任务1

连接数据库

```
%load_ext sql
```

```
import pymysql
pymysql.install_as_MySQLdb()
%sql mysql://stu2000013058:stu2000013058@162.105.146.37:43306
```

```
%sql use stu2000013058;
```

- 创建testInedx

```
 %%sql
DROP TABLE IF EXISTS testIndex;
SET @i = 0;
CREATE TABLE testIndex SELECT (@i :=@i + 1) AS id, userId AS A, movieId AS B,
tag AS C FROM dataset.tags;
```

针对A列的分组和自连接操作，观察A列上建立索引前后的性能差异

- 建立索引前：分组操作

```
%%time
%%sql

SELECT min(B) FROM testIndex GROUP BY A;
```

- 建立索引前：自连接操作

```
%%time
%%sql

SELECT * FROM testIndex AS tmp1 JOIN testIndex AS tmp2 ON tmp1.A = tmp2.A;
```

- 建立索引

```
%%sql

CREATE INDEX my_A ON testIndex(A);
```

- 建立索引后：分组操作

```
%%time
%%sql

SELECT min(B) FROM testIndex GROUP BY A;
```

- 建立索引后：自连接操作

```
%%time
%%sql

SELECT * FROM testIndex AS tmp1 JOIN testIndex AS tmp2 ON tmp1.A = tmp2.A;
```

- 我们发现建立索引之后效率反而变低了，我们推测这是因为
  - 确实使用了索引进行查询
  - 索引的开销超过了顺序读的开销

针对select B where A类型的查询，观察基于(A, B)的组合索引相对于A上的单列索引的性能提升

- 使用A上的单列索引进行查询

```
%%time
%%sql

SELECT B FROM testIndex WHERE A = 424;
```

- 删除A上的索引，建立(A, B)上的组合索引

```
%%sql

DROP INDEX my_A ON testIndex;
CREATE INDEX my_A ON testIndex(A, B);
```

- 使用基于(A, B)的组合索引进行查询

```
%%time
%%sql

SELECT B FROM testIndex WHERE A = 424;
```

- 在这个查询中，使用组合索引优于单列索引

对字符串的子字符串建立索引，观察函数索引的作用

- 建立索引前的三次查询(查询前3个字母、前两个字母和查询全部)

```
%%time
%%sql

SELECT * FROM testIndex WHERE SUBSTRING(C, 1, 3) = "pow";
```

```
%%time
%%sql

SELECT * FROM testIndex WHERE SUBSTRING(C, 1, 2) = "po";
```

```
%%time
%%sql

SELECT * FROM testIndex WHERE C = "powerful ending";
```

- 建立对前三个字母的索引

```
%%sql

CREATE INDEX my_C ON testIndex(C(3));
```

- 重新进行查询

```
%%time
%%sql

SELECT * FROM testIndex WHERE SUBSTRING(C, 1, 3) = "pow";
```

```
%%time
%%sql

SELECT * FROM testIndex WHERE SUBSTRING(C, 1, 2) = "po";
```

```
%%time
%%sql

SELECT * FROM testIndex WHERE C = "powerful ending";
```

- 删除索引

```
%%sql

DROP INDEX my_C ON testIndex;
```

- 可以看到，前3个字母和总字符串的查询的时间均发生了改变，意味着会从索引处进行查询，而前2个字母的查询时间没有发生太大变化，我们推测此时没有使用索引

任务2

- 连接数据库

```
%load_ext sql

import pymysql
pymysql.install_as_MySQLdb()
%sql mysql://stu2100013111:stu2100013111@162.105.146.37:43306

%sql use stu2100013111;
```

基于集合：
首先，使用C1，将会话的开始时间和结束时间以及对应的类型从 dbo.Sessions 表中选择出来。（其中 +1 表示开始，-1 表示结束）
其次，在C2 中，基于C1的结果，计算每个时间点上的并发会话数量，通过对类型进行累积求和并按照应用程序和时间戳进行排序。
最后，在C2 的结果上进行聚合操作，按照应用程序分组，并选择每个应用程序的最大并发数量。

```
%%time
%%sql

WITH TimePoints AS
(
    SELECT app, starttime AS ts FROM dbo.Sessions
),
Counts AS
(
    SELECT app, ts,
        (SELECT COUNT(*)
        FROM dbo.Sessions AS S
        WHERE P.app = S.app
            AND P.ts >= S.starttime
            AND P.ts < S.endtime) AS concurrent
    FROM TimePoints AS P
)
SELECT app, MAX(concurrent) AS mx
FROM Counts
GROUP BY app;
```

