。

在 DocumentDB SQL 中，与在传统 SQL 中不同，在实际从数据库中检索出值之前，值类型经常是未知的。为了高效执行查询，大多数运算符具有严格的类型要求。

不同于 JavaScript，DocumentDB SQL 不会执行隐式转换。例如，类似 SELECT \* FROM Person p WHERE p.Age = 21 的查询与包含值为 21 的 Age 属性的文档相匹配。任何其他 Age 属性与字符串“21”匹配或包含其他无数可能的变量（“021”、“21.0”、“0021”和“00021”等等）的文档则不匹配。这与 JavaScript 相反，在 JavaScript 中，字符串会隐式转换为数字（基于运算符 ex: ==）。此选择对于 DocumentDB SQL 中的高效索引匹配至关重要。

## 参数化 SQL 查询

DocumentDB 支持使用带有常用的 @ 表示法的参数进行查询。参数化 SQL 为用户输入提供可靠的处理和转义，可防止通过 SQL 注入发生意外的数据泄露。

例如，你可以编写一个将姓氏和省/自治区地址作为参数的查询，然后基于用户输入针对姓氏和省/自治区地址执行此查询。

SELECT \*   
FROM Families f  
WHERE f.lastName = @lastName AND f.address.state = @addressState

然后，可以将此请求作为参数化 JSON 查询发送到 DocumentDB，如下所示。

{   
 "query": "SELECT \* FROM Families f WHERE f.lastName = @lastName AND f.address.state = @addressState",   
 "parameters": [   
 {"name": "@lastName", "value": "Wakefield"},   
 {"name": "@addressState", "value": "NY"},   
 ]   
}

可以使用参数化查询设置 TOP 的参数，如下所示。

{   
 "query": "SELECT TOP @n \* FROM Families",   
 "parameters": [   
 {"name": "@n", "value": 10},   
 ]   
}

参数值可以为任何有效的 JSON（字符串、数字、布尔、null，甚至是数组或嵌套的 JSON）。此外，由于 DocumentDB 是无架构的，因此未针对任何类型对参数进行验证。

## 内置函数

DocumentDB 还支持使用许多内置函数进行常见操作，这些函数可以在查询（如用户定义的函数 (UDF)）中使用。

数学函数

ABS、CEILING、EXP、FLOOR、LOG、LOG10、POWER、ROUND、SIGN、SQRT、SQUARE、TRUNC、ACOS、ASIN、ATAN、ATN2、COS、COT、DEGREES、PI、RADIANS、SIN 和 TAN

类型检查函数

IS\_ARRAY、IS\_BOOL、IS\_NULL、IS\_NUMBER、IS\_OBJECT、IS\_STRING、IS\_DEFINED 和 IS\_PRIMITIVE

字符串函数

CONCAT、CONTAINS、ENDSWITH、INDEX\_OF、LEFT、LENGTH、LOWER、LTRIM、REPLACE、REPLICATE、REVERSE、RIGHT、RTRIM、STARTSWITH、SUBSTRING 和 UPPER

数组函数

ARRAY\_CONCAT、ARRAY\_CONTAINS、ARRAY\_LENGTH 和 ARRAY\_SLICE

空间函数

ST\_DISTANCE、ST\_WITHIN、ST\_ISVALID 和 ST\_ISVALIDDETAILED

如果你当前正使用内置函数适用的用户定义的函数 (UDF)，那么应使用相应的内置函数，因为它会更快更有效地运行。

### 数学函数

每个数学函数均执行一个计算，通常基于作为参数提供的输出值，并返回数值。以下是受支持的内置数学函数表。

使用情况

说明

ABS (num\_expr)

返回指定数值表达式的绝对（正）值。

CEILING (num\_expr)

返回大于或等于指定数值表达式的最小整数值。

FLOOR (num\_expr)

返回小于或等于指定数值表达式的最大整数。

EXP (num\_expr)

返回指定数值表达式的指数。

LOG (num\_expr [,base])

返回指定数值表达式的自然对数，或使用指定底数的对数

LOG10 (num\_expr)

返回指定数值表达式以 10 为底的对数值。

ROUND (num\_expr)

返回一个数值，四舍五入到最接近的整数值。

TRUNC (num\_expr)

返回一个数值，截断到最接近的整数值。

SQRT (num\_expr)

返回指定数值表达式的平方根。

SQUARE (num\_expr)

返回指定数值表达式的平方。

POWER (num\_expr, num\_expr)

返回指定数值表达式相对指定值的幂。

SIGN (num\_expr)

返回指定数值表达式的符号值 (-1, 0, 1)。

ACOS (num\_expr)

返回角度（弧度），其余弦是指定的数值表达式；也被称为反余弦。

ASIN (num\_expr)

返回角度（弧度），其正弦是指定的数值表达式。也被称为反正弦。

ATAN (num\_expr)

返回角度（弧度），其正切是指定的数值表达式。这也被称为反正切。

ATN2 (num\_expr)

返回正 x 轴与射线（原点到点 (y, x)）之间的角度（弧度），其中 x 和 y 是两个指定的浮点表达式的值。

COS (num\_expr)

返回指定表达式中指定角度的三角余弦（弧度）。

COT (num\_expr)

返回指定数值表达式中指定角度的三角余切。

DEGREES (num\_expr)

返回指定角度（弧度）的相应角度（度）。

PI ()

返回 PI 的常数值。

RADIANS (num\_expr)

