Transwarp Data Hub Version 4.2

Hyperbase使用手册

星环信息科技（上海）有限公司

Transwarp Data Hub v4.2

Hyperbase使用手册

版本：1.1v

发布日期： 2015-12-01

版本号：T00142-04-011

免责声明

本说明书依据现有信息制作，其内容如有更改，恕不另行通知。星环信息科技（上海）有限公司在编写该说明书的时候已尽最大努力保证期内容准确可靠，但星环信息科技（上海）有限公司不对本说明书中的遗漏、不准确或印刷错误导致的损失和损害承担责任。具体产品使用请以实际使用为准。

注释：Hadoop® 和 SPARK® 是Apache™ 软件基金会在美国和其他国家的商标或注册的商标。 Java® 是Oracle公司在美国和其他国家的商标或注册的商标。 Intel® 和Xeon® 是英特尔公司在美国、中国和其他国家的商标或注册的商标。

修订历史记录

修改记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

文档版本T00142-04-011 (2015-12) 第一次发布。

文档版本T00142-04-010 (2015-11) 第二次发布。

文档版本T00142-04-010 (2015-06) 第一次发布。

文档版本T00141-04-020 (2015-04) 第一次发布。

文档版本T00141-04-020 (2015-04) 第一次发布。

文档版本T00140-04-010 (2014-12) 第一次发布。

目录

1. 简介

1.1. 公司介绍

星环信息科技(上海)有限公司是目前中国国内极少数掌握企业级大数据Hadoop和Spark核心技术的高科技公司，从事大数据时代核心平台数据库软件的研发与服务。Apache Hadoop技术已成为公认的替代传统数据库的大数据产品。公司产品Transwarp Data Hub (TDH)的整体架构及功能特性比肩硅谷同行，产品性能在业界处于领先水平。

1.2. Transwarp Hyberbase介绍

Transwarp Hyperbase实时数据库是建立在Apache HBase基础之上，融合了多种索引技术、分布式事务处理、全文实时搜索、图形数据库在内的实时NoSQL数据库。

在本文档中，我们将简要概述HBase的数据模型，如何通过和HBase Shell交互建Hyperbase表、读表和写入数据。然后我们将介绍如何在Inceptor中创建Hyperbase表的映射表，并且对映射表进行SQL分析。

2. 数据模型

2.1. 简介

Hyperbase以“表”为结构来组织数据，“表”中有“行”和“列”，但是Hyperbase中的这些概念和关系型数据库的二维表不同。在谷歌的Bigtable论文中，Bigtable被描述为一个“稀疏的，分布式的，持久的，多维度有序map”。下面，我们围绕这个描述来解说Hyperbase中的数据模型。

Hyperbase中图表对象如下：

- **表（Table）**：Hyperbase以表为单位组织数据。表名的数据类型为string。
- **行（Row）**：：表中数据以行存储。每行数据都有一个独特的RowKey。表中各行数据按RowKey排序。Row key没有数据类型，以byte[]（字节数组）存储。
- **列族（Column Family）**：行中数据以列族分组。各行数据拥有的列族必须相同。但是并不是每个列族中都需要有数据。列族名的数据类型为string。
- **列限定符（Column Qualifier）**：列族中可以有一列或者多列数据。各列根据列限定符识别。各行的拥有的列不一定需要相同。列名没有数据类型，以byte[]存储。
- **单元格（Cell）**：行、列族和列限定符的组合指向独特的单元格。单元格中存放的数据成为单元格的值。单元格的值没有数据类型，以byte[]存储。
- **时间戳（Timestamp）**：单元格的值可以有不同版本。各个版本由版本号区分。默认版本号为单元格值被写入时的时间戳。

为了更好地理解这些概念，下面我们举一个Hyperbase表的例子。

2.2. 例子

Row Key（Account Number）	Column Family - Personal		Column Family - Contact		Column Family - Balance	Timestamp
	column qualifier - name	column qualifier - password	column qualifier - email	column qualifier - cellphone	column qualifier - balance	
0001		5678				t16
					10000.00	t05
				12345678912		t04
			zs@mail.com			t03
		1234				t02
	Zhang San					t01
0002					56000.00	t18
			ls@transwarp.io			t17
					1000.00	t10
				13513572468		t09
			ls@school.edu			t08
		2468				t07
	Li Si					t06
0003					500.00	t15
				13612345678		t14
			ww@hmail.com			t13
		1357				t12
	Wang Wu					t11

该表模拟银行用户表中的部分信息。表的Row Key为账户号码Account Number；各行以账户号码排序。该表有三个列族：Personal，Contact和Balance。Personal列族含有两列，

它们的列限定符为name和password。Contact列族含有两列，它们的列限定符为email和cellphone。Balance列族含有一列，它的列限定符为balance。RowKey为0001的行有两个版本，时间戳分别为t3和t1。RowKey为0002的行有三个版本，时间戳分别为t8, t5, t2。RowKey为0003的行有一个版本，时间戳为t6。不同版本的单元格值按时间戳降序排列，方便读取最新的值。表中无值的单元格在系统中是不存在的，表的“稀疏性”就体现在这里。

在Hyperbase中，表不是二维的，而是一个嵌套的map，由多层 **键值对** 组成：

```
{Table: {RowKey: {ColumnFamily: {ColumnQualifier: {Timestamp, Value}}}}}
```

所以，bank_info表可以如下表示：

```
bank_info:{
    0001:{
        Personal: {
            name: {t1: Zhang San}
            password: {t3: 5678
                       t1: 1234}
                  }
        Contact: {
            email: {t1: zs@mail.com}
            cellphone: {t1: 12345678912}
                  }
        Balance: {
            balance: {t1: 10000.00}
                  }
          }
    0002: {
        Personal: {
            name: {t2: Li Si}
            password: {t2: 2468}
                  }
        Contact: {
            email: {t5: ls@transwarp.io
                    t2: ls@school.edu
                   }
            cellphone: {t2: 13513572468}
                  }
        Balance: {
            balance: {t8: 56000.00
                      t2: 1000.00}
                  }
          }
    0003: {
        Personal: {
            name: {t6: Wang Wu}
            password: {t6: 1357}
                  }
        Contact: {
            email: {t6: ww@hmail.com}
            cellphone: {t6: 13612345678}
                  }
        Balance: {
            balance: {t6: 500.00}
                  }
```

```
  }
}
```

3. Hyperbase命令行快速入门

执行’hbase shell’操作可以进入Hyperbase命令行和Hyperbase进行简单交互。本章中，我们介绍如何通过Hyperbase Shell进行一些简单操作，包括建表，填入数据和删除数据。

和Inceptor命令行不同，Hyperbase的命令行指令区分大小写，例如create指令不能写成CREATE。

3.1. 建表

语法：create

```
create 'table_name', 'column_family_1', 'column_family_2', ...
```

现在我们以先前章节中的银行用户表为例建表：

举例

```
hbase(main):001:0> create 'bi', 'ps', 'ct', 'bl'
```

说明

表、列族和列限定符名都要尽量短，以减少对Hyperbase读写时的I/O负载。所以我们将bank_info, personal, contact, balance缩短为bi, ps, ct和bl。建表以后，可以通过describe语句查看表的描述：

```
hbase(main):008:0> describe 'bi'
DESCRIPTION
 ENABLED
 'bi',
 {NAME => 'bl', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
 REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1', TTL => 'FOR true
 EVER', MIN_VERSIONS => '0', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536',
 IN_MEMORY => 'false', BLOCKCACHE => 'true'},
 {NAME => 'ct', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
 REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1', TTL => 'FOREVER',
 MIN_VERSIONS => '0', KEE
 P_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE
 => 'true'},
 {NAME => 'ps', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
 REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1', TTL => 'FOREVER',
 MIN_VERSIONS => '0', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY
 => 'false', BLOCKCACHE => 'true'}
```

如我们所见，在表的描述中，列族有很多个参数，列族与列族间的参数由“{}”分隔开。我们将在之后的章节中解释其中重要的部分。建表时，可以通过直接指定列族参数来定义列族。所有的参数中只有NAME必须指定，其余参数为可选。所以上面的建表语句也可以像下面这样写：

```
hbase(main):001:0> create 'bi', {NAME=>'ps'}, {'NAME'=>'ct'}, {'NAME'=>'bl'}
```

3.2. 填入数据

下面，我们用put指令向表内填入数据：

语法：put

```
put 'table_name', 'row_key', 'column_family:column_qualifier', 'cell_value',
 timestamp
```

说明

在put语句末尾的timestamp是可选的。如果用户不指定timestamp，Hyperbase将根据put的时间给出一个timestamp。我们建议用户不要自己定义timestamp。

