

JoyGraph GQL 手册

前言

JoyGraph 是基于图论为基础，基于 c++ 开发的图数据库，提供了大规模、分布式图存储解决方案。支持多图、在线扩容、分布式查询、批计算等功能。

适用于反欺诈、实时推荐、知识图谱、社交网络存储分析等场景。提供类 openCypher 的数据查询语言。

数据模型

JoyGraph 遵循强 schema 约束。也就是说在插入点边前必须保证 schema 存在和数据类型匹配。

JoyGraph 支持 6 种基础数据模型。



- 图空间

图空间 (Space) 在逻辑上支持了多图存储，是 JoyGraph 的多图存储的基本单元。实现了不同团队或者项目的数据逻辑隔离。不同图空间的数据是相互隔离的，可以指定不同的存储副本数、权限、分片等。

- 标签

标签 (Tag) 由一组事先预定义的属性构成。是点存储数据存储、读取的基础模版。

● 边类型

边类型与标签类似也是由一组事先预定义的属性构成。不同点在于它是边存储数据存储、读取的基础模版。

● 属性

属性是指以键值对（Key-value pair）形式存储的信息。

● 点

点用来保存实体对象，特点如下：

点是用点标识符（VID）标识的。VID 在同一图空间中唯一。VID 是一个 int64，或者 fixed_string(N)。

点必须有至少一个 Tag，也可以有多个 Tag。但不能没有 Tag。

● 边

边是用来连接点的，表示两个点之间的关系或行为，特点如下：

两点之间可以有多条边。

边是有方向的，不存在无向边。

四元组 <起点 VID、Edge type、边排序值 (Rank)、终点 VID> 用于唯一标识一条边。

一条边有且仅有一个 Edge type。

一条边有且仅有一个类型为 int64 的 rank 值（默认为 0）。

JoyGraph 数据类型

JoyGraph 支持数据类型如下：

数值类型	类型名称	备注
整形	INT64 或 INT	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
	INT32	-2,147,483,648 ~ 2,147,483,647
	INT16	-32,768 ~ 32,767
	INT8	-128 ~ 127
浮点数	FLOAT	3.4E +/- 38 (6~7 位)
	DOUBLE	1.7E +/- 308 (15~16 位)
布尔	BOOL	True/False
字符串	STRING	变长字符串
	FIXED_STRING(<length>)	定长字符串
时间和日期	DATE	YYYY-MM-DD
	TIME	hh:mm:ss.msmsmsusus

数值类型	类型名称	备注
	DATETIME	YYYY-MM-DDThh:mm:ss.msmsmsususus
	TIMESTAMP	1970-01-01T00:00:01
空类型	NULL	NULL/NOT NULL
列表	List	[1, 2, 3]
集合	Set	{1, 5, 3}
映射	Map	{key: 'Value', listKey: [{inner: 'Map1'}, {inner: 'Map2'}]}
地理位置	GEOGRAPHY	POINT(3 8)、LINESTRING(3 8, 4.7 73.23)、POLYGON((0 1, 1 2, 2 3, 0 1))
类型转换	将表达式的类型转换为另一个类型	(type_name)expression

运算符

兼容 opencypher 语法运算符

名称	符号
比较运算符	=、>、>=、<、<=、!=、IS [NOT] NULL、IS [NOT] EMPTY
Bool 运算符	AND、OR、NOT、XOR
管道符	
引用符	\$^、\$\$、\$-
集合运算符	UNION、UNION DISTINCT、UNION ALL、INTERSECT、MINUS
字符串运算符	+、CONTAINS、(NOT) IN、(NOT) STARTS WITH、(NOT) ENDS WITH、正则表达式
列表运算符	+、IN、[]

GQL 语法指导

本文介绍 JoyGraph 查询语言的基础语法，包括用于 Schema 创建和常用增删改查操作的语句。可以分为四类语法：空间管理、schema 管理、数据 CRUD、函数、用户管理。

GQL 关键字不区分大小写，所以在编写查询语句时需要特别注意。

图空间管理

创建图空间

为实现多图隔离，用户可以在同一实例中创建多个图空间。

语法：

```
CREATE SPACE [IF NOT EXISTS] <graph_space_name>
( [partition_num = <partition_number> ,] [replica_factor = <replica_number> ,] vid_type
= {FIXED_STRING(<N>)|INT[64]} ) [COMMENT = '<comment>']
```