返回输入的数值表达式（度）的弧度。

SIN (num\_expr)

返回指定表达式中指定角度的三角正弦（弧度）。

TAN (num\_expr)

返回指定表达式中输入表达式的正切。

例如，你现在可以运行以下查询：

**查询**

SELECT VALUE ABS(-4)

**结果**

[4]

与 ANSI SQL 相比，DocumentDB 的函数主要的差异在于它们被设计为可良好地适用于无架构和混合架构数据。例如，如果你有一个缺少 Size 属性或有一个非数值的值（如“unknown”）的文档，那么会跳过该文档，而不是返回错误。

### 类型检查函数

类型检查函数使你能够检查 SQL 查询内表达式的类型。当类型是变量或未知时，可使用类型检查函数动态确定文档内属性的类型。以下是受支持的内置类型检查函数表。

使用情况

说明

IS\_ARRAY (expr)

返回一个布尔值，它指示值的类型是否为数组。

IS\_BOOL (expr)

返回一个布尔值，它指示值的类型是否为布尔。

IS\_NULL (expr)

返回一个布尔值，它指示值的类型是否为 null。

IS\_NUMBER (expr)

返回一个布尔值，它指示值的类型是否为数字。

IS\_OBJECT (expr)

返回一个布尔值，它指示值的类型是否为 JSON 对象。

IS\_STRING (expr)

返回一个布尔值，它指示值的类型是否为字符串。

IS\_DEFINED (expr)

返回一个布尔，它指示属性是否已经分配了值。

IS\_PRIMITIVE (expr)

返回一个布尔值，它指示值的类型是否为字符串、数字、布尔或 null。

使用这些函数，你现在可以运行以下查询：

**查询**

SELECT VALUE IS\_NUMBER(-4)

**结果**

[true]

### 字符串函数

下面的标量函数对字符串输入值执行操作，并返回字符串、数值或布尔值。以下是内置字符串函数表：

|  |  |
| --- | --- |
| 使用情况 | 说明 |
| [LENGTH (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_length) | 返回指定字符串的字符数 |
| [CONCAT (str\_expr, str\_expr [, str\_expr])](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_concat) | 返回一个字符串，该字符串是连接两个或多个字符串值的结果。 |
| [SUBSTRING (str\_expr, num\_expr, num\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_substring) | 返回部分字符串表达式。 |
| [STARTSWITH (str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_startswith) | 返回一个布尔值，该值指示第一个字符串表达式是否以第二个字符串表达式结尾 |
| [ENDSWITH (str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_endswith) | 返回一个布尔值，该值指示第一个字符串表达式是否以第二个字符串表达式结尾 |
| [CONTAINS (str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_contains) | 返回一个布尔值，该值指示第一个字符串表达式是否包含第二个字符串表达式。 |
| [INDEX\_OF (str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_index_of) | 返回第一个指定的字符串表达式中第一次出现第二个字符串表达式的起始位置，如果未找到字符串，则返回 -1。 |
| [LEFT (str\_expr, num\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_left) | 返回具有指定字符数的字符串的左侧部分。 |
| [RIGHT (str\_expr, num\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_right) | 返回具有指定字符数的字符串的右侧部分。 |
| [LTRIM (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_ltrim) | 返回删除前导空格后的字符串表达式。 |
| [RTRIM (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_rtrim) | 返回截断所有尾随空格后的字符串表达式。 |
| [LOWER (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_lower) | 返回在将大写字符数据转换为小写后的字符串表达式。 |
| [UPPER (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_upper) | 返回在将小写字符数据转换为大写后的字符串表达式。 |
| [REPLACE (str\_expr, str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_replace) | 将出现的所有指定字符串值替换为另一个字符串值。 |
| [REPLICATE (str\_expr, num\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_replicate) | 将一个字符串值重复指定的次数。 |
| [REVERSE (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_reverse) | 返回字符串值的逆序排序形式。 |

使用这些函数，你现在可以运行以下查询。例如，你可以返回大写形式的家庭名称，如下所示：

**查询**

SELECT VALUE UPPER(Families.id)  
FROM Families

**结果**

[  
 "WAKEFIELDFAMILY",   
 "ANDERSENFAMILY"  
]

或如此示例中一样连接字符串：

**查询**

SELECT Families.id, CONCAT(Families.address.city, ",", Families.address.state) AS location  
FROM Families

**结果**

[{  
 "id": "WakefieldFamily",  
 "location": "NY,NY"  
},  
{  
 "id": "AndersenFamily",  
 "location": "seattle,WA"  
}]

也可在 WHERE 子句中使用字符串函数来筛选结果，如下例所示：

**查询**

SELECT Families.id, Families.address.city  
FROM Families  
WHERE STARTSWITH(Families.id, "Wakefield")

**结果**

[{  
 "id": "WakefieldFamily",  
 "city": "NY"  
}]

### 数组函数

下面的标量函数对数组输入值执行操作，并返回数值、布尔值或数组值。以下是内置数组函数表：

|  |  |
| --- | --- |
| 使用情况 | 说明 |
| [ARRAY\_LENGTH (arr\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_array_length) | 返回指定数组表达式的元素数。 |
| [ARRAY\_CONCAT (arr\_expr, arr\_expr [, arr\_expr])](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_array_concat) | 返回一个数组，该数组是连接两个或更多数组值的结果。 |
| [ARRAY\_CONTAINS (arr\_expr, expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_array_contains) | 返回一个布尔，它指示数组是否包含指定的值。 |
| [ARRAY\_SLICE (arr\_expr, num\_expr [, num\_expr])](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_array_slice) | 返回部分数组表达式。 |