举例

```
put 'bi', '0001', 'ps:nm', 'Zhang San'
put 'bi', '0001', 'ps:pw', '1234'
put 'bi', '0001', 'ct:em', 'zs@mail.com'
put 'bi', '0001', 'ct:cp', '12345678912'
put 'bi', '0001', 'bl:bl', '10000.00'
put 'bi', '0002', 'ps:nm', 'Li Si'
put 'bi', '0002', 'ps:pw', '2468'
put 'bi', '0002', 'ct:em', 'ls@school.edu'
put 'bi', '0002', 'ct:cp', '13513572468'
put 'bi', '0002', 'bl:bl', '1000.00'
put 'bi', '0003', 'ps:nm', 'Wang Wu'
put 'bi', '0003', 'ps:pw', '1357'
put 'bi', '0003', 'ct:em', 'ww@hmail.com'
put 'bi', '0003', 'ct:cp', '13612345678'
put 'bi', '0003', 'bl:bl', '500.00'
put 'bi', '0001', 'ps:pw', '5678'      //用户Zhang San修改密码
put 'bi', '0002', 'ct:em', 'ls@transwarp.io'        //用户Li Si修改邮箱
put 'bi', '0002', 'bl:bl', 56000        //用户Li Si的存款发生改变
```

数据全部填写完成后可以用scan来查看表中数据

语法：scan

```
scan 'table_name'
```

举例

```
hbase(main):052:0> scan 'bi'
ROW                             COLUMN+CELL
 0001                           column=bl:bl, timestamp=1422973263541, value=10000.00
 0001                           column=ct:cp, timestamp=1422973240271,
 value=12345678912
 0001                           column=ct:em, timestamp=1422973219713,
 value=zs@mail.com
 0001                           column=ps:nm, timestamp=1422973047378, value=Zhang
 San
```

```
0001                          column=ps:pw, timestamp=1422973504458, value=5678
0002                          column=bl:bl, timestamp=1422973543652, value=56000
0002                          column=ct:cp, timestamp=1422973339893,
value=13513572468
0002                          column=ct:em, timestamp=1422973526791,
value=ls@transwarp.io
0002                          column=ps:nm, timestamp=1422973288836, value=Li Si
0002                          column=ps:pw, timestamp=1422973299902, value=2468
0003                          column=bl:bl, timestamp=1422973464663, value=500.00
0003                          column=ct:cp, timestamp=1422973432593,
value=13612345678
0003                          column=ct:em, timestamp=1422973416242,
value=ww@hmail.com
0003                          column=ps:nm, timestamp=1422973393169, value=Wang Wu
0003                          column=ps:pw, timestamp=1422973405207, value=1357
```

注意 ，scan时，Hyperbase只会显示单元格值中拥有最新timestamp的版本。

3.3. 删除数据

3.3.1. 删除单元格

为表删除一个单元格中的数据用delete指令，删除时，需要指定表名、row key，列族和列限定符。

语法：delete

```
delete 'table_name', 'row_key', 'column_family:column_qualifier'
```

3.3.2. 删除整行数据

将一行中的数据全部删除用deleteall指令，执行时，需要指定表名和row key。

语法：deleteall

```
deleteall 'table_name', 'row_key'
```

3.3.3. 删除一个列族

删除一整个列族用以下两个指令，效果相同。

语法：alter … 'delete'…

```
alter 'table_name', 'delete'=>'column_family_name'
```

```
alter 'table_name', NAME => 'column_family_name', METHOD => 'delete'
```

举例

下例删除bi表中的bl列族：

```
hbase(main):020:0> alter 'bi', 'delete' => 'bl'
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
```

3.4. 添加一个列族

建表后，我们还是可以向表添加新的列族，此时需要直接指定列族参数,只有NAME为必选参数，其余参数可选。

语法

```
alter 'table_name', {NAME=>'column_family_name,...}
```

举例 下例为bi表添加一个名为bl的列族。

```
hbase(main):054:0> alter 'bi', {NAME=>'bl'}
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
```

3.5. 删除表

在删除一张表之前，要先disable这张表：

语法：disable

```
disable 'table_name'
```

disable之后才能执行删除命令：

语法：drop

```
drop 'table_name'
```

举例 我们先试着删除一张名为’ta’的表：

```
hbase(main):014:0> drop 'ta'
ERROR: Table ta is enabled. Disable it first.'
Here is some help for this command:
Drop the named table. Table must first be disabled:
  hbase> drop 't1'
  hbase> drop 'ns1:t1'
```

Hyperbase 报错，要先disable表。

```
hbase(main):002:0> disable 'ta'
```

```
hbase(main):003:0> drop 'ta'
```

删除完成。在disable一张表以后，该表即不能使用，无法进行scan, put等操作。重新使用该表，要使用enable命令：

语法：enable

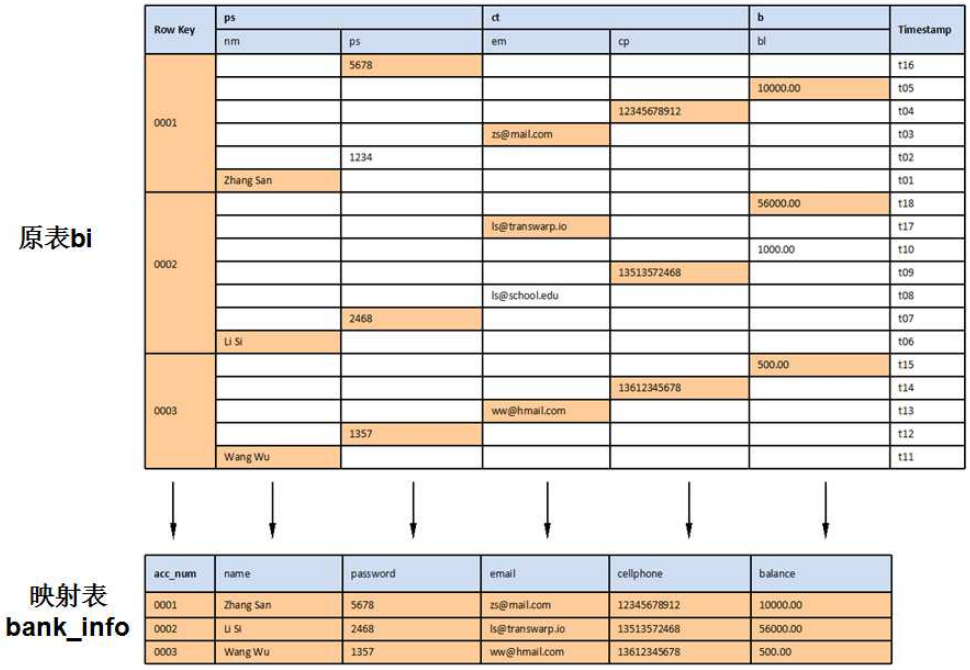
```
enable 'table_name'
```

4. 在Inceptor命令行中对Hyperbase表进行操作

要在Inceptor中对Hyperbase表进行交互式查询，要先在Inceptor中建一张外表，然后将Hyperbase表通过映射建立和一张二维表的对应关系。映射时，只有最新版本的单元格值会被保存。

举例

将bank_info映射为二维表的逻辑如下图所示：



Row Key	ps		ct		b	Timestamp
	nm	ps	em	cp	bl	
0001		5678				t16
					10000.00	t05
				12345678912		t04
			zs@mail.com			t03
		1234				t02
	Zhang San					t01
0002					56000.00	t18
			ls@transwarp.io			t17
					1000.00	t10
				13513572468		t09
			ls@school.edu			t08
		2468				t07
	Li Si					t06
0003					500.00	t15
				13612345678		t14
			ww@hmail.com			t13
		1357				t12
	Wang Wu					t11

acc_num	name	password	email	cellphone	balance
0001	Zhang San	5678	zs@mail.com	12345678912	10000.00
0002	Li Si	2468	ls@transwarp.io	13513572468	56000.00
0003	Wang Wu	1357	ww@hmail.com	13612345678	500.00

语法

```
CREATE EXTERNAL TABLE table_name (row_key_column data_type,
 column_name_1 data_type, column_name_2 data_type, ...)
 STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler|
io.transwarp.hyperbase.HyperbaseStorageHandler' WITH SERDEPROPERTIES
 ("hbase.columns.mapping" = ":row_key_column, column_family:column_qualifer_1,
 column_family:column_qualifier_2, ...") TBLPROPERTIES ("hbase.table.name" =
 "hbase_table_name")
```

说明

- Inceptor表的第一列必须是对应Hyperbase表中的Row Key。
- Inceptor表中剩余的列将是对应Hyperbase表中的所有column_family:column_qualifier组合。
- `(row_key_column data_type, column_name_1 data_type, column_name_2 data_type, …)` 和 `:row_key_column, column_family:column_qualifer_1, column_family:column_qualifier_2, …` 中的各个字段有一一对应的关系。也就是说`row_key_column`对应`:row_key_column`，`column_name_1`

对应`column_family:column_qualifer_1`, `column_name_2`对应`column_family:column_qualifer_2`，以此类推。

