超擎 SuperScylla 时空数据库软件 V1.0

产品

用户手册

苏州超擎图形软件科技发展有限公司

2020 年 7 月

文档信息

项目名称：			
系统名称：			
修订人：		文档版本号：	1.0
职位：		文档版本日期：	2020 年 07 月 20 日
审核人：		审核日期：	

文档分发列表

分发人	**日期**	**电话/传真/电子邮件**

接收人	**活动类型***	**归还日期**	**电话/传真/电子邮件**

**活动类型：批准、审核、通知、归档、活动请求、参加会议、其他（请详细说明）*

文档版本履历

版本号	**修订时间**	**修订人**	**审核人**	**批准人**	**修改内容**
V1.0	2020.7.20	郭志忠	钱柯健		创建版本

文档版权说明

目录

1 产品介绍

SuperScylla 是一款面向时空动态数据的分布式 NoSQL 数据库，提供数据接入、数据管理、数据查询等数据管理基础能力。

SuperScylla 作为一款数据库级别的产品，可广泛应用于物联网数据管理、轨迹数据管理、海量数据高并发写入等场景，尤其适合大量写入、统计和分析的场景。

2 使用指南

2.1 客户端 Cqlsh 的安装与使用

安装：

在 SuperScylla 安装完成后会自动安装 cqlsh。

在未安装 SuperScylla 的机器上，使用 yum install -y cqlsh 进行安装。

连接与运行：

语法：

cqlsh ip port（ip 与 port 间的空格不能省略）

示例：cqlsh 192.168.1.201 9042

退出：

在 cqlsh 命令行输入 exit 即可退出 cqlsh。

2.2 数据定义（DDL）

2.2.1 创建 KeySpace

SuperScylla 中 KEYSPACE 的概念等同于关系数据库的 Database 概念。通过 KEYSPACE 对数据管理，通过 create keyspace/alter keyspace/drop keyspace 来实现创建/修改/删除 keyspace，参数包括名称、配置选项 option。

语法：

```
CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name WITH options;
```

例如：

```
CREATE KEYSPACE excelsior
WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};


CREATE KEYSPACE excalibur
WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 3}
AND durable_writes = false;
```

支持的 options 有：

名称	类型	必须	缺省值	描述
replication	map	yes		用于 KeySpace 的复制策略
durable_writes	simple	no	true	是否将 CommitLog 用于更新此 KeySpace（强烈建议启用）。

- Replication

replication 是强制性的，并且必须至少包含“ class”子选项，该子选项定义要使用的replication类。其余子选项取决于所使用的复制策略。默认情况下,支持以下“类”:

SimpleStrategy：在一个数据中心的情况下使用简单的策略。 在这个策略中，第一个副本被放置在所选择的节点上，剩下的节点被放置在环的顺时针方向，而不考虑机架或节点的位置。

NetworkTopologyStrategy：该策略用于多个数据中心。 在此策略中，您必须分别为每个数据中心提供复制因子。

replication_factor：

复制因子是放置在不同节点上的数据的副本数。 超过两个复制 因子是很好的获得没有单点故障。所以 3 个以上是很好的复制因子。

- durable_writes

默认情况下，表的 durable_writes 属性设置为 true，但可以将其设置为 false（即设置为持久写入）。当 Durable_writes 设 置为 false 时，策略不能设置为简单策略。

示例：

```
CREATE KeySpace lkse WITH REPLICATION = {'class':'SimpleStrategy', 'replication_factor' : 3};（常用）
```

```
CREATE KeySpace lkse WITH REPLICATION = {'class':'SimpleStrategy', 'replication_factor' : 3} AND DURABLE_WRITES =false;
```

查询 KeySpace 列表

语法：DESCRIBE keyspaces;

使用 KeySpace

语法：USE <identifier>;

示例：USE lkse;

查询指定 KeySpace 信息

语法：DESCRIBE <identifier>;

示例：DESCRIBE lkse;

2.2.2 创建 Table

SuperScylla 支持创建/修改/删除表，通过 create table/alter table/drop table 来实现，参数包括名称、定义类型的列、主键列或主键列的组合，以及其他选项。