示例：

```
CREATE SPACE IF NOT EXISTS family(partition_num=15, replica_factor=1,
vid_type=FIXED_STRING(30))
```

克隆图空间

新建一个相同配置的图空间，并复制所有的 schema 定义。

语法：

```
CREATE SPACE <new_graph_space_name> AS <old_graph_space_name>
```

示例：

```
CREATE SPACE family as family_3
```

列出创建成功的图空间

示例：

```
Show spaces;
```

选择图空间

列出所有的图空间。

语法：

```
Use <graph_sapce_name>
```

示例：

```
USE family;
```

显示图空间信息

语法：

```
DESC[RIBE] SPACE <graph_space_name>
```

示例：

```
DESCRIBE SPACE family;
```

删除图空间

语法：

```
DROP SPACE [IF EXISTS] <graph_space_name>
```

示例：

```
DROP SPACE family
```

Schema 管理

管理 Tag

创建 Tag

语法：

```
CREATE TAG [IF NOT EXISTS] <tag_name> ( <prop_name> <data_type> [NULL | NOT  
NULL] [DEFAULT <default_value>]  
[COMMENT '<comment>'] [{, <prop_name> <data_type> [NULL | NOT NULL]  
[DEFAULT <default_value>] [COMMENT  
'<comment>']} ...] ) [TTL_DURATION = <ttl_duration>][TTL_COL = <prop_name>]  
[COMMENT = '<comment>']
```

示例：

```
CREATE TAG member(name string, age int, married bool, salary double, create_time  
timestamp) TTL_DURATION = 100, TTL_COL = "create_time"
```

删除 Tag

语法：

```
DROP TAG [IF EXISTS] <tag_name>
```

示例：

```
DROP TAG member;
```

修改 Tag

语法：

```
ALTER TAG <tag_name> <alter_definition> [,  
alter_definition] ...] [ttl_definition [,  
ttl_definition] ... ] [COMMENT = '<comment>']
```

示例：

```
ALTER TAG member ADD (p3 int, p4 string)
```

显示所有 Tags

语法：

```
SHOW TAGS
```

示例：

```
SHOW TAGS
```

显示 Tag 详细信息

语法：

```
DESC[RIBE] TAG <tag_name>
```

示例：

```
DESCRIBE TAG member
```

管理 Edge type

创建 Edge type

语法：

```
CREATE EDGE [IF NOT EXISTS] <edge_type_name> ( <prop_name> <data_type>  
[NULL | NOT NULL] [DEFAULT <default_value>] [COMMENT '<comment>'] [{,  
<prop_name> <data_type> [NULL | NOT NULL] [DEFAULT <default_value>]  
[COMMENT '<comment>']} ...] ) [TTL_DURATION = <ttl_duration>] [TTL_COL =  
<prop_name>] [COMMENT = '<comment>']
```

示例：

```
CREATE EDGE relate(relate_type string) TTL_DURATION = 100, TTL_COL = "p2"
```

删除 Edge type

语法：

```
DROP EDGE [IF EXISTS] <edge_type_name>
```

示例：

```
DROP EDGE relate
```

修改 Edge type

语法：

```
ALTER EDGE <edge_type_name> <alter_definition> [,alter_definition] ...]  
[ttl_definition , ttl_definition] ... ] [COMMENT = '<comment>']
```

示例：

```
ALTER EDGE relate ADD (p3int, p4 string)
```

查看所有的 Edge type

语法：

```
SHOW EDGES
```

示例：

```
SHOW EDGES
```

查看 Edge type 定义

语法：

```
DESC[RIBE] EDGE <edge_type_name> DESCRIBE EDGE
```

示例：

```
DESCRIBE EDGE relate
```

图数据 CRUD

点操作

插入点

语法：

```
INSERT VERTEX [IF NOT EXISTS] <tag_name>(<prop_name_list>) [, <tag_name>
(<prop_name_list>), ...] {VALUES | VALUE} VID:(<prop_value_list>[, <prop_value_list>])
```

示例：

```
INSERT VERTEX member (name,age) VALUES "mother":("母亲",36), "father":("父亲",
40), "daughter":("女儿",12), "son":("儿子", 8)
```