数组函数可用于在 JSON 内操纵数组。例如，下面的查询返回其中一位父母是“Robin Wakefield”的所有文档。

**查询**

SELECT Families.id   
FROM Families   
WHERE ARRAY\_CONTAINS(Families.parents, { givenName: "Robin", familyName: "Wakefield" })

**结果**

[{  
 "id": "WakefieldFamily"  
}]

以下是使用 ARRAY\_LENGTH 获取每个家庭的子女数的另一个示例。

**查询**

SELECT Families.id, ARRAY\_LENGTH(Families.children) AS numberOfChildren  
FROM Families

**结果**

[{  
 "id": "WakefieldFamily",  
 "numberOfChildren": 2  
},  
{  
 "id": "AndersenFamily",  
 "numberOfChildren": 1  
}]

### 空间函数

DocumentDB 支持以下用于查询地理空间的开放地理空间信息联盟 (OGC) 内置函数。有关 DocumentDB 中地理支持的更多详细信息，请参阅 [Working with geospatial data in Azure DocumentDB（在 Azure DocumentDB 中使用地理数据）](/documentation/articles/documentdb-geospatial)。

使用情况

说明

ST\_DISTANCE (point\_expr、point\_expr)

返回两个 GeoJSON 点表达式之间的距离。

ST\_WITHIN (point\_expr、polygon\_expr)

返回布尔表达式，该表达式指示第一个参数中指定的 GeoJSON 点是否位于第二个参数中的 GeoJSON 多边形内。

ST\_ISVALID

返回一个布尔值，该值指示指定的 GeoJSON 点或多边形表达式是否有效。

ST\_ISVALIDDETAILED

如果指定的 GeoJSON 点或多边形表达式有效，则返回包含布尔值的 JSON 值；如果无效，则额外加上作为字符串值的原因。

空间函数可用于对空间数据执行邻近查询。例如，以下查询使用 ST\_DISTANCE 内置函数返回所有家族文档，且这些文档在指定位置的 30 公里内。

**查询**

SELECT f.id   
FROM Families f   
WHERE ST\_DISTANCE(f.location, {'type': 'Point', 'coordinates':[31.9, -4.8]}) < 30000

**结果**

[{  
 "id": "WakefieldFamily"  
}]

如果在你的索引策略中包含空间索引，则将通过索引有效地进行“距离查询”。有关空间索引的更多详细信息，请参阅以下章节。如果你没有指定路径的空间索引，仍然可以通过指定 x-ms-documentdb-query-enable-scan 请求标头（其值设置为“true”）执行空间查询。在 .NET 中，可以通过将可选的 **FeedOptions** 参数传递到 [EnableScanInQuery](https://msdn.microsoft.com/library/microsoft.azure.documents.client.feedoptions.enablescaninquery.aspx#P:Microsoft.Azure.Documents.Client.FeedOptions.EnableScanInQuery) 设置为 true 的查询来完成此操作。

ST\_WITHIN 可用于检查点是否在多边形内。多边形通常用来表示边界，例如邮政编码、省/自治区边界或自然构成物。再次说明，如果在你的索引策略中包含空间索引，则将通过索引有效地进行“within”查询。

ST\_WITHIN 中的多边形参数只能包含单个环形，也就是说，多边形本身不能包含洞。有关 ST\_WITHIN 查询的多边形中允许的点数目上限，请查看 [DocumentDB 限制](/documentation/articles/documentdb-limits)。

**查询**

SELECT \*   
FROM Families f   
WHERE ST\_WITHIN(f.location, {  
 'type':'Polygon',   
 'coordinates': [[[31.8, -5], [32, -5], [32, -4.7], [31.8, -4.7], [31.8, -5]]]  
})

**结果**

[{  
 "id": "WakefieldFamily",  
}]

[AZURE.NOTE] 与 DocumentDB 查询中不匹配类型的工作方式类似，如果任一参数中指定的位置值格式不正确或无效，则会评估为**未定义**，并且会在查询结果中跳过已评估的文档。如果你的查询没有返回任何结果，请运行 ST\_ISVALIDDETAILED 进行调试，以了解空间类型无效的原因。

ST\_ISVALID 和 ST\_ISVALIDDETAILED 可用来检查空间对象是否有效。例如，下列查询检查纬度值 (-132.8) 超出范围的点的有效性。ST\_ISVALID 仅返回一个布尔值，ST\_ISVALIDDETAILED 则返回布尔值和字符串，字符串中包含被视为无效的原因。

**查询**

SELECT ST\_ISVALID({ "type": "Point", "coordinates": [31.9, -132.8] })

**结果**

[{  
 "$1": false  
}]

这些函数也可以用来验证多边形。例如，在这里我们使用 ST\_ISVALIDDETAILED 来验证未闭合的多边形。

**查询**

SELECT ST\_ISVALIDDETAILED({ "type": "Polygon", "coordinates": [[   
 [ 31.8, -5 ], [ 31.8, -4.7 ], [ 32, -4.7 ], [ 32, -5 ]   
 ]]})