在表中使用 geopoint 类型，需要遵循系统内置的写法（frozen<geopoint>）：

```
create table demo(id int,name text,geo frozen<geopoint>);
```

geopoint 类型的列，需要设置为 Partition Key，才能进行空间查询，否则只能作为普通数值列。无法为 geopoint 类型创建 Secondary Index 和 Clustering Key。SuperScylla 允许建表时候，指定 geopoint 数据的分块范围，用来处理不同密集程度的

数据。

通过 cql 语句：create table demo(id int, geo frozen<geopoint>) with geohash_level = x;

x 是整数，取值范围在 1-10。如果不指定 geohash_level,则 geohash_level 默认值为 6;

SuperScylla 采用 geohash 索引，对数据进行空间划分，geohash_level 越小，每块的范围越大,反之每块的范围越小。geohash 范围大,则数据误读多,查询效率低。geohash 范围小，则存贮碎片化严重，网络通讯代价高。（实际测试表明，geohash 越大，数据写入速度越快）

因此评估你的数据范围，选择合适的 geohash_level 尤为重要。结合场景，我们推荐配置如下：

广场、建筑内的应用，如区域人员监控、定位等。推荐 geohash_level 设置为 7、8;

城市级别的数据应用，如出租车轨迹。推荐 geohash_level 设置为 5、6;

全国范围的应用，如火车、飞机等轨迹。推荐 geohash_level 设置为 3、4;

下表为不同 geohash_level ，每个网格代表的地理范围，常用取值有 3 - 8;

geohash_level	地理范围
1	≤ 5,000km
2	≤ 1,250km
3	≤ 156km
4	≤ 39.1km
5	≤ 4.89km
6	≤ 1.22km
7	≤ 153m
8	≤ 38.2m
9	≤ 4.77m
10	≤ 1.19m

语法：

```
CREATE (TABLE | COLUMNFAMILY) <tablename>('<column-definition>', '<column-definition>')(WITH <option> AND <option>)
```

示例：

单个主键:

```
create table demo(id int primary key,name text,geo frozen<geopoint>);
create table demo(id int ,name text,geo frozen<geopoint>,primary key(id));
```

复合主键:

```
create table demo(uuid timeuuid,utc bigint,point frozen<geopoint>,valid int,primary(point,utc,uuid));
```

2.2.3 SCHEMA

SuperScylla 支持通过 desc 命令来查看 schema 信息，包括 keyspace、table。

命令	功能
desc keyspaces;	列出当前所有 keyspace
desc keyspace 名称;	打印指定 keyspace 的 schema 信息
desc tables;	列出当前 keyspace 中所有 table
desc table 名称;	打印指定 table 的 schema 信息

2.3 创建 UDT

SuperScylla 支持用户创建自定的数据类型（User Define Type，简称 UDT），通过 create type/alter type/drop type 语句来创建/修改/删除类型。一旦创建后，UDT 将通过类型名称进行引用。

UDT 包括名称、具有命名和类型的字段组成，字段可以是任意类型，包括集合或其他 UDT。

在 SuperScylla 使用 geopoint 类型，需要通过 UDT 实现，即在 KeySpace 里创建 UDT 的 geopoint。

语法：

```
create type type_name(col_name data_type,col_name data_type);
```

示例：

```
create type geopoint(x double,y double);
```

（geopoint 类型须使用上述语句创建自定义类型）

查看已创建自定义类型的信息：

DESCRIBE type_name;

2.4 数据操作（DML）

2.4.1 查询操作

支持基本条件、空间条件对 table 进行条件过滤，非主键字段需要配合 Secondary index 来实现。

- 查询条件支持：

  Select * from table_name where clause;

  1）基本条件：等于、大于、小于、IN

  2）空间条件：intersects() 、contains() 、within() 、coverby() 、nearby()

- 查询函数支持：

  Select function(field) from geo_table;

  基本函数：count()、max() 、min() 、sum() 、avg()

  空间函数： distance()、as_wkt()、string()

针对没有建索引的字段查询，SuperScylla 需要进行全表扫描过滤，这种情况在海量数据时，响应时间可能很长。因此对于无法通过索引命中的查询，需要在 where 语句后面加上 ALLOW FILTERING 语句，允许数据库通过扫表过滤数据。