删除点

语法：

```
DELETE VERTEX <vid> [, <vid> ...] DELETE VERTEX
```

示例：

```
DELETE VERTEX "son"
```

更新点

语法：

```
UPDATE VERTEX ON <tag_name> <vid> SET <update_prop> [WHEN <condition>]
[YIELD <output>]
```

示例：

```
UPDATE VERTEX ON member "son" SET age = age + 2
```

Upsert 点

语法：

```
UPSERT VERTEX ON <tag> <vid> SET <update_prop> [WHEN <condition>] [YIELD
<output>]
```

示例：

```
UPSERT VERTEX ON member "son" SET age = 31
```

边操作

插入边

语法：

```
INSERT EDGE [IF NOT EXISTS] <edge_type> (<property_name>[, <property_name>...])
{VALUES | VALUE} <src_vid> -> <dst_vid>[@<rank>] : (<property_value>[,
```



```
<property_value>...]) [, <src_vid> -> <dst_vid>[@<rank>] : (<property_name>[,  
<property_name>...]), ...];
```

示例:

```
INSERT EDGE relate (relate_type) VALUES "mother"->"son": ("have")
```

删除边

语法:

```
DELETE EDGE <edge_type> <src_vid> -> <dst_vid>[@<rank>] [,<src_vid> ->  
<dst_vid>[@<rank>] ...]
```

示例:

```
DELETE EDGE relate_type "son" ->"mother "@0
```

更新边

语法:

```
UPDATE EDGE ON <edge_type> <src_vid> -> <dst_vid>[@<rank>] SET  
<update_prop> [WHEN <condition>] [YIELD<output>]
```

示例:

```
UPDATE EDGE ON relate_type "son" ->"mother "@0 SET relate_type = "keep"
```

Upsert 边

语法:

```
UPSERT EDGE ON <edge_type> <src_vid> -> <dst_vid> [@rank] SET <update_prop>  
[WHEN <condition>] [YIELD <properties>]
```

示例:

```
UPSERT EDGE on relate_type "son" ->"mother "@0 SET relate_type = "keep"
```

索引操作

创建索引

语法:

```
CREATE {TAG | EDGE} INDEX [IF NOT EXISTS] <index_name> ON {<tag_name> |  
<edge_name>} ([<prop_name_list>]) [COMMENT = '<comment>']
```

示例：

```
CREATE TAG INDEX person on member()
```

显示所有索引

语法：

```
SHOW {TAG | EDGE} INDEXES
```

示例：

```
SHOW TAG INDEXES
```

查看索引信息

语法：

```
DESCRIBE {TAG | EDGE} INDEX <index_name>
```

示例：

```
DESCRIBE TAG INDEX person
```

重建索引

语法：

```
REBUILD {TAG | EDGE} INDEX [<index_name_list>]
```

示例：

```
REBUILD TAG INDEX person
```

查看索引的状态

语法：

```
SHOW {TAG | EDGE} INDEX STATUS
```

示例：

```
SHOW TAG INDEX STATUS
```

删除索引

语法：

```
DROP {TAG | EDGE} INDEX [IF EXISTS] <index_name>
```

示例：

DROP TAG INDEX person

查询操作

子图查询

语法：

```
GET SUBGRAPH [WITH PROP] [<step_count> STEPS] FROM {<vid>, <vid>...} [{IN | OUT  
| BOTH} <edge_type>, <edge_type>...] [YIELD [VERTICES AS <vertex_alias>] [,EDGES  
AS <edge_alias>]]
```

示例：

```
GET SUBGRAPH 1 STEPS FROM "mother" YIELD VERTICES AS nodes, EDGES  
AS relationships
```

路径查询

语法：

```
FIND { SHORTEST | ALL | NOLOOP } PATH [WITH PROP] FROM <vertex_id_list> TO  
<vertex_id_list><br/>OVER <edge_type_list> [REVERSELY | BIDIRECT] [<WHERE  
clause>] [UPTO <N> STEPS] [| ORDER BY $-.path] [|LIMIT <M>]
```