**结果**

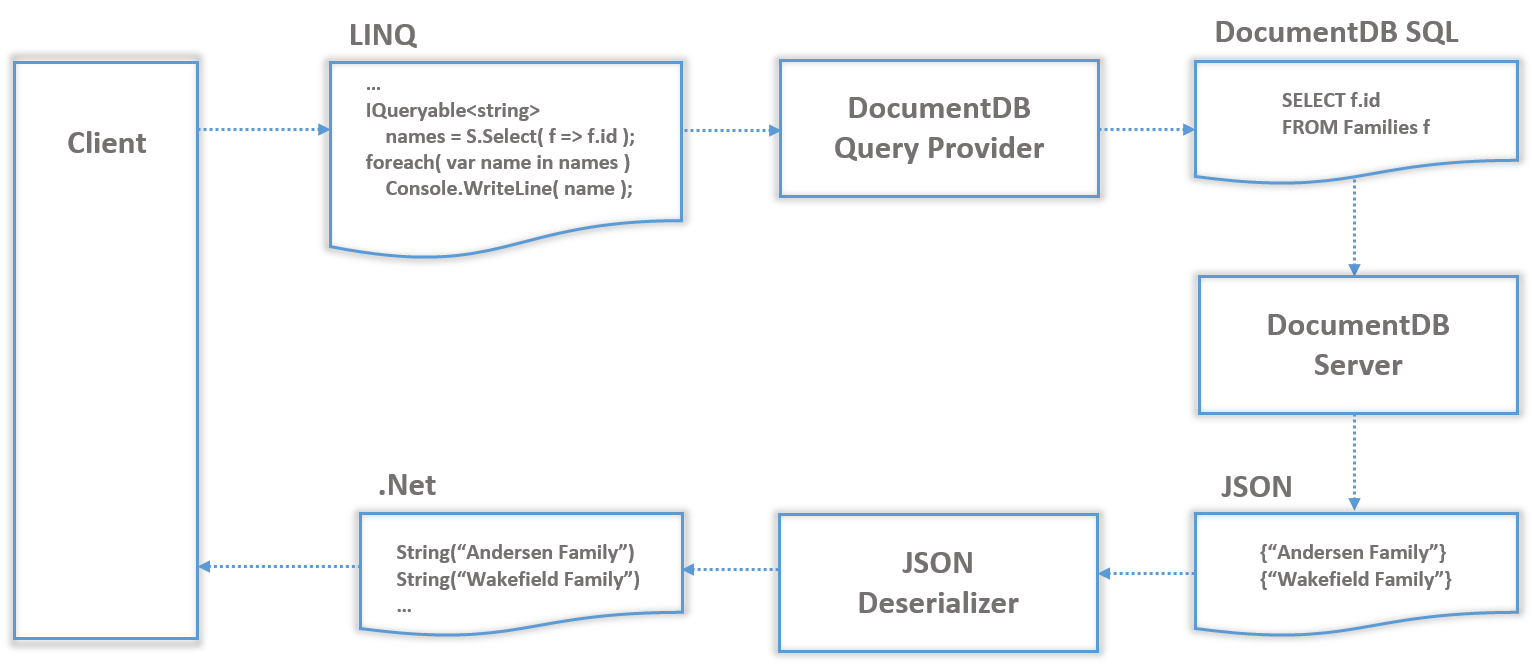
[{  
 "$1": {   
 "valid": false,   
 "reason": "The Polygon input is not valid because the start and end points of the ring number 1 are not the same. Each ring of a polygon must have the same start and end points."   
 }  
}]

这会完成空间函数和 DocumentDB 的 SQL 语法。现在，让我们来看看 LINQ 查询的工作方式，以及它如何与我们目前为止所看到的语法进行交互。

## LINQ 到 DocumentDB SQL

LINQ 是一个 .NET 编程模型，它将计算表示为对对象流的查询。DocumentDB 提供一个客户端库，通过促进 JSON 与 .NET 对象之间的转换，以及从 LINQ 查询的子集到 DocumentDB 查询的映射，来与 LINQ 进行交互。

下面的图片展示了使用 DocumentDB 支持 LINQ 查询的体系结构。使用 DocumentDB 客户端，开发人员可以创建直接查询 DocumentDB 查询提供程序的 **IQueryable** 对象，该提供程序随后会将 LINQ 查询转换为 DocumentDB 查询。该查询会被传递到 DocumentDB 服务器以检索一组 JSON 格式的结果。在客户端上，返回的结果会反序列化为 .NET 对象的流。



支持使用 DocumentDB 的 LINQ 查询的体系结构 - SQL语法、JSON 查询语言、数据库概念和 SQL 查询

### .NET 和 JSON 映射

.NET 对象与 JSON 文档之间的映射是自然的 - 每个数据成员字段映射到 JSON 对象，其中字段名称映射到对象的“key”部分，并且“value”部分以递归方式映射到该对象的值部分。请考虑以下示例。创建的 Family 对象会映射到 JSON 文档，如下所示。反之亦然，JSON 文档会映射回 .NET 对象。

**C# 类**

public class Family  
{  
 [JsonProperty(PropertyName="id")]  
 public string Id;  
 public Parent[] parents;  
 public Child[] children;  
 public bool isRegistered;  
};  
  
public struct Parent  
{  
 public string familyName;  
 public string givenName;  
};  
  
public class Child  
{  
 public string familyName;  
 public string givenName;  
 public string gender;  
 public int grade;  
 public List<Pet> pets;  
};  
  
public class Pet  
{  
 public string givenName;  
};  
  
public class Address  
{  
 public string state;  
 public string county;  
 public string city;  
};  
  