- hbase_table_name是对应的Hyperbase表的表名。
- 通过这种方式在Inceptor中创建的表称为hbase_table_name的*映射表*。映射关系成立后，在Inceptor Shell中对映射表做的改动和在HBase Shell中对Hyperbase表做的改动都会同步体现在映射表和Hyperbase表中。
- 建映射表时，Hyperbase表中不需要有数据。事实上，我们只要在HBase Shell中定义了Hyperbase表，就可以在Inceptor中建它的映射表。在建映射表时，WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":row_key_column, column_family:column_qualifer_1, column_family:column_qualifier_2, …")子句同时也定义了Hyperbase表各列族中的列。
- 上面，我们说建映射表时，Hyperbase表中不需要有数据。但是在使用HyperbaseStorageHandler作为SerDe的情况下（通常，为表建STRUCT_INDEX时会用到这个storage handler），在建映射表时，Hyperbase表中*不能*有数据。原因是在Hbase Shell中通过put指令写入表中的数据编码方式和HyperbaseStorageHandler的编码方式不同。建映射表时，如果Hyperbase表中已经有通过put写入过数据，这些数据的编码便不能和映射表的编码对应，映射表数据的写入就会发生错误。所以，当选择HyperbaseStorageHandler作为storage handler时，要在Hbase Shell中建表后，通过Inceptor Shell向表内填入数据，而不是在HBase Shell中使用put。事实上，Inceptor Shell的表达能力要远远强于HBase Shell，我们建议如果您的Hyperbase表是主要用途在SQL查询，在HBase Shell建表完成后，所有的操作都在Inceptor Shell中完成。在Inceptor Shell中，我们除了不能对映射表进行分区和分桶，其他Inceptor SQL对映射表都适用。此外，在Inceptor中，单条INSERT, UPDATE和DELETE不能对普通二维表使用。但是可以对Hyperbase映射表使用。

举例

下例是使用HBaseStorageHandler作为storage handler建映射表的例子。在“Hyperbase索引”章节会有使用HyperbaseStorageHandler做storage handler的例子。下面，我们为bi表建映射表：

```
transwarp> CREATE EXTERNAL TABLE bank_info (acc_num STRING, name STRING,
 password STRING, email STRING, cellphone STRING, balance DOUBLE)  STORED
 BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH SERDEPROPERTIES
 ("hbase.columns.mapping"=":key, ps:nm, ps:pw, ct:em, ct:cp, bl:bl") TBLPROPERTIES
 ("hbase.table.name"="bi");
```

让我们查看一下bank_info中的内容:

```
transwarp> select * from bank_info;
0001    Zhang San    5678    zs@mail.com     12345678912     10000.0
0002    Li Si        2468    ls@transwarp.io 13513572468     56000.0
0003    Wang Wu      1357    ww@hmail.com    13612345678     500.0
```

和预想的一样，映射表中只有最新信息。 下面，我们可以在Inceptor中对bank_info使用Inceptor SQL查询：

举例

```
transwarp> SELECT * FROM bank_info WHERE acc_num = '0003';
```

```
0003    Wang Wu      1357    ww@hmail.com    13612345678     500.0
```

4.1. 在Inceptor中对映射表进行操作

在Inceptor中，我们可以对一张映射表做所有除了分区和分桶外所有的InceptorSQL操作。InceptorSQL操作强大而易用，我们建议除了建Hyperbase表以外，所有对表的操作在Inceptor Shell中进行。 如果一张表是映射表，我们还能对其执行一些普通Inceptor表不支持的操作，包括单条INSERT, UPDATE和DELETE。

单条INSERT:

语法

```
INSERT INTO table_name (column_name1, column_name2, ...) VALUES (value1, value2, ...)
```

举例

```
transwarp> INSERT INTO bank_info (acc_num, name, password, email, cellphone, balance)
 VALUES ('0004', 'Sun Liu', '2357', 'zl@qmail.com', '13509876543', 200.00);
transwarp> SELECT * FROM bank_info;
0001    Zhang San       6789    zs@mail.com     12345678912     10000.0
0002    Li Si   2468    ls@transwarp.io 13513572468     56000.0
0003    Wang Wu 1357    ww@hmail.com    13612345678     500.0
0004    Sun Liu 2357    zl@qmail.com    13509876543     200.0
```

如果字段有struct类型，需要使用named_struct函数来实现。

举例

假设字段col1为struct类型：struct(col1_c1 : string, col1_c2 : string)， 那么需要进行如下操作：

```
INSERT INTO table_name (col1, col2, col3) VALUES (NAMED_STRUCT('col1_c1', 'value1',
 'col1_c2', 'value2'), 'value3', 'value4');
```

UPDATE

使用UPDATE语句可以在Inceptor Shell里对Hyperbase表进行更新。在Inceptor Shell中对映射表UPDATE有两种形式：

形式一:

语法

```
UPDATE table_name SET column_name = value, column_name = value, ... WHERE
 filter_condition;
```

举例

下例更新用户Sun Liu的密码：

```
transwarp> UPDATE bank_info SET password = '1123' WHERE acc_num = '0004';
transwarp> SELECT * FROM bank_info WHERE acc_num = '0004';
0004    Sun Liu 1123    zl@qmail.com    13509876543     200.0
```

形式二:

语法

```
UPDATE table_name SET (col1, col2, col3, ...) = (SELECT col1, col2, col3 FROM [from
 source]);
```

说明

- 括号中必须是一个完整的查询语句。
- 在SET后面的字段列表及查询语句中需要包含对应的rowkey字段（如col1）

举例

以下是一张有不同信息的银行用户表bank_info2:

```
transwarp> SELECT * FROM bank_info2;
0005    Zhou Qi 5813    zq@lmail.com    13497531246     3000.0
```

我们可以通过以下UPDATE语句将bank_info2中的信息并入bank_info中：

```
transwarp> UPDATE bank_info SET (acc_num, name, password, email, cellphone, balance)
 = (SELECT acc_num, name, password, email, cellphone, balance FROM bank_info2);
transwarp> SELECT * FROM bank_info;
0001    Zhang San       6789    zs@mail.com     12345678912     10000.0
0002    Li Si           2468    ls@transwarp.io 13513572468     56000.0
0003    Wang Wu         1357    ww@hmail.com    13612345678     500.0
0004    Sun Liu         1123    zl@qmail.com    13509876543     200.0
0005    Zhou Qi         5813    zq@lmail.com    13497531246     3000.0
```

DELETE

语法

```
DELETE FROM table_name WHERE filter_condition
```

举例

我们可以用DELETE指令删除上面刚刚并入bank_info中的一条数据：

```
transwarp> DELETE FROM bank_info WHERE acc_num = '0005';
transwarp> SELECT * FROM bank_info;
0001    Zhang San       6789    zs@mail.com     12345678912     10000.0
0002    Li Si   2468    ls@transwarp.io 13513572468     56000.0
0003    Wang Wu 1357    ww@hmail.com    13612345678     500.0
0004    Sun Liu 1123    zl@qmail.com    13509876543     200.0
```

下面我们举一个在Inceptor Shell中的常见查询的例子：

JOIN

从Inceptor Shell我们还可以对两张Hyperbase表进行JOIN操作。我们在Hyperbase中建一张交易表jy，列族为ac，am；列为ac:ac，am:am。Rowkey是交易流水号。ac:ac记录进行交易的账户，am:am记录交易的金额，记录银行用户表账户的收入和支出。正数为收入，负数为支出。

```
hbase(main):023:0> create 'jy', 'ac','am'
hbase(main):026:0> put 'jy', 'a001', 'ac:ac', '0001'
hbase(main):027:0> put 'jy', 'a001', 'am:am', '-2000'
hbase(main):028:0> put 'jy', 'a002', 'ac:ac', '10000'
hbase(main):029:0> put 'jy', 'a002', 'ac:ac', '0002'
hbase(main):030:0> put 'jy', 'a002', 'am:am', '10000'
hbase(main):031:0> put 'jy', 'a003', 'ac:ac', '0001'
hbase(main):032:0> put 'jy', 'a003', 'am:am', '1000'
hbase(main):033:0> scan 'jy'
ROW                                COLUMN+CELL
 a001                              column=ac:ac, timestamp=1423136264981, value=0001
 a001                              column=am:am, timestamp=1423136284893,
 value=-2000
 a002                              column=ac:ac, timestamp=1423136319545, value=0002
 a002                              column=am:am, timestamp=1423136334595,
 value=10000
 a003                              column=ac:ac, timestamp=1423136354122, value=0001
 a003                              column=am:am, timestamp=1423136363431, value=1000
```