基于游标：
通过游标来循环遍历游标中的数据行，跟踪并发的数量，并进行更新，最终将结果取出。

```
%%time
%%sql

DECLARE
  @app AS varchar(10),
  @prevapp AS varchar (10),
  @ts AS datetime,
  @type AS int,
  @concurrent AS int,
  @mx AS int;

DECLARE @AppsMx TABLE
(
  app varchar (10) NOT NULL PRIMARY KEY,
  mx int NOT NULL
);

DECLARE sessions_cur CURSOR FAST_FORWARD FOR
  SELECT app, starttime AS ts, +1 AS type
  FROM dbo.Sessions

  UNION ALL

  SELECT app, endtime, -1
  FROM dbo.Sessions

  ORDER BY app, ts, type;

OPEN sessions_cur;

FETCH NEXT FROM sessions_cur
  INTO @app, @ts, @type;

SET @prevapp = @app;
SET @concurrent = 0;
SET @mx = 0;

WHILE @@FETCH_STATUS = 0
BEGIN
  IF @app <> @prevapp
  BEGIN
    INSERT INTO @AppsMx VALUES(@prevapp, @mx);
    SET @concurrent = 0;
    SET @mx = 0;
    SET @prevapp = @app;
  END

  SET @concurrent = @concurrent + @type;
  IF @concurrent > @mx SET @mx = @concurrent;

```

```
  FETCH NEXT FROM sessions_cur
    INTO @app, @ts, @type;
END

IF @prevapp IS NOT NULL
  INSERT INTO @AppsMx VALUES(@prevapp, @mx);

CLOSE sessions_cur;

DEALLOCATE sessions_cur;

SELECT * FROM @AppsMx;
```

```
%%time
%%sql

WITH C1 AS
(
  SELECT app, starttime AS ts, +1 AS type
  FROM dbo.Sessions

  UNION ALL

  SELECT app, endtime, -1
  FROM dbo.Sessions
),
C2 AS
(
  SELECT *,
    SUM(type) OVER(PARTITION BY app ORDER BY ts, type
                   ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cnt
  FROM C1
)
SELECT app, MAX(cnt) AS mx
FROM C2
GROUP BY app;