如： select * from table where b > 1 allow filtering （b 是普通列）

语法：

```
SELECT [ DISTINCT ] ( select_clause | '*' )
FROM table_name
[ WHERE where_clause ]
[ ORDER BY ordering_clause ]
[ PER PARTITION LIMIT (integer | bind_marker) ]
[ LIMIT (integer | bind_marker) ]
[ ALLOW FILTERING ]
[ BYPASS CACHE ]
select_clause ::= selector [ AS identifier ] ( ',' selector [ AS identifier ] )
selector ::= column_name
| term
| CAST '(' selector AS cql_type ')'
| function_name '(' [ selector ( ',' selector )* ] ')'
| COUNT '(' '*' ')'
where_clause ::= relation ( AND relation )*
relation ::= column_name operator term
'(' column_name ( ',' column_name )* ')' operator tuple_literal
TOKEN '(' column_name ( ',' column_name )* ')' operator term
operator ::= '=' | '<' | '>' | '<=' | '>=' | '!=' | IN | CONTAINS | CONTAINS
KEY
ordering_clause ::= column_name [ ASC | DESC ] ( ',' column_name [ ASC | DESC ] )*
```

例如：

```
SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);

SELECT name AS user_name, occupation AS user_occupation FROM users;

SELECT time, value
FROM events
WHERE event_type = 'myEvent'
AND time > '2011-02-03'
```

```
AND time <= '2012-01-01'


SELECT COUNT (*) AS user_count FROM users;
```

其中空间查询有以下查询算子：

- **intersects 查询**

  语法：

  ```
  select * from table where column_name.intersects(GEO_WKT);
  ```

  其中，column_name 必须是 geopoint 类型列，GEO_WKT 支持点、线、面几何类型。

  示例：

  ```
  select * from demo where point.intersects('POINT(120.122 32.431)');
  ```

- **contains 查询**

  语法：

  ```
  select * from table where column_name.contains(GEO_WKT);
  ```

  其中，column_name 必须是 geopoint 类型列，GEO_WKT 支持点、线、面几何类型。

  示例：

  ```
  select * from demo where point.contains('POINT(120.122 32.431)');
  ```

- **within 查询**

  语法：

  ```
  select * from table where column_name.within(GEO_WKT);
  ```

  其中，column_name 必须是 geopoint 类型列，GEO_WKT 支持点、线、面几何类型。

  示例：

  ```
  select * from demo where point.within('POLYGON((120.122 32.431,120.134
  32.442,120.136 32.421))');
  ```

- coverby 查询

语法：

```
select * from table where column_name.coverby(GEO_WKT);
```

其中，column_name 必须是 geopoint 类型列，GEO_WKT 支持点、线、面几何类型。

示例：

```
select * from demo where point.coverby('POLYGON((120.122 32.431,120.134 32.442,120.136 32.421))');
```

- nearby 查询

语法：

```
select * from table where column_name.nearby(GEO_WKT, meter);
```

其中，column_name 必须是 geopoint 类型列，GEO_WKT 支持点几何类型，meter 为 double 型。

示例：

```
select * from table where column_name.nearby(GEO_WKT, meter);
```

- count 函数

语法：

```
select count(*) from table where clause;
```

返回符合条件的行数。

示例：

```
select count(*) from demo where point.intersects('POINT(120.122 32.431)');
```

- distance 函数

语法：

```
select distance(column_name, GEO_WKT) from table;
```

其中，column_name 必须是 geopoint 类型列，GEO_WKT 支持点几何类型。返回 double

型距离值（米）。

示例：

```
select distance(point, ‘POINT(120.122 32.431)’) from demo;
```

- **as_wkt 函数**

语法：

```
select as_wkt(column_name) from table;
```

其中，column_name 必须是 geopoint 类型列。返回几何列的 WKT 值。

示例：

```
select as_wkt(point) from demo;
```

- **string 函数**

语法：

```
select string(column_name) from table;
```

其中，column_name 支持基本类型，不支持 geopoint 类型列。返回列值字符串。

示例：

```
select string(id) from demo;
```

2.4.2 插入操作

插入数据支持单条 insert sql，也支持批量插入模式（Batch）。通过批量插入模式，加速数据写入。

插入语法如下：

```
INSERT INTO `table_name` ( `names_values` )
: [ USING `update_parameter` ( AND `update_parameter` )* ]

names_values: `names` VALUES `tuple_literal`
```

```
names: '(' `column_name` ( ',' `column_name` )* ')'
```