然后在Inceptor Shell中建jy的映射表jiaoyi:

```
transwarp> CREATE EXTERNAL TABLE jiaoyi (jy_num STRING, acc_num STRING, amount
 DOUBLE) STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH
 SERDEPROPERTIES ("hbase.columns.mapping"=":key,ac:ac,am:am") TBLPROPERTIES
 ("hbase.table.name"="jy");
transwarp> SELECT * FROM jiaoyi;
a001    0001    -2000.0
a002    0002    10000.0
a003    0001    1000.0
```

现在我们通过对jiaoyi和bank_info的JOIN操作查看各个账户持有人执行过的交易：

```
transwarp> SELECT name, amount FROM bank_info JOIN jiaoyi ON acc_num = acc_num;
Zhang San       -2000.0
Zhang San       1000.0
Li Si           10000.0
```

5. 索引

在数据库中对表生成索引可以加快查询。Hyperbase支持两种索引：

- 组合索引COMBINE_INDEX
- 结构索引STRUCT_INDEX

COMBINE_INDEX使用一列或者多列生成索引，而STRUCT_INDEX对STRUCT类型中的一个字段生成索引。这两种索引分别可以是全局的（global）和局部的（local）。全局的索引与与原表独立,以一张表（ **索引表** ）形式存在。局部的索引就在原表中，以一个新的列（ **索引列** ）的形式存在。

5.1. 索引的生成逻辑

Hyperbase表中的数据以Row Key排序，所以以Row Key为键进行的查询最高效。当我们用某个其他列或者某个struct类型中的一个字段作查询键时，系统需要全表扫描，效率较低。用查询键建索引可以帮助系统提高查询效率。Hyperbase中的索引便是利用了Row Key的有序。假设我们对一张表a中的cf:cq列生成索引。Hyperbase对每行记录都生成一条新记录，这条新记录的Row Key是原记录的cf:cq值与原记录Row Key的合并，新记录本身没有内容。这些新记录就是用cf:cq生成的索引。根据cf:cq值对a表进行查询时，系统会 先根据索引的Row Key进行查询。 索引的所有信息都包含在索引行的Row Key中。所以，在下面的讨论中，除非另外指出，“索引”表示“索引行的Row Key”。

5.2. 索引的定义子句（index definition clause）

索引的定义子句会指定索引类型（COMBINE_INDEX, STRUCT_INDEX, THEMIS_INDEX），索引名称，生成索引的列或STRUCT字段，索引长度和一些可选项。相同类型的全局索引和局部索引使用的索引定义子句相同。

5.2.1. COMBINE_INDEX

COMBINE_INDEX的定义子句为

语法

```
COMBINE_INDEX|INDEXED=cf1:cq1:n1|cf1:cq2:n2|...|rowKey:rowKey:m,[UPDATE=true]
```

说明

- cf1:cq1:b1|cf1:cq2:n2|…部分指定生成索引的列。cf1:cq1:n1表示生成的索引中将有用cf1:cq1列生成的一段长度为n1的字段。如果cf1:cp1值长度不足n1，Hyperbase将用0表示空格将索引补足。如果长度超过n1，超过的部分将在索引的末尾显示。cf1:cq2:n2表示生成的索引中将有用cf1:cq2列生成的一段长度为n2的字段。如果cf1:cp2值长度不足n2，Hyperbase将用0表示空格将索引补足。如果长度超过n2，超过的部分将在索引的末尾显示。以此类推
- 每条生成的索引中都会包含一段原表Row Key生成的长度为m的字段，由rowKey:rowKey:m指定。

- 索引中，在列值和原表Row Key生成的字段之后紧跟几个SHORT型数据用于存储列值和原表Row Key真实的长度。
- 如果用于生成索引的列值比指定的长度长，在索引中无法显示的部分将在这些SHORT型数据后面补足。
- UPDATE=true为可选项，代表在创建索引之前会通过索引列查询数据，如果能查到，就删除前面的索引，重新建索引。
- 对于一条记录：'row1', 'f1:q1', 'abc'，使用…'COMBINE_INDEX|INDEXED=f1:q1:5|rowKey:rowKey:8' 生成的索引行为: 'abc00row100000304', 'f:q', null

其中，'abc00row100000304' 中的最后四位是两个SHORT型，分别记录了' abc' 和' row1' 的长度。'f:q' 为Hyperbase为索引表自动生成的列名。null代表单元格中没有任何数据。

- 我们也可以用多列组合生成索引。对于记录：

'row1', 'f1:q1', 'value1', 'row2', 'f1:q2', 'value2'

使用 …'COMBINE_INDEX|INDEXED=f1:q1:4|f1:q2:7|rowKey:rowKey:8' 生成的索引行如下：

'valuvalue200406e1', 'f:q', null

5.2.2. STRUCT_INDEX

STRUCT是一种复杂数据类型，包含一系列命名字段。这些字段可以各自拥有不同的数据类型。举个例子，我们定义一个地址（ad）STRUCT，包含街址（st）、楼号（bd）、房间号（rm）三个字段：

```
ad STRUCT &lt;st:string, bd:string, rm:string&gt;
```

一个ad STRUCT值可以是这样的：

```
STRUCT('481 Guiping Road','18','401')
```

一张表中可以有struct类型的列。我们可以使用一个struct类型中的一个字段为键来建索引。STRUCT_INDEX的定义子句为：

语法

```
STRUCT_INDEX|INDEXED=cf:cq,TYPE=field_data_type:field_data_type:...,LOCATION=n,
[DCOP=true]
```

说明

- cf:cq指定用于建索引的STRUCT类型所在的列。
- TYPE=field_data_type:field_data_type:…告诉系统该STRUCT类型中各个字段的数据类型。需要注意的是，Hyperbase中的STRING类型分为VALUE_STRING和KEY_STRING，KEY_STRING用于Row Key中的STRING类型，VALUE_STRING用于其他列族里的STRING类型。举个例子，一张Hyperbase表中，如果上面提到的地址STRUCT类型在Row Key

中，建STRUCT_INDEX时字段类型的指定将是：TYPE=KEY_STRING:KEY_STRING:KEY_STRING。但是，如果地址STRUCT出现在其他列族中，建STRUCT_INDEX时字段类型的指定将是：TYPE=VALUE_STRING:VALUE_STRING:VALUE_STRING。

- LOCATION=n指定索引用到了STRUCT类型中的哪一个字段。STRUCT类型中的字段从0开始编号。所以如果我们要用上面提到的地址STRUCT中的街址字段做STRUCT_INDEX，那么这里我们需要填写LOCATION=0。
- DCOP=true 为可选项，说明当一个列同时插入多条数据时，会进行过滤，只选择一条最新的数据插入，其他数据丢弃，根据Hbase里面的KVCompare进行排序。
- STRUCT_INDEX比较复杂。首先，我们回忆一下，Hyperbase表以bype[]形式存储数据，也就是说Hyperbase表中的数据没有数据类型。但是为了生成STRUCT_INDEX，系统必须要能够分辨STRUCT的字段及其类型。Hyperbase做不到这一点，我们必须通过Hyperbase表在Inceptor中的映射表来控制。所以要对一张Hyperbase表生成STRUCT_INDEX，这张表必须在Inceptor中有映射表而且映射表的storage handler必须为HyperbaseStorageHandler（不同于普通映射表的HBaseStorageHandler。 **注意**，因为HyperbaseStorageHandler的对数据的编码方式和HBase Shell中put指令的编码方式不一样，表中的所有数据必须从Inceptor Shell插入，而*不能*从Hyperbase插入，否则会导致映射表数据写入出错。

5.3. 索引的创建操作

下面我们介绍具体如何创建各种类型的全局索引（索引表）和局部索引（索引列）。**注意**，创建索引表和索引列并不生成索引信息。生成索引的方法在下一节。

5.3.1. 全局索引

语法：add_index

```
add_index 'table_name', 'index_name', 'index_definition_clause'
```

说明 该语句为table_name表添加全局索引，生成一张名为’table_name_index_name’的索引表。在index_definition_clause处使用对应索引类型的索引定义子句。比如下面语法创建全局索引：

```
add_index 'table_name', 'index_name', 'COMBINE_INDEX|INDEXED=cf1:cq1:n1|
cf1:cq2:n2|...|rowKey:rowKey:m,[UPDATE=true]'
```

注意，这里的索引表表名将是“原表表名_索引名”。也就是说，如果原表表名为a，索引名为b，那么索引表表名为a_b。**注意区分索引表表名和索引名**。

举例：单列生成的全局COMBINE_INDEX 为bi表以ps:nm为键创建名为index的索引。生成的索引表表名为bi_index。

```
hbase(main):025:0> add_index 'bi','index','COMBINE_INDEX|INDEXED=ps:nm:8|
rowKey:rowKey:10'
```

举例：多列生成的全局COMBINE_INDEX 为bi表以ps:nm和ps:pw为键创建名为index_2的组合索引。生成的索引表表名为bi_index2。

```
hbase(main):026:0> add_index 'bi','index2','COMBINE_INDEX|INDEXED=ps:nm:8|ps:pw:8|
rowKey:rowKey:10,UPDATE=true'
```