示例：

```
FIND SHORTEST PATH FROM "mother" TO "son" OVER *
```

通用查询语句

GQL 在设计上兼容了 openCypher 的部分语法。

MATCH 语句提供基于模式（pattern）匹配的搜索功能。

一个 MATCH 语句定义了一个搜索模式，用该模式匹配存储中的数据，然后用 RETURN 子句检索数据。

MATCH 语句使用原生索引查找起始点或边，起始点或边可以在模式的任何位置。即一个有效的 MATCH 语句，必须有一个属性、Tag 或 Edge type 已经创

建索引，或者在 WHERE 子句中用 id() 函数指定了特定点的 VID。

匹配点

语法：

```
MATCH <pattern> [<WHERE clause>] RETURN <output>;
```

示例：

```
MATCH (v) WHERE id(v) == 'son' RETURN v;
```

匹配 Tag

示例：

```
MATCH (v:member) RETURN v;
```

匹配点的属性

示例：

```
MATCH (v:member{name:"儿子"}) RETURN v;
```

匹配点 ID

示例：

```
MATCH (v) WHERE id(v) == 'son' RETURN v;
```

匹配连接的点

示例：

```
MATCH (v:member{name:"儿子"})--(v2) RETURN v2.name AS Name;
```

匹配路径

示例：

```
MATCH p=(v:member{name:"儿子"})-->(v2) RETURN p;
```

匹配边

示例：

```
MATCH (v:member{name:"儿子"})-[e]-(v2) RETURN e;
```

匹配 EDGE TYPE

示例：

```
MATCH ()-[e:follow]-() RETURN e;
```

匹配边的属性

示例：

```
MATCH (v:member{name:"儿子"})-[e:follow{degree:95}]->(v2) RETURN e;
```

匹配多个 EDGE TYPE

示例：

```
MATCH (v:member{name:"儿子"})-[e:relate_type |:follow]->(v2) RETURN e;
```

匹配多条边

示例：

```
MATCH (v:member{name:"女儿"})-[e:follow|:relate_type]->(v2) RETURN e;
```

匹配定长路径

示例：

```
MATCH p=(v:member{name:"母亲"})-[e:follow*2]->(v2) RETURN DISTINCT v2 AS  
Friends;
```

匹配变长路径

示例：

```
MATCH p=(v:member{name:"儿子"})-[e:follow*1..3]->(v2) RETURN v2 AS Friends;
```

匹配多个 EDGE TYPE 的变长路径

示例：

```
MATCH p=(v:member{name:"女儿"})-[e:follow|relate_type*2]->(v2)  
  
RETURN DISTINCT v2;
```

索引操作

LOOKUP 语句

前提条件：

请确保 LOOKUP 语句有至少一个索引可用。如果需要创建索引，但是已经有相关的点、边或属性，用户必须在创建索引后重建索引，才能使其生效。

语法：

```
LOOKUP ON {<vertex_tag> | <edge_type>}  
[WHERE <expression> [AND <expression> ...]]  
[YIELD <return_list> [AS <alias>]];  
  
<return_list>  
  
<prop_name> [AS <col_alias>] [, <prop_name> [AS <prop_alias>] ...];
```

示例：

```
LOOKUP ON member  
  
WHERE member.name == "儿子"  
  
YIELD properties(vertex).name AS name |  
  
GO FROM $-.VertexID OVER relate_type  
  
YIELD $-.name;
```

GO 语句

GO 用指定的过滤条件遍历图，并返回结果。

语法：

```
GO [[<M> TO] <N> STEPS ] FROM <vertex_list>  
OVER <edge_type_list> [{REVERSELY | BIDIRECT}]  
[ WHERE <conditions> ]  
[YIELD [DISTINCT] <return_list>]  
[{SAMPLE <sample_list> | <limit_by_list_clause>}]  
[[ GROUP BY {col_name | expr | position} YIELD <col_name>]  
[[ ORDER BY <expression> [{ASC | DESC}]]
```

```
[| LIMIT [<offset>,<number_rows>];
```

```
<vertex_list> ::=
```

```
<vid> [, <vid> ...]
```

```
<edge_type_list> ::=
```

```
edge_type [, edge_type ...]
```

```
| *
```

示例:

```
GO FROM "son", "daughter" OVER relate_type \
```

```
WHERE relate_type == "brother" \
```

```
YIELD DISTINCT properties($$).name AS member_name;
```

FETCH 语句

FETCH 可以获取指定点或边的属性值。

语法:

```
FETCH PROP ON {<tag_name>[, tag_name ...] | *}
```

```
<vid> [, vid ...]
```

```
[YIELD <return_list> [AS <alias>]];
```