例如：

```
INSERT INTO NerdMovies (movie, director, main_actor, year)
VALUES ('Serenity', 'Joss Whedon', 'Nathan Fillion', 2005)
USING TTL 86400;
```

2.4.3 删除操作

Delete 操作的 where 条件仅支持 primary key，且构成 primary key 的列必须要写全，这是因为 delete 语句的执行过程需要从海量数据中快速定位到数据地址。

如：delete from table where a = 1 （a 是 pk 列）

删除语法如下：

```
DELETE [ `simple_selection` ( ',' `simple_selection` ) ]
: FROM `table_name`
: [ USING `update_parameter` ( AND `update_parameter` )* ]
: WHERE `where_clause`
: [ IF ( EXISTS | `condition` ( AND `condition` )*) ]
```

例如：

```
DELETE FROM NerdMovies USING TIMESTAMP 1240003134
WHERE movie = 'Serenity';

DELETE phone FROM Users
WHERE userid IN (C73DE1D3-AF08-40F3-B124-3FF3E5109F22,
B70DE1D0-9908-4AE3-BE34-5573E5B09F14);
```

2.4.4 更新操作

update 操作的 where 条件仅支持 primary key，且构成 primary key 的列必须要写全，这是因为 update 语句的执行过程需要从海量数据中快速定位到数据地址。

如：update table set b = 2 where a = 1 （a 是 pk 列，b 是普通列）

更新语法：

```
UPDATE `table_name`
: [ USING `update_parameter` ( AND `update_parameter` )* ]
: SET `assignment` ( ',' `assignment` )*
: WHERE `where_clause`
: [ IF ( EXISTS | `condition` ( AND `condition` )*) ]

update_parameter: ( TIMESTAMP | TTL ) ( `integer` | `bind_marker` )


assignment: `simple_selection` '=' `term`
:| `column_name` '=' `column_name` ( '+' | '-' ) `term`
:| `column_name` '=' `list_literal` '+' `column_name`


simple_selection: `column_name`
:| `column_name` '[' `term` ']'
:| `column_name` '.' `field_name


condition: `simple_selection` `operator` `term`
```

示例：

```
UPDATE NerdMovies USING TTL 400
SET director = 'Joss Whedon',
main_actor = 'Nathan Fillion',
year = 2005
WHERE movie = 'Serenity';

UPDATE UserActions
SET total = total + 2
WHERE user = B70DE1D0-9908-4AE3-BE34-5573E5B09F14
AND action = 'click';
```

备注：

更新语句的 where 表达式必须使用 Primary Key 进行过滤。

2.4.5 批量模式

语法：

```
BEGIN [ UNLOGGED | COUNTER ] BATCH
: [ USING `update_parameter` ( AND `update_parameter` )* ]
: `modification_statement` ( ';' `modification_statement` )*
: APPLY BATCH

modification_statement: `insert_statement` | `update_statement` |
`delete_statement`
```

示例：

```
BEGIN BATCH
INSERT INTO users (userid, password, name) VALUES ('user2', 'ch@ngem3b',
'second user');
UPDATE users SET password = 'ps22dhds' WHERE userid = 'user3';
INSERT INTO users (userid, password) VALUES ('user4', 'ch@ngem3c');
DELETE name FROM users WHERE userid = 'user1';
APPLY BATCH;
```

2.5 索引指南

SuperScylla 的索引概念有 primary key，partition key，clustering key，Secondary index 等四种。

2.5.1 PRIMARY KEY

是一个宏观概念，用于从表中取出数据（可以理解为一级索引），primary key 可以由一个或多个 column 组合而成，当 primary key 由 多列组合而成时，即为 composite primary key（多列 key）。

Primary Key 由 Partition Key 和 Clustering Key 组成， Partition Key 决定了数据的分区机制，Clustering Key 列可以支持范围过滤查询。

支持以下几种模式：

a. primary key(id)：单列 partition key，只能=、in 条件查询；

b. primary key(id, time)：单列 partition key、单列 clustering key，time 是 clustering key，time 列可以范围查询。

c. primary key((id,age), time)：组合列 partition key、单列 clustering key，（id,age）为 partition key，time 为 clustering key；

d. primary key(id, age, time)：单列 partition key、多列 clustering key，id 为 partition key，（age, time）为 clustering key；