举例：全局STRUCT_INDEX 我们用银行用户的信息做例子。表中现在有用户的地址信息，具有地址STRUCT类型来存储。 Zhang San，Li Si，Wang Wu的住址分别如下：

```
Zhang San: {"street":"481 Guiping Road", "building":"21", "room":"401"}
Li Si:     {"street":"481 Guiping Road", "building":"18", "room":"301"}
Wang Wu: {"street":"300 Guiling Road", "building":"01", "room":"203"}
```

现在以地址中的street字段为表bi建STRUCT_INDEX，索引名为si，因为street字段是STRUCT中的第一个字段，location为0。注意，因为这里的地址不是Row Key，所以地址中各个字段的数据类型为VALUE_STRING（具体解释见上文“索引定义子句”中的STRUCT_INDEX部分）：

```
hbase(main):020:0> add_index 'bi','si','STRUCT_INDEX|
INDEXED=ct:ad,TYPE=VALUE_STRING:VALUE_STRING:VALUE_STRING,LOCATION=0'
```

5.3.2. 局部索引

局部索引是表中的一个列族，可以在建表时创建或者在建表后通过alter添加。创建时，要将列族参数中的LOCAL_INDEX设为对应索引的定义子句。

语法：建表时建索引

```
create 'table_name', 'column_family_name', ..., {NAME => 'local_index_name',
 LOCAL_INDEX=>'index_definition_clause'}
```

语法：建表后建索引

```
hbase(main):011:0>alter 'table_name', {NAME=>'local_index_name',
 LOCAL_INDEX=>'index_definition_clause'}
```

举例：建表时建COMBINE_INDEX

在建bi表时用ps:nm列建名为ci的局部索引：

```
hbase(main):036:0> create 'bi','ps','ct','bl',{NAME => 'ci',
 LOCAL_INDEX=>'COMBINE_INDEX|INDEXED=ps:nm:8|rowKey:rowKey:10'}
```

举例：为表添加COMBINE_INDEX

为bi表添加用ct:cp列建的名为li的局部索引：

```
hbase(main):018:0>alter 'bi', {NAME => 'li', LOCAL_INDEX=>'COMBINE_INDEX|
INDEXED=ct:cp:10|rowKey:rowKey:10'}
```

举例：建表时建STRUCT_INDEX

```
hbase(main):004:0> create 'bi', 'ps','ct','bl', {NAME => 'si',
 LOCAL_INDEX=>'STRUCT_INDEX|
INDEXED=ct:add,TYPE=VALUE_STRING:VALUE_STRING:VALUE_STRING,LOCATION=0'}
```

举例：为表添加STRUCT_INDEX

```
hbase(main):023:0> alter 'bi', {NAME => 'si', LOCAL_INDEX=>'STRUCT_INDEX|
INDEXED=ct:add,TYPE=VALUE_STRING:VALUE_STRING:VALUE_STRING,LOCATION=0'}
```

5.4. 索引的生成操作

索引创建完成后，索引表（列）中暂时没有数据。索引（表）列中的数据会在两个时候生成：

1. 原表有新数据插入时，系统会自动为新数据生成索引
2. 对全局索引执行rebuild_index，对局部索引执行rebuild_local_index，系统会起一个MR任务，为原表中已有的数据生成索引，使得索引表（列）中的信息和原表同步。

下面我们介绍如何在第二种情况下生成各类索引。

5.4.1. 生成全局索引

语法：rebuild_index

```
rebuild_index 'table_name','index_name'
```

举例：同步全局COMBINE_INDEX

下例我们同步之前创建的bi表的索引index：

```
hbase(main):027:0> rebuild_index 'bi','index'
```

系统会返回如下信息，说明同步成功：

```
15/03/03 22:57:52 INFO client.RMProxy: Connecting to ResourceManager at tw-
node2125/172.16.2.125:8032
15/03/03 22:57:52 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing not
 performed. Implement the Tool interface and execute your application with ToolRunner
 to remedy this.
15/03/03 22:57:54 INFO mapreduce.JobSubmitter: number of splits:1
15/03/03 22:57:54 INFO mapreduce.JobSubmitter: Submitting tokens for job:
 job_1424888448910_0003
15/03/03 22:57:54 INFO impl.YarnClientImpl: Submitted application
 application_1424888448910_0003
15/03/03 22:57:54 INFO mapreduce.Job: The url to track the job: http://tw-
node2125:8088/proxy/application_1424888448910_0003/
15/03/03 22:57:54 INFO mapreduce.Job: Running job: job_1424888448910_0003
15/03/03 22:58:11 INFO mapreduce.Job: Job job_1424888448910_0003 running in uber
 mode : false
15/03/03 22:58:11 INFO mapreduce.Job:  map 0% reduce 0%
15/03/03 22:58:23 INFO mapreduce.Job:  map 100% reduce 0%
15/03/03 22:58:48 INFO mapreduce.Job:  map 100% reduce 89%
```

```
15/03/03 22:58:52 INFO mapreduce.Job:  map 100% reduce 100%
15/03/03 22:58:53 INFO mapreduce.Job: Job job_1424888448910_0003 completed
 successfully
15/03/03 22:58:53 INFO mapreduce.Job: Counters: 59
        File System Counters
                FILE: Number of bytes read=255
                FILE: Number of bytes written=270637
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=65
                HDFS: Number of bytes written=1198
                HDFS: Number of read operations=7
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=3
        Job Counters
                Launched map tasks=1
                Launched reduce tasks=1
                Rack-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=9615
                Total time spent by all reduces in occupied slots (ms)=26780
                Total time spent by all map tasks (ms)=9615
                Total time spent by all reduce tasks (ms)=26780
                Total vcore-seconds taken by all map tasks=9615
                Total vcore-seconds taken by all reduce tasks=26780
                Total megabyte-seconds taken by all map tasks=9845760
                Total megabyte-seconds taken by all reduce tasks=27422720
        Map-Reduce Framework
                Map input records=3
                Map output records=3
                Map output bytes=243
                Map output materialized bytes=255
                Input split bytes=65
                Combine input records=0
                Combine output records=0
                Reduce input groups=3
                Reduce shuffle bytes=255
                Reduce input records=3
                Reduce output records=3
                Spilled Records=6
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=483
                CPU time spent (ms)=13740
                Physical memory (bytes) snapshot=454623232
                Virtual memory (bytes) snapshot=5797310464
                Total committed heap usage (bytes)=354488320
        HBase Counters
                BYTES_IN_REMOTE_RESULTS=538
                BYTES_IN_RESULTS=538
                MILLIS_BETWEEN_NEXTS=665
                NOT_SERVING_REGION_EXCEPTION=0
                NUM_SCANNER_RESTARTS=0
                REGIONS_SCANNED=1
                REMOTE_RPC_CALLS=3
                REMOTE_RPC_RETRIES=0
                RPC_CALLS=3
```

```
        RPC_RETRIES=0
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=0
    File Output Format Counters
        Bytes Written=1198
```

我们可以通过在Hbase Shell中scan索引表来查看刚刚生成的索引内容。注意，scan后跟索引表 **表名**，而不是索引名。

```
hbase(main):004:0> scan 'bi_index'
ROW                                                          COLUMN+CELL
 Li Si\x00\x00\x000002\x00\x00\x00\x00\x00\x00\x00\x05\x00\x column=f:q,
 timestamp=1422994031643, value=
 04
 Wang Wu\x000003\x00\x00\x00\x00\x00\x00\x00\x07\x00\x04     column=f:q,
 timestamp=1422994032031, value=
 Zhang Sa0001\x00\x00\x00\x00\x00\x00\x00\x09n\x00\x04       column=f:q,
 timestamp=1425396627950, value=
```

下面我们同步之前用ps:nm和ps:pw创建的组合索引index2：

```
hbase(main):005:0> rebuild_index 'bi','index2'
```

查看组合索引表bi_index2:

```
hbase(main):006:0> scan 'bi_index2'
ROW                                                          COLUMN+CELL
 Li Si\x00\x00\x002468\x00\x00\x00\x000002\x00\x00\x00\x00\x column=f:q,
 timestamp=1422994031756, value=
 00\x00\x00\x05\x00\x04\x00\x04
 Wang Wu\x001357\x00\x00\x00\x000003\x00\x00\x00\x00\x00\x00 column=f:q,
 timestamp=1422994032103, value=
 \x00\x07\x00\x04\x00\x04
 Zhang Sa5678\x00\x00\x00\x000001\x00\x00\x00\x00\x00\x00\x0 column=f:q,
 timestamp=1425396652672, value=
 0\x09n\x00\x04\x00\x04
```

举例：同步全局STRUCT_INDEX

下面我们同步上文以地址STRUCT中的street字段生成的全局STRUCT_INDEX：

```
hbase(main):021:0> rebuild_index 'bi','si'
```