// Create a Family object.  
Parent mother = new Parent { familyName= "Wakefield", givenName="Robin" };  
Parent father = new Parent { familyName = "Miller", givenName = "Ben" };  
Child child = new Child { familyName="Merriam", givenName="Jesse", gender="female", grade=1 };  
Pet pet = new Pet { givenName = "Fluffy" };  
Address address = new Address { state = "NY", county = "Manhattan", city = "NY" };  
Family family = new Family { Id = "WakefieldFamily", parents = new Parent [] { mother, father}, children = new Child[] { child }, isRegistered = false };

**JSON**

{  
 "id": "WakefieldFamily",  
 "parents": [  
 { "familyName": "Wakefield", "givenName": "Robin" },  
 { "familyName": "Miller", "givenName": "Ben" }  
 ],  
 "children": [  
 {  
 "familyName": "Merriam",   
 "givenName": "Jesse",   
 "gender": "female",   
 "grade": 1,  
 "pets": [  
 { "givenName": "Goofy" },  
 { "givenName": "Shadow" }  
 ]  
 },  
 {   
 "familyName": "Miller",   
 "givenName": "Lisa",   
 "gender": "female",   
 "grade": 8   
 }  
 ],  
 "address": { "state": "NY", "county": "Manhattan", "city": "NY" },  
 "isRegistered": false  
};

### LINQ 到 SQL 转换

DocumentDB 查询提供程序执行从 LINQ 查询到 DocumentDB SQL 查询的最有效映射。在以下描述中，我们假设读者对 LINQ 已经有了一个基本的了解。

首先，对于类型系统，我们支持所有 JSON 基元类型 – 数值类型、布尔、字符串和 null。仅支持这些 JSON 类型。支持以下的标量表达式。

* 常量值 – 这些包含评估查询时基元数据类型的常量值。
* 属性/数组索引表达式 – 这些表达式引用对象或数组元素的属性。
* family.Id;  
  family.children[0].familyName;  
  family.children[0].grade;  
  family.children[n].grade; //n is an int variable
* 算术表达式 - 这些表达式包含数值和布尔值上的常用算术表达式。有关完整列表，请参阅 SQL 规范。
* 2 \* family.children[0].grade;  
  x + y;
* 字符串比较表达式 - 这些表达式包含字符串值与某些常量字符串值的比较。
* mother.familyName == "Smith";  
  child.givenName == s; //s is a string variable
* 对象/数组创建表达式 - 这些表达式返回复合值类型或匿名类型的对象，或此类对象组成的数组。可以嵌套这些值。
* new Parent { familyName = "Smith", givenName = "Joe" };  
  new { first = 1, second = 2 }; //an anonymous type with 2 fields   
  new int[] { 3, child.grade, 5 };

### 支持的 LINQ 运算符列表

以下是 DocumentDB .NET SDK 中包含的 LINQ 提供程序支持的 LINQ 运算符列表。

* **Select**：投影转换为 SQL SELECT（包括对象构造）
* **Where**：筛选器转换为 SQL WHERE，且支持 &&、|| 和 ! 与 SQL 运算符之间的转换
* **SelectMany**：允许将数组展开到 SQL JOIN 子句。可以用于链/嵌套表达式以对数组元素进行筛选
* **OrderBy 和 OrderByDescending**：转换为 ORDER BY 升序/降序：
* **CompareTo**：转换为范围比较。通常用于字符串，因为它们在 .NET 中不可进行比较
* **Take**：转换为 SQL TOP，用于限制查询中的结果
* **数学函数**：支持从 .NET 的 Abs、Acos、Asin、Atan、Ceiling、Cos、Exp、Floor、Log、Log10、Pow、Round、Sign、Sin、Sqrt、Tan 和 Truncate 转换为等效的 SQL 内置函数。
* **字符串函数**：支持从 .NET 的 Concat、Contains、EndsWith、IndexOf、Count、ToLower、TrimStart、Replace、Reverse、TrimEnd、StartsWith、SubString 和 ToUpper 转换为等效的 SQL 内置函数。
* **数组函数**：支持从 .NET 的 Concat、Contains 和 Count 转换为等效的 SQL 内置函数。
* **地理空间扩展函数**：支持从 stub 方法 Distance、Within、IsValid 和 IsValidDetailed 转换为等效的 SQL 内置函数。
* **用户定义的函数扩展函数**：支持从 stub 方法 UserDefinedFunctionProvider.Invoke 转换为相应的用户的定义函数。
* **其他**：支持联合与条件运算符的转换。可以根据上下文将 Contains 转换为字符串 CONTAINS、ARRAY\_CONTAINS 或 SQL IN。

### SQL 查询运算符

以下示例演示了一些标准 LINQ 查询运算符是如何转换为 DocumentDB 查询的。

#### Select 运算符

语法为 input.Select(x => f(x))，其中 f 是一个标量表达式。

**LINQ Lambda 表达式**

input.Select(family => family.parents[0].familyName);

**SQL**

SELECT VALUE f.parents[0].familyName  
FROM Families f

**LINQ Lambda 表达式**

input.Select(family => family.children[0].grade + c); // c is an int variable

**SQL**

SELECT VALUE f.children[0].grade + c  
FROM Families f

**LINQ Lambda 表达式**

input.Select(family => new  
{  
 name = family.children[0].familyName,  
 grade = family.children[0].grade + 3  
});