例如：

primary key (key_part_one) ;

primary key (key_part_one,key_part_two);

在组合主键的情况下，第一部分称作 partition key（key_part_one 即为 partition key），第二部分是 clustering key（key_part_two）。

示例：

单列做主键

```
create table demo1(id int,name text,primary key(id));
```

多列主键

```
create table demo1(id int,name text,age int,primary key(id,age));
```

此示例中的 id 即为 partition key，age 即为 clustering key。

多列 partition key 及多列 clustering key 做主键

```
create table demo1(id int,name text,age int,x float,y float,uuid timeuuid,
primary key((id,age),(x,y));
```

2.5.2 PARTITION KEY

负责将数据分布在集群节点上，可以由一列或多列组成。当 primary key 在表中的 key 只有一个 field 的情况下，其与 partition key 是等效的。

2.5.3 CLUSTERING KEY

负责在 partition 中的数据排序，可以由一列或多列组成。

SuperScylla 中所有的数据都只能根据 Primary Key 中的字段来排序。 因此，如果想根据某个 column 来排序，必须将改 column 加到 Primary key 中，如 primary key (id, c1, c2 ,c3)，其中 id 时 partition key, c1, c2 ,c3 是 Clustering Key。

如果想用 id 和 c1 作为 partition key，只需添加括号写成: primary key ((id, c1), c2 ,c3))即可。

2.5.4 SECONDARY INDEX

SuperScylla 中除了 primary key 外，还有二级索引，作为辅助索引就是通过找到一级索引，进而获取数据。

二级索引支持非 primary key 列的查询，仅支持等于、in 查询。建议列值控制在一定个数范围内，否则索引意义不大。

分布式数据库，需要用户根据自己的场景，设计合理的 schema，以达到最高性能。

Partition Key 应当选择列值不多、能区分出业务单元的字段。例如用城市名称来做 partition key，将不同城市数据放到不同分区，提高数据库性能。

Clustering Key 用来满足需要范围过滤的字段，如数值类型、时间类型等字段，Clustering key 必须建表时候指定，不能后期变更。

Secondary Index 用来做辅助索引，仅支持 =、in 查询，适合 key=value 类型的场景，value 的可能数值量尽量少。

语法：CREATE INDEX <identifier> ON <tablename>;

CREATE INDEX 命令用于在用户指定的列上创建一个索引。如果指定的列已存在数据，在执行 CREATE INDEX 后，则会在指定数据列上创建索引。

创建索引的规则：

1. 由于主键已编入索引，无法在主键列上创建二级索引。

2. 二级索引不支持集合索引，如 create index on table(id,age);

示例：

创建索引：

```
CREATE INDEX test_index ON demo (valid);
```

查询：

```
select * from demo where valid=xxx;
```

二级索引的查询只支持“=”、”in”查询。

3 使用示例

3.1 连接 SuperScylla

执行命令 `cqlsh 192.168.1.199 9042`

```
[root@localhost ~]# cqlsh 192.168.1.199 9042
Connected to  at 192.168.1.199:9042.
[cqlsh 5.0.1 | Cassandra 3.1.2 | CQL spec 3.3.1 | Native protocol v4]
Use HELP for help.
cqlsh>
```

3.2 创建 KeySpace

执行 cql

`CREATE KeySpace ks WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 3};`

```
cqlsh> CREATE KEYSPACE ks WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 3};
cqlsh>
```

3.3 查询 KeySpace 集合

执行 cql `Desc keyspaces;`

```
cqlsh> desc keyspaces;

ks             system_auth  se_hn              se_jsjk
system_schema  system       system_distributed  system_traces

cqlsh>
```

3.4 使用指定的 KeySpace

执行 cql `use ks;`

```
cqlsh> use ks;
cqlsh:ks>
```

3.5 创建 geopoint 类型

执行 cql `create type geopoint(x double,y double);`

```
cqlsh:ks> create type geopoint(x double,y double);
cqlsh:ks>
```