索引同步成功，我们可以查看一下索引表里的数据：

```
hbase(main):022:0> scan 'bi_si'
```

```
ROW                                          COLUMN+CELL
 \x00\x00\x00\x10300 Guiling Road0003\x00\x column=f:q, timestamp=1425471440113,
 value=
 FF
 \x00\x00\x00\x10481 Guiping Road0001\x00\x column=f:q, timestamp=1425470399756,
 value=
 FF
 \x00\x00\x00\x10481 Guiping Road0002\x00\x column=f:q, timestamp=1425471272718,
 value=
 FF
```

语法：rebuild_local_index

```
rebuild_local_index
```

举例：同步局部COMBINE_INDEX

上文中，我们曾经为bi表添加过一个用ct:cp列建的、名为li的局部索引。现在我们同步这个索引：

```
hbase(main):048:0> rebuild_local_index 'bi', 'li'
```

查看一下刚刚建好的索引：

```
hbase(main):057:0> scan_local_index 'bi'
ROW                                                    COLUMN+CELL
 12345678910001\x00\x00\x00\x00\x00\x00\x00\x0B2\x00\x04     column=li:q,
 timestamp=1426692335411, value=
 13513572460002\x00\x00\x00\x00\x00\x00\x00\x0B8\x00\x04     column=li:q,
 timestamp=1422994031890, value=
 13612345670003\x00\x00\x00\x00\x00\x00\x00\x0B8\x00\x04     column=li:q,
 timestamp=1422994032237, value=
```

5.5. 索引的删除操作

5.5.1. 全局索引的删除

全局索引虽然是一张表，但是不能像删除普通表一样先disable然后drop。要用专门的delete_index命令：

语法：delete_index

```
delete_index 'table_name','index_name'
```

5.5.2. 局部索引的删除

局部索引是表的一个列族。删除局部索引和删除普通列族用同样的指令：

```
alter 'table_name','delete'=>'index_name'
```

6. 事务处理

事务处理（transaction）是数据库保证原子性（atomicity）的方法。原子性是指一系列任务在系统中只会有完成和未完成两种状态——不会有只完成了一半的情况。事务处理的任务都是对表本身有修改的语句，包括增删改，也就是SQL中的DML语句：LOAD/INSERT/UPDATE/DELETE （见本指南的“在Inceptor Shell中的操作”章节和《Inceptor 使用手册》的DML章节）。TDH支持通过Inceptor Shell和JDBC进行事务处理。

6.1. Inceptor Shell中的操作

在Inceptor Shell中进行事务操作时，必须在local mode下。进入local mode模式运行以下指令：

```
set ngmr.exec.mode=local
```

通过Inceptor Shell进行事务处理的对象必须是*事务表*。事务表存储在Hyperbase中，Inceptor对它的映射表进行事务处理。和普通映射表不同，事务表的映射表是*Inceptor的托管表*。下面我们先介绍如何建表。 ==== 建表 建事务表的映射表时，不能在Hyperbase中建表，必须直接在Inceptor中建表并使用HBaseTransactionStorageHandler作为Storage Handler。建表命令执行时，系统会自动在Hyperbase中建立对应的事务表。

语法

```
CREATE TABLE table_name (column_name data_type, column_name data_type, ...) STORED
 BY 'org.apache.hadoop.hive.hbase.transaction.HBaseTransactionStorageHandler'
 TBLPROPERTIES ('hbase.table.name'='hyperbase_table_name')
```

说明

- TBLPROPERTIES子句中的’hbase.table.name’=’hyperbase_table_name’为系统指定在Hyperbase中建的表的名称。
- 建托管映射表时，不需要定义映射表的列和Hyperbase表的列之间的对应。也就是说，这里不需要 WITH SERDEPROPERTIES (’hbase.columns.mapping’=’:key, cf:cq, …’) 这一子句。系统会根据该映射表的列名自动生成Hyperbase表中的列族。

举例

下面，我们建一张只包含了账户号码和存款金额的事务表balance：

```
transwarp> CREATE TABLE balance (acc_num STRING, amount DOUBLE) STORED BY
 'org.apache.hadoop.hive.hbase.transaction.HBaseTransactionStorageHandler'
 TBLPROPERTIES ('hbase.table.name' = 'bl');
```

关于插入数据：事务表中的数据必须从Inceptor插入表，不可以通过HBase Shell插数据。

6.1.1. 事务表的索引

事务表支持全局的COMINE_INDEX和STRUCT_INDEX，不支持局部索引。事务表索引的使用方式和普通全局索引的使用方式完全相同，只是需要不同的创建（**add_transaction_index**）和删除（**delete_transaction_index**）指令。注意，虽然事务表指令大多在Inceptor Shell中完成，索引还是需要在HBase Shell中创建。

语法：add_transaction_index

```
add_transaction_index, 'table_name','index_name','index_definition_clause'
```

语法：delete_transaction_index

```
delete_transaction_index, 'table_name','index_name'
```

Hyperbase暂不支持将索引表和原表同步的功能(全局索引的rebuild_index功能），索引只能在表中插入数据时生成。所以，*注意*一定要在事务表为空表时创建事务表索引，否则表中原有数据将没有索引。

6.1.2. 事务处理

Inceptor SQL的事务处理指令为COMMIT TRANSACTION （提交事务）或者ROLLBACK TRANSACTION（回滚事务，撤回事务）。在Inceptor中对事务表的映射处理时，系统会默认任何active session都包含在一个事务中，所以无需像在一些其他数据库中专门通过BEGIN TRANSACTION开始事务。在Inceptor中COMMIT TRANSACTION或者ROLLBACK TRANSACTION后，一个事务即结束，系统会默认接 着自动开始另一个事务。概括来说，Inceptor中事务处理的代码将如下：

语法

```
DML STATEMENTS;
DML STATEMENTS;
.
.
.
COMMIT/ROLLBACK ;
DML STATEMENTS;
```

下面，我们通过两个例子来解释如何进行事务处理。

举例：单表事务处理 下面，我们举一个只有一张表参与事务处理的例子。假设上面建的balance表中现在有如下数据：

```
transwarp> SELECT * FROM balance;
0001    10000.0
0002    56000.0
0003    500.0
```

现在，账户号码为0001的用户要给账户号码为0003的用户转账500元：

```
transwarp> UPDATE balance SET amount = 10000.0-500 WHERE acc_num = '0001';
transwarp> UPDATE balance SET amount = 500.0+500 WHERE acc_num = '0003';
transwarp> SELECT * FROM balance;
0001    10000.0
0002    56000.0
0003    500.0
transwarp> COMMIT;
transwarp> SELECT * FROM balance;
0001    9500.0
0002    56000.0
0003    1000.0
```

我们可以看到，两次UPDATE之后查看表，表的内容没有变化，只有COMMIT TRANSACTION之后，变化才产生。

举例：多表事务处理

下面，我们看一个有两张表参与事务处理的例子。我们介绍第二张事务表identification，包含了账户号码，用户的名字和身份证号码：

```
transwarp> SELECT * FROM identification;
0001    Zhang San   320102199107020038
0002    Li Si       320102198910310001
0003    Wang Wu     320102199709161003
```

现在，银行有一个新用户Zhao Liu开户，并存入5000.0元。银行需要同时更新两张表中的信息：

```
transwarp> INSERT INTO identification (acc_num, name, in) VALUES ('0004','Zhao Liu',
 '320102196308030038');
transwarp> INSERT INTO balance (acc_num, balance) VALUES ('0004','5000.0);
transwarp> COMMIT;
```

6.1.3. 通过JDBC进行事务处理

下面我们提供一个通过JDBC进行事务处理的例子。

举例

```
import java.sql.*;

/**
 * Created by root on 1/22/15.
 */
public class HiveTransactionJDBC {
    private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";
    public static void main(String[] args) throws SQLException {
        try {
            Class.forName(driverName);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }
```

```
        {
            Connection connection =
DriverManager.getConnection("jdbc:transwarp://172.16.1.51:10000/default", "", "");
            Statement stmt = connection.createStatement();

            //建表
            stmt.execute("drop table if exists Account");
            String create =
                    "create table Account ( "
                            + "carno string, ID string, money double "
                            + ") "
                            + "row format delimited collection items terminated by
'|' "
                            + "stored by
'org.apache.hadoop.hive.hbase.transaction.HBaseTransactionStorageHandler' "
                            + "with
serdeproperties('hbase.columns.mapping'=':key,f:ID,f:money') "
                            + "tblproperties('hbase.table.name'='Account')";
            //设为local mode，事务处理必须在local mode中进行
            stmt.execute(create);

            connection.setAutoCommit(false);
            stmt.execute("set ngmr.exec.mode=local");
            //进行三次insert
            stmt.executeUpdate("insert into Account(carno, ID, money)
values('carno0001', 'ID0001', 10001)");
            stmt.executeUpdate("insert into Account(carno, ID, money)
values('carno0002', 'ID0002', 10002)");
            stmt.executeUpdate("insert into Account(carno, ID, money)
values('carno0003', 'ID0003', 10003)");
            //提交这三次insert
            connection.commit();

            //stmt.execute("begin transaction");
            ResultSet rs1 = stmt.executeQuery("select count(*) from Account");
            while(rs1.next()) {
                //因为刚才的三次insert提交了，所以查询表内数据条数应该得到3
                System.out.println("Should be 3 : " + rs1.getInt(1));
            }

            //再进行两次insert
            stmt.executeUpdate("insert into Account(carno, ID, money)
values('carno0004', 'ID0004', 10004)");
            stmt.executeUpdate("insert into Account(carno, ID, money)
values('carno0005', 'ID0005', 10005)");
            //回归刚才的两次insert
            connection.rollback();

            ResultSet rs2 = stmt.executeQuery("select count(*) from Account");
            while(rs2.next()) {
                //因为刚才的两次insert被回滚，查询表内数据条数应该还是3
                System.out.println("Should also be 3 : " + rs2.getInt(1));
            }
        }
    }
}
```