**SQL**

SELECT VALUE {"name":f.children[0].familyName,   
 "grade": f.children[0].grade + 3 }  
FROM Families f

#### SelectMany 运算符

语法为 input.SelectMany(x => f(x))，其中 f 是返回集合类型的标量表达式。

**LINQ Lambda 表达式**

input.SelectMany(family => family.children);

**SQL**

SELECT VALUE child  
FROM child IN Families.children

#### Where 运算符

语法为 input.Where(x => f(x))，其中 f 是返回布尔值的标量表达式。

**LINQ Lambda 表达式**

input.Where(family=> family.parents[0].familyName == "Smith");

**SQL**

SELECT \*  
FROM Families f  
WHERE f.parents[0].familyName = "Smith"

**LINQ Lambda 表达式**

input.Where(  
 family => family.parents[0].familyName == "Smith" &&   
 family.children[0].grade < 3);

**SQL**

SELECT \*  
FROM Families f  
WHERE f.parents[0].familyName = "Smith"  
AND f.children[0].grade < 3

### 复合 SQL 查询

可以将以上运算符组合在一起，形成功能更强大的查询。由于 DocumentDB 支持嵌套的集合，因此可以连接或嵌套运算符组合。

#### 串联

语法为 input(.|.SelectMany())(.Select()|.Where())\*。串联的查询可以可选的 SelectMany 查询开始，后接多个 Select 或 Where 运算符。

**LINQ Lambda 表达式**

input.Select(family=>family.parents[0])  
 .Where(familyName == "Smith");

**SQL**

SELECT \*  
FROM Families f  
WHERE f.parents[0].familyName = "Smith"

**LINQ Lambda 表达式**

input.Where(family => family.children[0].grade > 3)  
 .Select(family => family.parents[0].familyName);

**SQL**

SELECT VALUE f.parents[0].familyName  
FROM Families f  
WHERE f.children[0].grade > 3

**LINQ Lambda 表达式**

input.Select(family => new { grade=family.children[0].grade}).  
 Where(anon=> anon.grade < 3);

**SQL**

SELECT \*  
FROM Families f  
WHERE ({grade: f.children[0].grade}.grade > 3)

**LINQ Lambda 表达式**

input.SelectMany(family => family.parents)  
 .Where(parent => parents.familyName == "Smith");

**SQL**

SELECT \*  
FROM p IN Families.parents  
WHERE p.familyName = "Smith"

#### 嵌套

语法为 input.SelectMany(x=>x.Q())，其中 Q 是 Select、SelectMany 或 Where 运算符。

在嵌套查询中，内部查询应用到外部集合的每个元素。一个重要的功能是内部查询可以引用外部集合（如自联接）中元素的字段。

**LINQ Lambda 表达式**

input.SelectMany(family=>   
 family.parents.Select(p => p.familyName));

**SQL**

SELECT VALUE p.familyName  
FROM Families f  
JOIN p IN f.parents

**LINQ Lambda 表达式**

input.SelectMany(family =>   
 family.children.Where(child => child.familyName == "Jeff"));

**SQL**

SELECT \*  
FROM Families f  
JOIN c IN f.children  
WHERE c.familyName = "Jeff"

**LINQ Lambda 表达式**

input.SelectMany(family => family.children.Where(  
 child => child.familyName == family.parents[0].familyName));

**SQL**

SELECT \*  
FROM Families f  
JOIN c IN f.children  
WHERE c.familyName = f.parents[0].familyName

## 执行 SQL 查询

DocumentDB 通过一个 REST API 来公开资源，任何可以发出 HTTP/HTTPS 请求的语言都可以调用该 REST API。此外，DocumentDB 还为多种常用语言（如 .NET、Node.js、JavaScript 和 Python）提供编程库。REST API 和各种库均支持通过 SQL 进行查询。除了 SQL 之外，.NET SDK 还支持 LINQ 查询。

以下示例演示了如何对 DocumentDB 数据库帐户创建和提交该查询。

### REST API

DocumentDB 通过 HTTP 提供开放的 RESTful 编程模型。可以使用 Azure 订阅预配数据库帐户。DocumentDB 资源模型由数据库帐户下的一组资源组成，每个资源都可使用逻辑和稳定的 URI 进行寻址。本文档中将一组资源称为一个源。数据库帐户由一组数据库组成，每个数据库包含多个集合，而每个集合又包含文档、UDF 和其他资源类型。

这些资源的基本交互模型借助采用其标准解释的 HTTP 谓词 GET、PUT、POST 和 DELETE。POST 谓词用于创建新的资源、执行存储过程或发出 DocumentDB 查询。查询始终为只读操作，且无任何副作用。

以下示例演示了针对包含我们目前看到的两个示例文档的集合，使用 POST 进行 DocumentDB 查询。查询对 JSON 名称属性进行简单的筛选。请注意，使用 x-ms-documentdb-isquery 和 Content-Type: application/query+json 标头表示该操作是一个查询。

**请求**

POST https://<REST URI>/docs HTTP/1.1  
...  
x-ms-documentdb-isquery: True  
Content-Type: application/query+json  
  
{   
 "query": "SELECT \* FROM Families f WHERE f.id = @familyId",   
 "parameters": [   
 {"name": "@familyId", "value": "AndersenFamily"}   
 ]   
}