3.6 创建表

执行 cql

```
create table t0 (id int,name text,geo frozen<geopoint>, primary key(geo,id));
```

```
cqlsh:ks> create table t0 (id int,name text,geo frozen<geopoint>, primary key(geo,id));
cqlsh:ks>
```

3.7 查询 KeySpace 内的表集合

执行 cql `desc tables;`

```
cqlsh:ks> desc tables;

t0

cqlsh:ks>
```

3.8 插入数据

执行 cql

```
insert into t0 (id, name, geo ) values ( 0, 'a', (50,50));

insert into t0 (id, name, geo ) values ( 1, 'b', (60,60));
```

```
cqlsh:ks> insert into t0 (id, name, geo ) values ( 0, 'a', (50,50));
cqlsh:ks> insert into t0 (id, name, geo ) values ( 1, 'b', (60,60));
cqlsh:ks>
```

3.9 执行查询

3.9.1 查询和指定点相交的记录

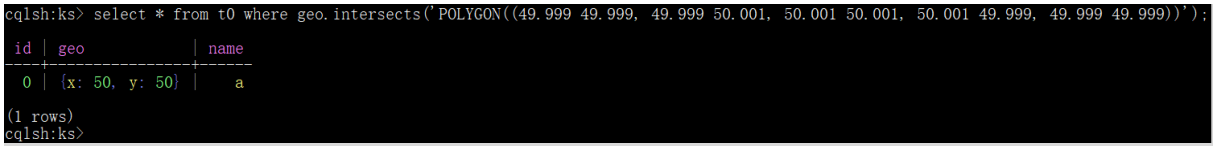
执行 cql `select * from t0 where geo.intersects('POINT(50 50)');`

```
cqlsh:ks> select * from t0 where geo.intersects('POINT(50 50)');

 id | geo             | name
----+-----------------+------
  0 | {x: 50, y: 50} |    a

(1 rows)
cqlsh:ks>
```

3.9.2 查询和指定区域相交的记录

执行 cql

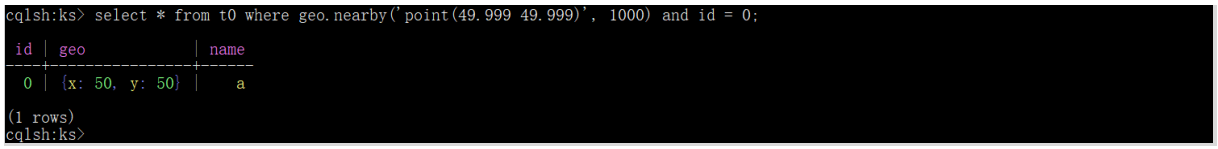
```
select * from t0 where geo.intersects('POLYGON((49.999 49.999, 49.999 50.001, 50.001 50.001, 50.001 49.999, 49.999 49.999))');
```



3.9.3 查询指定点距离范围内的记录

执行 cql

```
select * from t0 where geo.nearby('point(49.999 49.999)', 1000) and id = 0;
```

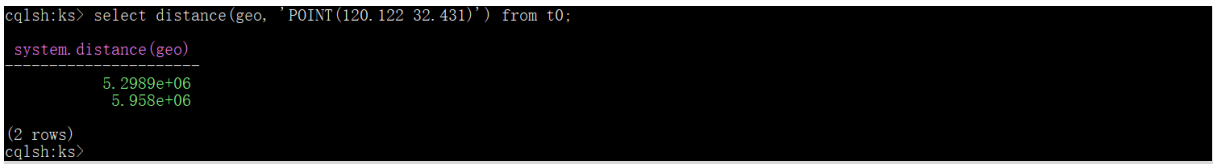


3.9.4 以 wkt 格式输出

执行 cql `select as_wkt(geo) from t0;`

3.9.5 查询到指定点的距离

执行 cql `select distance(geo, 'POINT(120.122 32.431)') from t0;`



3.10 删除表

执行 cql `drop table t0;`

```
cqlsh:ks> drop table t0;
cqlsh:ks>
```

3.11 删除 KeySpace

执行 cql `drop keyspace ks;`

```
cqlsh:ks> drop keyspace ks;
cqlsh:ks> desc keyspaces;

se_hn          system_auth  system_distributed  system_traces
system_schema  system       se_jsjk

cqlsh:ks>
```