7. Hbase Object Store使用案例

7.1. 应用背景

在HBase中存储图片。

7.2. HBase表设计

极简化设计，如下：

对象	对象名
表名	picture
列族1	pic
列族2	lobp

lobp是pic的索引列。

7.3. 配置

需要考虑重新配置的参数如下：

1. 对于列族pic来说，有必要重新配置其flush size，因为若flush size较小，图片较大，flush频率会很高。
2. TTL需要设置为一个合理的值，毕竟存储空间有限，较为久远的图片数据可以舍弃。

7.4. 创建表

在创建表的时候可以根据实际情况考虑进行region的pre-splitting。

创建表picture有两种方式：HBase Shell创建；Java API创建。关键在于如何使pic列族存储lob数据。

7.4.1. Hbase Shell 创建

Shell创建的关键在于编写json文件。

表pic的json文件“picture”内容如下（文件路径为/tmp/picture）：

```
{
  "tableName" : "picture",
  "base" : {
    "SPLIT_KEYS" : [ ],
    "families" : [ {
      "FAMILY" : "pic",
      "DATA_BLOCK_ENCODING" : "PREFIX",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : 0,
```

```
      "VERSIONS" : 1,
      "COMPRESSION" : "SNAPPY",
      "MIN_VERSIONS" : 0,
      "TTL" : 64800,
      "BLOCKSIZE" : 65536,
      "IN_MEMORY" : false,
      "BLOCKCACHE" : true
    }]
  },
  "fulltextindex" : {
  },
  "globalindex" : {
    "indexs" : [ ]
  },
  "lob" : {
    "indexs" : [ {
      "INDEX_NAME" : "LOBP",
      "UPDATE" : false,
      "DCOP" : true,
      "INDEX_CLASS" : "org.apache.hadoop.hbase.secondaryindex.indexbuilder.LOBIndex",
      "indexColumnInfos" : [ {
        "FAMILY" : "pic",
        "QUALIFY" : "",
        "SEGMENT_LENGTH" : -1,
        "LOCATION" : -1
      } ]
    } ]
  },
  "localindex" : {
    "indexs" : [ ]
  },
  "transaction" : {
    "THEMIS_ENABLE" : false
  }
}
```

如果实际情况下还有别的列族，请修改json文件。上述文件中有设置TTL，也可根据实际情况修改。

在HBase Shell下，执行如下操作：

```
hbase(main):001:0>  alterUseJson 'picture','/tmp/picture'
hbase(main):002:0>  alter 'picture',{NAME=>'pic',METADATA=>{'flushsize'=>'67108864'}}
  //修改列族pic的flushsize
hbase(main):003:0> desc 'picture'      //查看表信息
```

查看表picture的详细信息，确定正确创建。

7.4.2. Java API 创建

Java API创建代码如下

涉及到的一些变量：

```
public final static String PIC="pic";
```

```
public final static String LOB_INDEX="lobp";

public final static int flushSize = 1024 * 1024 * 256; // 256M
public final static int TTL = 24 * 60 * 60; // one day
```

创建方法：

```
public void createTable(String tableName, String startkey, String endkey, int
 numOfRegions) {
 Configuration conf = HBaseConfiguration.create();
 try {
  HBaseAdmin admin = new HBaseAdmin(conf);

  if (admin.tableExists(tableName)) {
   admin.disableTable(tableName);
   admin.deleteTable(tableName);
   System.out.println(tableName + " exists, delete it ...");
  }

  HTableDescriptor table = new HTableDescriptor(TableName.valueOf(tableName));
  HColumnDescriptor pic = new HColumnDescriptor(PIC);
 pic.setValue(HConstants.HREGION_MEMSTORE_SPECIAL_COLUMN_FAMILY_FLUSHSIZE_KEY,
 flushSize + "");
  pic.setTimeToLive(TTL);
  table.addFamily(pic);

  admin.createTable(table, Bytes.toBytes(startkey), Bytes.toBytes(endkey),
 numOfRegions);   //picture表预先分为numOfRegions个regions
  admin.close();

  addLobIndex(conf, tableName);  //为pic列族添加索引
  System.out.println(tableName + " is created...");
 } catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
 }
}

private void addLobIndex(Configuration conf, String tableName) {
 GlobalIndexAdmin gAdmin;
 try {
  gAdmin = new GlobalIndexAdmin(conf);
  gAdmin.addLOBForExistFamily(Bytes.toBytes(tableName), Bytes.toBytes(PIC),
 Bytes.toBytes(LOB_INDEX));
  gAdmin.close();
 } catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
 }
}
```

8. Hyperbase Json

8.1. 说明

Inceptor支持多种数据类型，如string、double等，但是Inceptor目前并不支持json类型，所以将json结构指定为binary，如此，就能通过Inceptor创建存储json结构的Hyperbase表，并通过相关的几个函数对表进行增、改、查等操作。

8.2. 相关函数

1. encode_json() ： 对json进行编码，以将其插入Hyperbase表；
2. extract_json() ： 查询时指定读取json中哪个属性；
3. update_json() ： 修改json某一或某些属性值.

8.3. 构建索引

对于json，可以为其中的数据建立索引，语法如下：

```
"globalindex" : {
  "indexs" : [ {
  "INDEX_NAME" : "id_index",
  "SPLIT_KEYS" : [ ],
  "UPDATE" : false,
  "DCOP" : true,
  "INDEX_CLASS" : "org.apache.hadoop.hbase.secondaryindex.indexbuilder.CombineIndex",
  "indexColumnInfos" : [ {
  "FAMILY" : "f",
  "QUALIFY" : "q1",
  "SEGMENT_LENGTH" : 20,
  "LOCATION" : -1,
  "BSONPATH" : "_id.id" // optional
  }, {
  "FAMILY" : "rowKey",
  "QUALIFY" : "rowKey",
  "SEGMENT_LENGTH" : 20,
  "LOCATION" : -1
} ]
} ]
}
```

其中，如下部分是必须包含，且不必修改的：

```
{
  "FAMILY" : "rowKey",
  "QUALIFY" : "rowKey",
  "SEGMENT_LENGTH" : 20,
  "LOCATION" : -1
}
```

8.4. 使用示例

8.4.1. 创建表stuff_infor:

```
drop table if exists stuff_infor;
create table stuff_infor (
  id string,
  info binary
)
stored by 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
tblproperties ("hbase.table.name" = "stuff_infor");
```

其中，info存储json。

8.4.2. 创建索引

如果要为stuff_infor的info创建索引，首先生成相应的json文件"addIndex"，内容如下：

```
{
  "tableName" : "stuff_infor",
  "base" : {
    "SPLIT_KEYS" : [ ],
    "families" : [ {
      "FAMILY" : "f",
      "DATA_BLOCK_ENCODING" : "PREFIX",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : 0,
      "VERSIONS" : 1,
      "COMPRESSION" : "SNAPPY",
      "MIN_VERSIONS" : 0,
      "TTL" : 64800,
      "BLOCKSIZE" : 65536,
      "IN_MEMORY" : false,
      "BLOCKCACHE" : true
    } ]
  },
  "fulltextindex" : {
  },
  "globalindex" : {
    "indexs" : [ {
        "INDEX_NAME" : "name_index",
        "SPLIT_KEYS" : [ ],
        "UPDATE" : false,
        "DCOP" : true,
        "INDEX_CLASS" :
 "org.apache.hadoop.hbase.secondaryindex.indexbuilder.CombineIndex",
        "indexColumnInfos" : [ {
            "FAMILY" : "f",
            "QUALIFY" : "q1",
            "SEGMENT_LENGTH" : 20,
            "LOCATION" : -1,
            "BSONPATH" : "name"
        }, {
            "FAMILY" : "rowKey",
```

```
            "QUALIFY" : "rowKey",
            "SEGMENT_LENGTH" : 20,
            "LOCATION" : -1
          }]
      }, {
        "INDEX_NAME" : "address_index",
        "SPLIT_KEYS" : [ ],
        "UPDATE" : false,
        "DCOP" : true,
        "INDEX_CLASS" :
  "org.apache.hadoop.hbase.secondaryindex.indexbuilder.CombineIndex",
        "indexColumnInfos" : [ {
            "FAMILY" : "f",
            "QUALIFY" : "q1",
            "SEGMENT_LENGTH" : 20,
            "LOCATION" : -1,
            "BSONPATH" : "address"
        }, {
            "FAMILY" : "rowKey",
            "QUALIFY" : "rowKey",
            "SEGMENT_LENGTH" : 20,
            "LOCATION" : -1
         } ]
    } ]
  },
  "lob" : {
    "indexs" : [ ]
  },
  "localindex" : {
    "indexs" : [ ]
  },
  "transaction" : {
    "THEMIS_ENABLE" : false
  }
}
```