**结果**

HTTP/1.1 200 Ok  
x-ms-activity-id: 8b4678fa-a947-47d3-8dd3-549a40da6eed  
x-ms-item-count: 1  
x-ms-request-charge: 0.32  
  
<indented for readability, results highlighted>  
  
{   
 "\_rid":"u1NXANcKogE=",  
 "Documents":[   
 {   
 "id":"AndersenFamily",  
 "lastName":"Andersen",  
 "parents":[   
 {   
 "firstName":"Thomas"  
 },  
 {   
 "firstName":"Mary Kay"  
 }  
 ],  
 "children":[   
 {   
 "firstName":"Henriette Thaulow",  
 "gender":"female",  
 "grade":5,  
 "pets":[   
 {   
 "givenName":"Fluffy"  
 }  
 ]  
 }  
 ],  
 "address":{   
 "state":"WA",  
 "county":"King",  
 "city":"seattle"  
 },  
 "\_rid":"u1NXANcKogEcAAAAAAAAAA==",  
 "\_ts":1407691744,  
 "\_self":"dbs\/u1NXAA==\/colls\/u1NXANcKogE=\/docs\/u1NXANcKogEcAAAAAAAAAA==\/",  
 "\_etag":"00002b00-0000-0000-0000-53e7abe00000",  
 "\_attachments":"\_attachments\/"  
 }  
 ],  
 "count":1  
}

第二个示例演示了从联接中返回多个结果的更复杂的查询。

**请求**

POST https://<REST URI>/docs HTTP/1.1  
...  
x-ms-documentdb-isquery: True  
Content-Type: application/query+json  
  
{   
 "query": "SELECT   
 f.id AS familyName,   
 c.givenName AS childGivenName,   
 c.firstName AS childFirstName,   
 p.givenName AS petName   
 FROM Families f   
 JOIN c IN f.children   
 JOIN p in c.pets",   
 "parameters": []   
}

**结果**

HTTP/1.1 200 Ok  
x-ms-activity-id: 568f34e3-5695-44d3-9b7d-62f8b83e509d  
x-ms-item-count: 1  
x-ms-request-charge: 7.84  
  
<indented for readability, results highlighted>  
  
{   
 "\_rid":"u1NXANcKogE=",  
 "Documents":[   
 {   
 "familyName":"AndersenFamily",  
 "childFirstName":"Henriette Thaulow",  
 "petName":"Fluffy"  
 },  
 {   
 "familyName":"WakefieldFamily",  
 "childGivenName":"Jesse",  
 "petName":"Goofy"  
 },  
 {   
 "familyName":"WakefieldFamily",  
 "childGivenName":"Jesse",  
 "petName":"Shadow"  
 }  
 ],  
 "count":3  
}

如果查询的结果无法包含在一页内，那么 REST API 通过 x-ms-continuation-token 响应标头返回继续标记。客户端可以通过在后续结果中包含该标头对结果进行分页。可以通过 x-ms-max-item-count 数量标头控制每页的结果数。

若要管理查询的数据一致性策略，请使用 x-ms-consistency-level 标头（如所有的 REST API 请求）。对于会话一致性，还需要回显查询请求中最新的 x-ms-session-token Cookie 标头。请注意，查询集合的索引策略也可以影响查询结果的一致性。使用默认的索引策略设置，集合的索引始终与文档内容保持同步，且查询结果将与为数据选择的一致性匹配。如果索引策略太放松而具有延迟，那么查询会返回过时的结果。有关详细信息，请参阅 [DocumentDB Consistency Levels（DocumentDB 一致性级别）](/documentation/articles/documentdb-consistency-levels)。

如果为集合配置的索引策略不能支持指定的查询，那么 DocumentDB 服务器会返回 400“错误的请求”。在对为哈希（等式）查找配置的路径，以及从索引中显式排除的路径进行范围查询时，将返回此内容。当索引不可用时，可通过指定 x-ms-documentdb-query-enable-scan 标头以允许查询执行扫描。

### C# (.NET) SDK

.NET SDK 支持 LINQ 和 SQL 查询。以下示例演示了如何执行本文档之前介绍的简单筛选查询。

foreach (var family in client.CreateDocumentQuery(collectionLink,   
 "SELECT \* FROM Families f WHERE f.id = "AndersenFamily""))  
{  
 Console.WriteLine("\tRead {0} from SQL", family);  
}  
  
SqlQuerySpec query = new SqlQuerySpec("SELECT \* FROM Families f WHERE f.id = @familyId");  
query.Parameters = new SqlParameterCollection();  
query.Parameters.Add(new SqlParameter("@familyId", "AndersenFamily"));  
  
foreach (var family in client.CreateDocumentQuery(collectionLink, query))  
{  
 Console.WriteLine("\tRead {0} from parameterized SQL", family);  
}  
  
foreach (var family in (  
 from f in client.CreateDocumentQuery(collectionLink)  
 where f.Id == "AndersenFamily"  
 select f))  
{  
 Console.WriteLine("\tRead {0} from LINQ query", family);  
}  
  
foreach (var family in client.CreateDocumentQuery(collectionLink)  
 .Where(f => f.Id == "AndersenFamily")  
 .Select(f => f))  
{  
 Console.WriteLine("\tRead {0} from LINQ lambda", family);  
}