示例:

```
FETCH PROP ON * "son", "daughter", "mother";
```

```
FETCH PROP ON serve "son" -> "father" YIELD properties(edge).relate_type;
```

UNWIND 语句

UNWIND 语句可以将列表拆分为单独的行，列表中的每个元素为一行。

UNWIND 可以作为单独语句或语句中的子句使用。

语法:

```
UNWIND <list> AS <alias> <RETURN clause>;
```

示例:

```
UNWIND [1,2,3] AS n RETURN n;
```

SHOW 语句

SHOW CHARSET 语句显示当前的字符集。

目前可用的字符集为 utf8 和 utf8mb4 。默认字符集为 utf8 。扩展 utf8 支持四字节字符，因此 utf8 和 utf8mb4 是等价的。

语法：

```
SHOW CHARSET;
```

示例：

```
SHOW CHARSET;
```

子句和选项

GROUP BY

GROUP BY 子句可以用于聚合数据。

示例：

```
MATCH (v:member)<-[:follow]-(:member) RETURN v.name AS Name, count(*) as cnt
ORDER BY cnt DESC;
```

GROUP BY

GROUP BY 子句可以用于聚合数据。

语法：

```
| GROUP BY <var> YIELD <var>, <aggregation_function(var)>
```

示例：

```
MATCH (v:member)<-[:follow]-(:member) RETURN v.name AS Name, count(*) as cnt
ORDER BY cnt DESC;

GO FROM "son" OVER follow BIDIRECT YIELD properties($$).name as Name

| GROUP BY $-.Name YIELD $-.Name as Member, count(*) AS Name_Count;
```

LIMIT

LIMIT 子句限制输出结果的行数。

语法：

```
| LIMIT [<offset>,<number_rows>;
```


示例：

```
GO FROM "son" OVER follow REVERSELY \
YIELD properties($$).name AS Friend, properties($$).age AS Age \
| ORDER BY $-.Age, $-.Friend | LIMIT 1, 3;
```

SAMPLE

SAMPLE 子句用于在结果集中均匀取样并返回指定数量的数据。

语法：

```
<go_statement> SAMPLE <sample_list>;
```

示例：

```
GO 1 TO 3 STEPS FROM "son" \
OVER * \
YIELD properties($$).name AS NAME, properties($$).age AS Age \
SAMPLE [2,2,2];
```

ORDER BY

ORDER BY 子句指定输出结果的排序规则。

在原生 GQL 中，必须在 YIELD 子句之后使用管道符 (|) 和 ORDER BY 子句。

在 openCypher 方式中，不允许使用管道符。在 RETURN 子句之后使用 ORDER BY 子句。

排序规则分为如下两种：

ASC （默认）：升序。

DESC：降序。

语法：

```
ORDER BY <expression> [ASC | DESC] [, <expression> [ASC | DESC] ...];
<RETURN clause> ORDER BY <expression> [ASC | DESC] [, <expression> [ASC |
DESC] ...];
```

示例：

```
FETCH PROP ON member "son", "daughter", "mother", "father" \
YIELD properties(vertex).age AS age, properties(vertex).name AS name | ORDER BY
$-.age ASC, $-.name DESC;
```

```
MATCH (v:member) RETURN v.name AS Name, v.age AS Age ORDER BY Name DESC;
```

RETURN

RETURN 子句定义了 GQL 查询的输出结果。如果需要返回多个字段, 用英文逗号 (,) 分隔。

RETURN 可以引导子句或语句:

RETURN 子句可以用于 GQL 中的 openCypher 方式语句中, 例如 MATCH 或 UNWIND 。

RETURN 可以单独使用, 输出表达式的结果。

示例:

```
MATCH (v:member) RETURN (v)-[e]->(v2);
```

TTL

TTL (Time To Live) 指定属性的存活时间, 超时后, 该属性就会过期。

示例:

```
CREATE TAG IF NOT EXISTS t1 (a timestamp);
```

```
ALTER TAG t1 ttl_col = "a", ttl_duration = 5;
```

```
INSERT VERTEX t1(a) values "101":(now());
```

WHERE

WHERE 子句可以根据条件过滤输出结果。

WHERE 子句通常用于如下查询:

原生 GQL, 例如 GO 和 LOOKUP 语句。

openCypher 方式, 例如 MATCH 和 WITH 语句。

示例:

```
MATCH (v:member) \
```

```
WHERE v.name == "儿子" \
```

```
XOR (v.age < 13 AND v.name == "儿子") \
```

```
OR NOT (v.name == "儿子" OR v.name == "女儿") \
```

```
RETURN v.name, v.age;
```

```
GO FROM "son" \
```

```
OVER follow \
```

```
WHERE follow.degree > 90 \
OR properties($$).age != 33 \
AND properties($$).name != "母亲" \
YIELD properties($$);
```

YIELD

YIELD 定义 GQL 查询的输出结果。

YIELD 可以引导子句或语句：

YIELD 子句可以用于原 GQL 语句中，例如 GO 、 FETCH 或 LOOKUP 。

YIELD 语句可以在独立查询或复合查询中使用。

语法：

```
YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...]
[WHERE <conditions>];
```

示例：

```
GO FROM "son" OVER follow
YIELD properties($$).name AS Friend, properties($$).age AS Age;
```

WITH

WITH 子句可以获取并处理查询前半部分的结果，并将处理结果作为输入传递给查询的后半部分。

语法：

```
YIELD [DISTINCT] <col> [AS <alias>] [, <col> [AS <alias>] ...]
[WHERE <conditions>];
```

示例：

```
MATCH (v) WHERE id(v)="son" WITH labels(v) AS tags_unf
UNWIND tags_unf AS tags_f RETURN tags_f;
```

用户管理

创建用户

执行 CREATE USER 语句可以创建新用户。当前仅 God 角色用户（即 root 用户）能够执行 CREATE USER 语句。

语法：

```
CREATE USER [IF NOT EXISTS] <user_name> [WITH PASSWORD '<password>'];
```

示例：

```
CREATE USER user1 WITH PASSWORD 'joygraph';
```

授权用户

执行 GRANT ROLE 语句可以将指定图空间的内置角色权限授予用户。当前仅 God 角色用户和 Admin 角色用户能够执行 GRANT ROLE 语句。

语法：

```
GRANT ROLE <role_type> ON <space_name> TO <user_name>;
```

示例：

```
GRANT ROLE USER ON family TO user1;
```

撤销用户权限

执行 REVOKE ROLE 语句可以撤销用户的指定图空间的内置角色权限。当前仅 God 角色用户和 Admin 角色用户能够执行 REVOKE ROLE 语句。角色权限的

说明。

语法：

```
REVOKE ROLE <role_type> ON <space_name> FROM <user_name>;
```

示例：

```
REVOKE ROLE USER ON family FROM user1;
```

查看用户权限

执行 SHOW ROLES 语句可以显示分配给用户的角色信息。

语法：

```
SHOW ROLES IN <space_name>;
```

示例：

```
SHOW ROLES IN family;
```

修改用户密码 (CHANGE PASSWORD)

执行 CHANGE PASSWORD 语句可以修改用户密码，修改时需要提供旧密码和新密码。

语法：

```
CHANGE PASSWORD <user_name> FROM '<old_password>' TO '<new_password>';
```

示例：

```
CHANGE PASSWORD user1 FROM 'joygraph' TO 'joygraph123';
```

修改用户密码 (ALTER USER)

执行 ALTER USER 语句可以修改用户密码，修改时不需要提供旧密码。当前仅 God 角色用户（即 root 用户）能够执行 ALTER USER 语句。

语法：

```
ALTER USER <user_name> WITH PASSWORD '<password>';
```

示例：

```
ALTER USER user1 WITH PASSWORD 'joygraph';
```

删除用户

执行 DROP USER 语句可以删除用户。当前仅 God 角色用户能够执行 DROP USER 语句。

语法：

```
DROP USER [IF EXISTS] <user_name>;
```

示例：

```
DROP USER user1;
```

查看用户列表

执行 SHOW USERS 语句可以查看用户列表。当前仅 God 角色用户能够执行 SHOW USERS 语句。

语法：

```
SHOW USERS;
```

示例：

SHOW USERS;