切记，要根据实际情况修改一些值，如TTL等；另外，注意globalindex中"BSONPATH"的值为json结构中的某一属性，如"name_index"的"BSONPATH"为"name"。上述创建两个索引表，也可根据实际情况创建组合索引。

将文件拷贝到/tmp下，在hbase shell下执行如下操作：

```
hbase(main):001:0> alterUseJson 'stuff_infor','/tmp/addIndex'
```

生成索引表"stuff_infor_name_index"和"stuff_infor_address_index".

注意，如果在创建索引的时候表中已经有了一些历史数据，那么就需要执行如下命令为历史数据生成索引信息：

```
hbase(main):006:0> rebuild_index 'stuff_infor','name_index'  //'name_index'是addIndex中的"INDEX_NAME"
hbase(main):006:0> rebuild_index 'stuff_infor','address_index'
```

如果不执行rebuild_index，在查询时如果使用json中信息作为查询条件，则查询结果中无符合条件的历史数据。

8.4.3. 插入stuff_infor

```
insert into stuff_infor(id, info) values ('1', encode_json("{\"name\":\"zhangsan\",
\"address\":\"street1\"}"));
insert into stuff_infor(id, info) values ('2', encode_json("{\"name\":\"lisi\",
\"address\":\"street1\"}"));
insert into stuff_infor(id, info) values ('3', encode_json("{\"name\":\"wangwu\",
\"address\":\"street2\"}"));
insert into stuff_infor(id, info) values ('4', encode_json("{\"name\":\"maliu\",
\"address\":\"street1\"}"));
```

8.4.4. 查询stuff_infor

```
select id, extract_json(info, "name") from stuff_infor;
select id, extract_json(info, "name"), extract_json(info, "address") from stuff_infor
 where extract_json(info, "name") = "lisi"; //如果已经为stuff_infor创建索引，会使用索引，否
则，会全表扫描
select id, extract_json(info, "name"), extract_json(info, "address") from stuff_infor
 where extract_json(info, "address") = "street1";
```

8.4.5. 更新stuff_infor

```
update stuff_infor set info=update_json(info, "{$set:{\"address\":\"street2\"}}")
 where extract_json(info, "name")="zhangsan";
```

目前，update只支持$set语法。

附录 A. 程序中调用Hyperbase所需的jar

A.1. 建表所需的jar

- hbase-client-0.98.6-transwarp.jar
- hbase-bson-0.98.6-transwarp.jar
- hbase-common-0.98.6-transwarp.jar
- hbase-protocol-0.98.6-transwarp.jar
- hadoop-common-2.5.2-transwarp.jar
- hadoop-auth-2.5.2-transwarp.jar
- hadoop-annotations-2.5.2-transwarp.jar
- hadoop-mapreduce-client-core-2.5.2-transwarp.jar
- commons-codec-1.7.jar
- commons-io-2.4.jar
- commons-lang-2.6.jar
- commons-logging-1.1.1.jar
- commons-collections-3.2.1.jar
- guava-14.0.1.jar
- protobuf-java-2.5.0.jar
- netty-3.6.2.Final.jar
- htrace-core-2.04.jar
- elasticsearch-1.3.1-transwarp.jar
- jackson-mapper-asl-1.8.8.jar
- findbugs-annotations-1.3.9-1.jar
- zookeeper-3.4.5-transwarp.jar
- log4j-1.2.17.jar
- commons-configuration-1.6.jar
- slf4j-api-1.6.4.jar
- slf4j-log4j12-1.6.4.jar
- lib/jsch-0.1.50.jar
- lib/jzlib-1.1.2.jar

您可以在您集群中的任意一台服务器上运行下面的脚本：

```
#!/bin/bash
```

```
rm -rf /tmp/hbase-jars
mkdir /tmp/hbase-jars
cd /tmp/hbase-jars

jardir=`pwd`

#echo $jarsource":/usr/lib/hbase/lib/hbase-client-0.98.6-transwarp.jar"

cp /usr/lib/hbase/lib/hbase-client-0.98.6-transwarp.jar .
cp /usr/lib/hbase/lib/hbase-bson-0.98.6-transwarp.jar .
cp /usr/lib/hbase/lib/hbase-common-0.98.6-transwarp.jar .
cp /usr/lib/hbase/lib/hbase-protocol-0.98.6-transwarp.jar .
cp /usr/lib/hadoop/hadoop-common-2.5.2-transwarp.jar .
cp /usr/lib/hadoop/hadoop-auth-2.5.2-transwarp.jar .
cp /usr/lib/hadoop/hadoop-annotations-2.5.2-transwarp.jar .
cp /usr/lib/hadoop-mapreduce/hadoop-mapreduce-client-core-2.5.2-transwarp.jar .
cp /usr/lib/hbase/lib/commons-codec-1.7.jar .
cp /usr/lib/hbase/lib/commons-io-2.4.jar .
cp /usr/lib/hbase/lib/commons-lang-2.6.jar .
cp /usr/lib/hbase/lib/commons-logging-1.1.1.jar .
cp /usr/lib/hbase/lib/commons-collections-3.2.1.jar .
cp /usr/lib/hbase/lib/guava-14.0.1.jar .
cp /usr/lib/hbase/lib/protobuf-java-2.5.0.jar .
cp /usr/lib/hbase/lib/netty-3.6.2.Final.jar .
cp /usr/lib/hbase/lib/htrace-core-2.04.jar .
cp /usr/lib/hbase/lib/elasticsearch-1.3.1-transwarp.jar .
cp /usr/lib/hbase/lib/jackson-mapper-asl-1.8.8.jar .
cp /usr/lib/hbase/lib/findbugs-annotations-1.3.9-1.jar .
cp /usr/lib/zookeeper/zookeeper-3.4.5-transwarp.jar .
cp /usr/lib/hbase/lib/log4j-1.2.17.jar .
cp /usr/lib/hbase/lib/commons-configuration-1.6.jar .
cp /usr/lib/hbase/lib/slf4j-api-1.6.4.jar .
cp /usr/lib/hbase/lib/slf4j-log4j12-1.6.4.jar .
cp /usr/lib/hadoop/lib/jsch-0.1.50.jar .
cp /usr/lib/hadoop/lib/jzlib-1.1.2.jar .


echo 'Your jars are ready in '$jardir
```

脚本会将上述所列的jar包拷贝到服务器上的/tmp/hbase-jars下。您要在本地进行开发还需要hbase-site.xml文件。这个文件在集群中任意一台服务器上的/etc/hbase/conf/目录下。

关于我们：

星环信息科技（上海）有限公司是一家大数据领域的高科技公司，致力于大数据基础软件的研发。星环科技目前掌握的企业级Hadoop和Spark核心技术在国内独树一帜，其产品Transwarp Data Hub (TDH)的整体架构及功能特性堪比硅谷同行，在业界居于领先水平，性能大幅领先Apache Hadoop，可处理从GB到PB级别的数据。星环科技的核心开发团队参与部署了国内最早的Hadoop集群，并在中国的电信、金融、交通、政府等领域的落地应用拥有丰富经验，是中国大数据核心技术企业化应用的开拓者和实践者。星环科技同时提供存储、分析和挖掘大数据的高效数据平台 和服务，立志成为国内外领先的大数据核心技术厂商。

行业地位：

来自知名外企的创业团队，成功完成近千万美元的A轮融资，经验丰富的企业级Hadoop发行版开发团队，国内最多落地案例。

核心技术：

高性能、完善的SQL on Hadoop、R语言的并行化支持，为企业数据分析与挖掘提供优秀选择。

应用案例：

已成功部署多个关键行业领域，包括电信、电力、智能交通、工商管理、税务、金融、广电、电商、物流等。

地址：上海市徐汇区桂平路481号18幢3层301室（漕河泾新兴技术开发区）
邮编：200233
电话：4008-079-976
网址：www.transwarp.io