```

基于窗口函数；
使用窗口函数来减少游标的额外开销，进行关于app的排序进行筛选，比游标进一步提高效率

结论

- 基于集合的方案

运行时间随着数据量增加，近似指数爆炸式增长，效率最差

- 基于游标的方案

运行时间成一个线性函数增长，效率一般

- 基于窗口函数的方案

运行时间平稳，增长极为缓慢，效率高

```
集合        游标        窗口
```

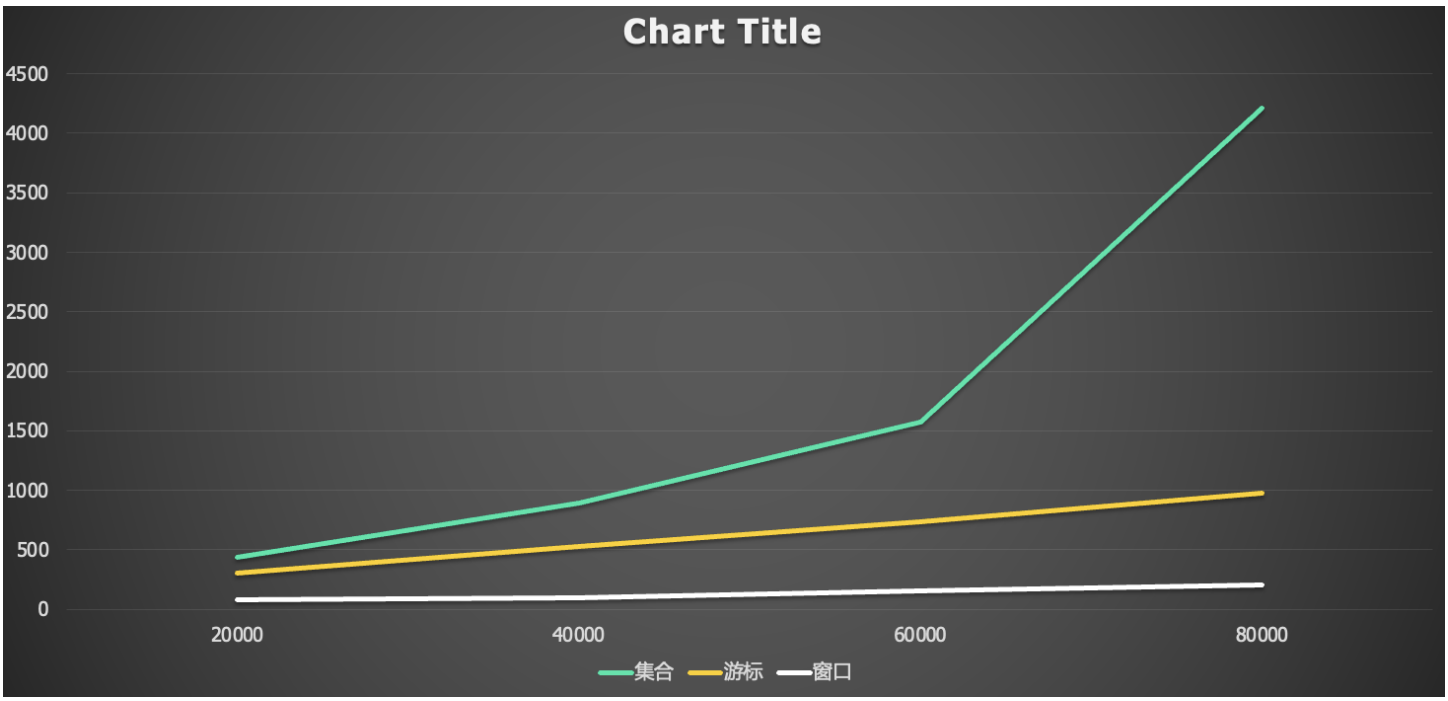
20000 435 301 77

40000 890 528 96

60000 1573 737 154

80000 4209 978 205

100000 7199 763 229



任务3

- 连接数据库

```
db = pymysql.connect(host='162.105.146.37', user='stu2000013058',
passwd='stu2000013058', port=43306, db = 'stu2000013058')
cursor = db.cursor()

%sql mysql://stu2000013058:stu2000013058@162.105.146.37:43306
%sql use stu2000013058;
```

- 创建skyline数据表，同时，建立一个分区的my_skyline数据表

```
%%sql

DROP TABLE IF EXISTS skyline;
CREATE TABLE skyline(
    id INT PRIMARY KEY,
    x INT,
    y INT
);

DROP TABLE IF EXISTS my_skyline;
CREATE TABLE my_skyline(
    id INT,
    x INT,
    y INT
)
    PARTITION BY RANGE(x)
    (
        PARTITION m0 VALUES LESS THAN(10),
        PARTITION m1 VALUES LESS THAN(20),
        PARTITION m2 VALUES LESS THAN(30),
        PARTITION m3 VALUES LESS THAN(40),
        PARTITION m4 VALUES LESS THAN(50),
        PARTITION m5 VALUES LESS THAN(60),
        PARTITION m6 VALUES LESS THAN(70),
        PARTITION m7 VALUES LESS THAN(80),
        PARTITION m8 VALUES LESS THAN(90),
        PARTITION m9 VALUES LESS THAN(101)
    );
```

- 生成随机数据

```
import random

cur_sql = "INSERT INTO skyline(id, x, y) VALUES (%d, %d, %d)"
my_cur_sql = "INSERT INTO my_skyline(id, x, y) VALUES (%d, %d, %d)"
n_samples = 1000
max_val = 100

for i in range(n_samples):
    x = random.randint(0, max_val)
    y = random.randint(0, max_val)
    cursor.execute(cur_sql % (i, x, y))
    cursor.execute(my_cur_sql % (i, x, y))

db.commit()
```

- 运行原始skyline查询，看看运行效率如何，同时简单观察一下原始集规模大小和相关性对结果集大小的影响

```
%%time
%%sql
```

```
3
4  SELECT *
5  FROM skyline h
6  WHERE NOT EXISTS (
7      SELECT *
8      FROM skyline h1
9      WHERE h1.x <= h.x
10     AND h1.y <= h.y
11     AND (h1.x < h.x OR h1.y < h.y)
12 );
```

- 1000条数据运行时间为19.4 ms，10000条数据39.1 ms
- 我们对表中的x，y均建立索引

```
1  %%sql
2
3  CREATE INDEX my_x ON skyline(x);
4  CREATE INDEX my_y ON skyline(y);
```

- 然后再运行上述查询

```
1  %%time
2  %%sql
3
4  SELECT *
5  FROM skyline h
6  WHERE NOT EXISTS (
7      SELECT *
8      FROM skyline h1
9      WHERE h1.x <= h.x
10     AND h1.y <= h.y
11     AND (h1.x < h.x OR h1.y < h.y)
12 );
```

- 最后，我们对分区表进行查询，查看其性能

```
1  %%time
2  %%sql
3
4  SELECT *
5  FROM my_skyline h
6  WHERE NOT EXISTS (
7      SELECT *
8      FROM my_skyline h1
9      WHERE h1.x <= h.x
10     AND h1.y <= h.y
11     AND (h1.x < h.x OR h1.y < h.y)
12 );
```

- 在本数据集上，分区表执行的时间要慢于原表