附录

函数

数学函数

函数说明

`double abs(double x)` 返回 x 的绝对值。

`double floor(double x)` 返回小于或等于 x 的最大整数。

`double ceil(double x)` 返回大于或等于 x 的最小整数。

`double round(double x)` 返回离 x 最近的整数值，如果 x 恰好在中间，则返回离 0 较远的整数。

`double sqrt(double x)` 返回 x 的平方根。

`double cbrt(double x)` 返回 x 的立方根。

`double hypot(double x,`

`double y)`

返回直角三角形（直角边长为 x 和 y ）的斜边长。

`double pow(double x,`

`double y)`

返回 x^y 的值。

`double exp(double x)` 返回 e^x 的值。

`double exp2(double x)` 返回 2^x 的值。

`double log(double x)` 返回以自然数 e 为底 x 的对数。

`double log2(double x)` 返回以 2 为底 x 的对数。

`double log10(double x)` 返回以 10 为底 x 的对数。

`double sin(double x)` 返回 x 的正弦值。

`double asin(double x)` 返回 x 的反正弦值。

`double cos(double x)` 返回 x 的余弦值。

`double acos(double x)` 返回 x 的反余弦值。

`double tan(double x)` 返回 x 的正切值。

`double atan(double x)` 返回 x 的反正切值。

`double rand()` 返回 $[0, 1)$ 内的随机浮点数。

`int rand32(int min, int`

`max)`

返回 $[\text{min}, \text{max})$ 内的一个随机 32 位整数。用户可以只传入一个参数，该参数会判定为 `max`，此时 `min` 默认为 0。如果不传入参数，此时会从带符号的 32 位 `int` 范围内随机返回。

`int rand64(int min, int`

`max)`

返回 $[\text{min}, \text{max})$ 内的一个随机 64 位整数。用户可以只传入一个参数，该参数会判定为 `max`，此时 `min` 默认为 0。如果不传入参数，此时会从带符号的 64 位 `int` 范围内随机返回。

`collect()` 将收集的所有值放在一个列表中。

`avg()` 返回参数的平均值。

`count()` 返回参数的数量。

`max()` 返回参数的最大值。

`min()` 返回参数的最小值。

`std()` 返回参数的总体标准差。

`sum()` 返回参数的和。

`bit_and()` 逐位做 AND 操作。

`bit_or()` 逐位做 OR 操作。

`bit_xor()` 逐位做 XOR 操作。

`int size()` 返回列表或映射中元素的数量。

`int range(int start, int
end, int step)`

返回`[start,end]` 中指定步长的值组成的列表。步长`step` 默认为 1。

`int sign(double x)` 返回 x 的正负号。 如果 x 为0，则返回0。 如果 x 为负数，则返回-1。 如果 x 为正数，则返回1。

`double e()` 返回自然对数的底 e (2.718281828459045)。

`double pi()` 返回数学常数 π (3.141592653589793)。

`double radians()` 将角度转换为弧度。`radians(180)` 返回 3.141592653589793。

字符串函数

函数说明

`int strcasecmp(string a, string b)` 比较两个字符串（不区分大小写）。当 $a=b$ 时，返回 0，当 $a>b$ 是，返回大于 0 的数，当

$a<b$ 时，返回小于 0 的数。

`string lower(string a)` 返回小写形式的字符串。

`string toLower(string a)` 和`lower()` 相同。

`string upper(string a)` 返回大写形式的字符串。

`string toUpper(string a)` 和`upper()` 相同。

`int length(string a)` 以字节为单位，返回给定字符串的长度。

`string trim(string a)` 删除字符串头部和尾部的空格。

`string ltrim(string a)` 删除字符串头部的空格。

`string rtrim(string a)` 删除字符串尾部的空格。

`string left(string a, int count)` 返回字符串左侧`count` 个字符组成的子字符串。如果`count` 超过字符串 a 的长度，则返回字符串

a 。

`string right(string a, int count)` 返回字符串右侧`count` 个字符组成的子字符串。如果`count` 超过字符串 a 的长度，则返回字符串

a 。

`string lpad(string a, int size, string
letters)`

在字符串 a 的左侧填充`letters` 字符串，并返回`size` 长度的字符串。

`string rpad(string a, int size, string
letters)`

在字符串 a 的右侧填充`letters` 字符串，并返回`size` 长度的字符串。


```
string substr(string a, int pos, int count)
```