此示例比较了每个文档内等式的两个属性，并使用匿名投影。

foreach (var family in client.CreateDocumentQuery(collectionLink,  
 @"SELECT {""Name"": f.id, ""City"":f.address.city} AS Family   
 FROM Families f   
 WHERE f.address.city = f.address.state"))  
{  
 Console.WriteLine("\tRead {0} from SQL", family);  
}  
  
foreach (var family in (  
 from f in client.CreateDocumentQuery<Family>(collectionLink)  
 where f.address.city == f.address.state  
 select new { Name = f.Id, City = f.address.city }))  
{  
 Console.WriteLine("\tRead {0} from LINQ query", family);  
}  
  
foreach (var family in  
 client.CreateDocumentQuery<Family>(collectionLink)  
 .Where(f => f.address.city == f.address.state)  
 .Select(f => new { Name = f.Id, City = f.address.city }))  
{  
 Console.WriteLine("\tRead {0} from LINQ lambda", family);  
}

下面的示例中演示了通过 LINQ SelectMany 进行快递的联接。

foreach (var pet in client.CreateDocumentQuery(collectionLink,  
 @"SELECT p  
 FROM Families f   
 JOIN c IN f.children   
 JOIN p in c.pets   
 WHERE p.givenName = ""Shadow"""))  
{  
 Console.WriteLine("\tRead {0} from SQL", pet);  
}  
  
// Equivalent in Lambda expressions  
foreach (var pet in  
 client.CreateDocumentQuery<Family>(collectionLink)  
 .SelectMany(f => f.children)  
 .SelectMany(c => c.pets)  
 .Where(p => p.givenName == "Shadow"))  
{  
 Console.WriteLine("\tRead {0} from LINQ lambda", pet);  
}

.NET 客户端自动遍历 foreach 块中所有的查询结果页，如上所示。在 REST API 部分介绍的查询选项也适用于 CreateDocumentQuery 方法中使用 FeedOptions 和 FeedResponse 的 .NET SDK。可使用 MaxItemCount 设置控制页面的数量。

你也可以通过使用 IQueryable 对象创建 IDocumentQueryable，然后读取 ResponseContinuationToken 值并将其作为 FeedOptions 中的 RequestContinuationToken 传回来显式控制分页。当配置的索引策略无法支持查询时，也可以设置 EnableScanInQuery 以启用扫描。对于分区集合，可以使用 PartitionKey 来针对单个分区运行查询（尽管 DocumentDB 可以自动从查询文本中提取此内容），并使用 EnableCrossPartitionQuery 来运行需要针对多个分区运行的查询。

有关包含查询的更多示例，请参阅 [DocumentDB .NET samples（DocumentDB .NET 示例）](https://github.com/Azure/azure-documentdb-net)。

### JavaScript 服务器端 API

DocumentDB 使用存储过程和触发器，为对集合直接执行基于 JavaScript 的应用程序逻辑提供编程模型。在集合级别上注册的 JavaScript 逻辑稍后可以对针对给定集合的文档的操作发出数据库操作。这些操作包装在环境 ACID 事务中。

以下示例演示了如何在 JavaScript 服务器 API 中使用 queryDocuments 来从存储过程和触发器内部进行查询。

function businessLogic(name, author) {  
 var context = getContext();  
 var collectionManager = context.getCollection();  
 var collectionLink = collectionManager.getSelfLink()  
  
 // create a new document.  
 collectionManager.createDocument(collectionLink,  
 { name: name, author: author },  
 function (err, documentCreated) {  
 if (err) throw new Error(err.message);  
  
 // filter documents by author  
 var filterQuery = "SELECT \* from root r WHERE r.author = 'George R.'";  
 collectionManager.queryDocuments(collectionLink,  
 filterQuery,  
 function (err, matchingDocuments) {  
 if (err) throw new Error(err.message);  
context.getResponse().setBody(matchingDocuments.length);  
  
 // Replace the author name for all documents that satisfied the query.  
 for (var i = 0; i < matchingDocuments.length; i++) {  
 matchingDocuments[i].author = "George R. R. Martin";  
 // we don't need to execute a callback because they are in parallel  
 collectionManager.replaceDocument(matchingDocuments[i].\_self,  
 matchingDocuments[i]);  
 }  
 })  
 });  
}

## 参考

1. [Azure DocumentDB 简介](/documentation/articles/documentdb-introduction)
2. [DocumentDB SQL specification（DocumentDB SQL 规范）](http://go.microsoft.com/fwlink/p/?LinkID=510612)
3. [DocumentDB .NET samples（DocumentDB .NET 示例）](https://github.com/Azure/azure-documentdb-net)
4. [DocumentDB Consistency Levels（DocumentDB 一致性级别）](/documentation/articles/documentdb-consistency-levels)
5. ANSI SQL 2011 <http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681>
6. JSON <http://json.org/>
7. Javascript 规范 <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
8. LINQ <http://msdn.microsoft.com/library/bb308959.aspx>
9. 针对大型数据库的查询评估技术 <http://dl.acm.org/citation.cfm?id=152611>
10. Query Processing in Parallel Relational Database Systems, IEEE Computer Society Press, 1994
11. Lu, Ooi, Tan, Query Processing in Parallel Relational Database Systems, IEEE Computer Society Press, 1994.
12. Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig Latin: A Not-So-Foreign Language for Data Processing, SIGMOD 2008.
13. G.Graefe.The Cascades framework for query optimization.IEEE Data Eng.Bull., 18(3): 1995.