从字符串 `a` 的指定位置`pos` 开始（不包括`pos` 位置的字符），提取右侧的`count` 个字符，组成新的字符串并返回。

```
string substring(string a, int pos, int count)
```

和`substr()` 相同。

```
string reverse(string)  逆序返回字符串。
```

```
string replace(string a, string b, string c)
```

将字符串 `a` 中的子字符串 `b` 替换为字符串 `c`。

```
list split(string a, string b)  在子字符串 b 处拆分字符串 a，返回一个字符串列表。
```

```
string toString()  将任意数据类型转换为字符串类型。
```

```
int hash()  获取任意对象的哈希值。
```

日期时间函数

```
int now()  根据当前系统返回当前时区的时间戳。
```

```
timestamp timestamp()  根据当前系统返回当前时区的时间戳。
```

```
date date()  根据当前系统返回当前日期（UTC 时间）。
```

```
time time()  根据当前系统返回当前时间（UTC 时间）。
```

```
datetime datetime()  根据当前系统返回当前日期和时间（UTC 时间）。
```

Schema 函数

函数说明

```
id(vertex)  返回点 ID。数据类型和点 ID 的类型保持一致。
```

```
map properties(vertex)  返回点的所有属性。
```

```
map properties(edge)  返回边的所有属性。
```

```
string type(edge)  返回边的 Edge type。
```

```
src(edge)  返回边的起始点 ID。数据类型和点 ID 的类型保持一致。
```

```
dst(edge)  返回边的目的点 ID。数据类型和点 ID 的类型保持一致。
```

```
int rank(edge)  返回边的 Rank 值。
```

列表函数

函数说明

```
keys(expr)  返回一个列表，包含字符串形式的点、边或映射的所有属性。
```

```
labels(vertex)  返回点的 Tag 列表。
```

```
nodes(path)  返回路径中所有点的列表。
```

```
range(start, end [, step])  返回[start,end] 范围内固定步长的列表，默认步长step 为 1。
```

```
relationships(path)  返回路径中所有关系的列表。
```

```
reverse(list)  返回将原列表逆序排列的新列表。
```

```
tail(list)  返回不包含原列表第一个元素的新列表。
```

```
head(list)  返回列表的第一个元素。
```

`last(list)` 返回列表的最后一个元素。

`coalesce(list)` 返回列表中第一个非空元素。

聚合函数

Count 函数

函数说明

`count()` 语法： `count({expr | *})` 。

`count()` 返回总行数（包括 NULL）。

`count(expr)` 返回满足表达式的非空值的总数。

`count()` 和 `size()` 是不同的。

Collect 函数

函数说明

`collect()` `collect()` 函数返回一个符合表达式返回结果的列表。该函数可以将多条记录或值合并进一个列表，实现数据聚合。

Reduce 函数

函数说明

`reduce(<accumulator> = <initial>,`

`<variable> IN <list>`

`reduce()` 将表达式逐个应用于列表中的元素，然后和累加器中的当前结果累加，最后返回完整结果。

Hash 函数

函数说明

`hash()` 函数返回参数的哈希值。其参数可以是数字、字符串、列表、布尔值、NULL 等类型的值，或者计算结果为这些类型的

表达式。`hash()` 函数采用 MurmurHash2 算法，种子（seed）为 0xc70f6907UL 。用户可以在 MurmurHash2.h 中查看其源代码。

Concat 函数

函数说明

concat() 函数至少需要两个或以上字符串参数，并将所有参数连接成一个字符串。

语法： concat(string1,string2,...)

concat_ws 函数

函数说明

concat_ws() 函数将两个或以上字符串参数与预定义的分隔符（separator）相连接。

谓词函数

函数说明

谓词函数只返回 true 或 false ， 通常用于 WHERE 子句中。

exists() 如果指定的属性在点、边或映射中存在， 则返回 true ， 否则返回 false 。

any() 如果指定的谓词适用于列表中的至少一个元素， 则返回 true ， 否则返回 false 。

all() 如果指定的谓词适用于列表中的每个元素， 则返回 true ， 否则返回 false 。

none() 如果指定的谓词不适用于列表中的任何一个元素， 则返回 true ， 否则返回 false 。

single() 如果指定的谓词适用于列表中的唯一一个元素， 则返回 true ， 否则返回 false 。