Group skyline 实验报告

丁霄汉 2017312365 吴超月 2017213865 罗 瑶 2017213866

1. 实验目的

实现论文《Finding Pareto Optimal Groups: Group-based Skyline》中的 G-Skyline 算法。

2. 实验任务

主要包括实现文中生成 skyline layers 的算法并对结果集进行预处理生成 directed skyline graph、point-wise 算法的实现、unit-wise+算法的实现。
注：由于 unit-wise+在 uni-wise 算法基础之上进行了进一步的剪枝，因此在实验中我们直接实现了 unit-wise+算法。

2.1 生成 DSG

生成 skyline layers 并构造后续算法所需的数据结构 directed skyline graph。

2.1.1 基本思路

主要分为以下几个步骤：
1. 获得给定的数据集后，按照第一维度属性值升序排序；
2. 判断属性维度，如果等于 2 则直接使用二维下的方法进行 skyline layers 生成，否则进入高维属性下的 skyline layers 生成方法；
3. 获得相应的 k 条 skyline 后，对每个节点利用论文中的预处理思想去掉不可能出现在 group skyline 中的点，生成预处理后的 DSG。

2.1.2 代码实现

1. Point 类

Point 类用于存放从文件中读入的每条数据，成员变量包括用于标记当前数据的标号以及存放各个维度数据的 double 类型数组。Point 类实现了 Comparable 的 compareTo 方法，使得 Point 可以按照其第一维度属性值进行升序排序。

此外，Point 类包含一个 isDominatedBy 方法利用定义判断当前点是否被给定的点所支配：如果当前点某个维度上的属性值比给定的点小，则不被其支配；如果所有维度上的属性值都不比给定的点相应的值小，且至少存在一个维度上的属性值比给定点的大，则该点被给定的点所支配：

```java
/**
 * 判断该点是否被p1支配，若被p1支配，则返回true
 * @param p1
 * @param d dimension
 * @return
 */
public boolean isDominatedBy(Point p1,int d){
    double[]attr = p1.getAttributes();
    boolean isGreater = false;
    for(int i = 0;i<d;i++){
        //如果当前有一个维度的属性比p1的小，则该点不被p1支配
        if(this.attributes[i]<attr[i]){
            return false;
        }
        //要求如果p1支配当前点，则必须有一位属性比当前点的小
        if(this.attributes[i]>attr[i]){
            isGreater = true;
        }
    }
    return isGreater;
}
```

2. DSGNode 类

DSGNode 类用于表示 DSG 中的每一个节点，成员变量包括用于标记当前数据标号的 pointIndex、记录数据所在的 skyline 层号的 layerIndex、存放各个维度数据的 double 类型数组 attributes、记录其所有父节点的列表 parents、记录其所有子节点的列表 children 以及记录属性维度个数的整形变量 d。

3. ProcessResult 类

ProcessResult 类保存程序执行过程中需要记录的数据，包括一个存放所有供后续算法使用的 DSGNode 列表、存放在预处理中已经找到的符合要求的 group 以及存放最终结果集的列表。

4. DSGGenerator 类

DSGGenerator 类为生成 directed skyline graph 的主要处理类，成员变量包括所生成的 group skyline 中的成员个数 k、传入的待分组数据集、记录生成的 k 条 skyline 的列表以及每条数据的维度个数 d。

在调用 generateDSG 方法生成 DSG 时，该方法首先调用成员方法 generateSkylines 生成 k 条 skyline。在 generateSkylines 方法中，首先对数据集按照第一维度属性升序排序，然后判断属性的总维度，若为 2，则调用 generate2D 方法利用论文中的 algorithm1 生成 k 条 skyline，否则调用 generateHighDimension 方法使用简单的遍历生成 k 条 skyline。在获得 k 条 skyline 后，利用论文中的方法对数据集进行预处理，并最终生成供后续使用的 DSG。

在调用 generateSkylines 方法生成 k 条 skyline 后，对每个节点进行预处理，筛选掉不可能出现在 group skyline 中的点，即判断其 unit group 的大小是否大于 k，如果大于则抛弃这些节点；再对每一个点遍历在其所在 skyline 之前的所有 skyline 上的点，判断这些点是否支配该点，若支配，则将该点加入对应点的 children 列表中，相应的将这些点加入该点的 parents 列表中，最终生成 DSG。

generate2D 方法即对论文中 algorithm1 的实现，在其中调用了 binarySearchLayer 方法加快对给定点所属 skyline 层号的判断。此外，在生成新的 skyline 时判断当前 skyline 的层数，若大于 k，则不再继续生成。

generateHighDimension 方法即将论文中的 algorithm1 算法应用到高维属性中，此时无法利用二维情况下的单调性，即需要遍历每个已加入 skyline 的节点进行判断新加入的节点属于哪个 skyline。

2.2 point-wise 算法实现

从预处理后得到的 DSG 出发，DSG 中的 DSGNode 已经按照 index 由小到大排列，用 point-wise 方法计算 group skyline

2.2.1 基本思路

point-wise 方法的基本思路就是从空集开始，逐层生成大小为 1，2…k-1，k 的 G-skyline groups。当已经得到大小为 i 的 G-skyline groups 之后，遍历每个大小为 i 的 G-skyline group，每个 G-skyline group 都有一个尾集合，尾集合包含所有 index 大于 G-skyline group 中所有点的 index 的点，遍历尾集合中的点，向 G-skyline group 中增加尾集合中的点构成大小为 i+1 的 candidate group，然后检查该 candidate group 是否是 G-skyline group，如果是，就将 candidate group 加入大小为 i+1 的 G-skyline groups。point-wise 尽可能的在每一步剪枝。以下是 point-wise 的几个剪枝方法：

（1） Subtree Pruning：一旦某个大小为 i 的 candidate group 被检查之后发现不是 G-skyline group，那么往其中加入任何点都不能构成大小为 i+1 的 G-skyline group.所以我们可以直接舍弃这个点不存储

（2） Tail Set Pruning：

A、G-skyline group中如果包含某个点p，那么点p的父母节点一定也在这个G-skyline group中。所以我们可以对G-skyline group的tail set进行删减，我们首先计算得到G-skyline group中所有节点的孩子节点集合，tail set中的点如果不在孩子节点集合中或者属于第一层skyline则直接从tail set中删除该点。这样可以减少candidate groups的大小。

B、假设G-skyline group中节点的最大层数为i，则加入的点p必须来自前i+1层。所以tail set中的点如果层数大于i+1也被直接删除。

总的来说，point-wise 包含三个循环，最外的循环每循环一次生成一层的所有 G-skyline groups，层数从 1 到 k；在生成每一层的 G-skyline groups 时，循环遍历所有上一层得到的的 G-skyline groups，遍历每一个上一层的 G-skyline group 时，对 tail set 进行剪枝；然后遍历剪枝后的 tail set，每次遍历得到一个 candidate group，检查是否为 G-skyline group，如果是，加入集合，否则直接舍弃。

2.2.2 代码实现

代码使用 java 实现。

（1） 每一个 G-skyline group 使用 List<Integer>表示，每个节点使用其 index 代表。最后得到的大小为 k 的 G-skyline groups 用 List<List<Integer>> groupListNew 表示。

（2） 使用 int[] children 来统计每个 G-skyline group 的孩子集合，下标 i 中的整数为 1 代表 index 为 i 的节点是 G-skyline group 的孩子，为 0 代表不是孩子。具体代码实现如下，遍历 G-skyline group 的每个节点的孩子节点列表，将 children 数组对应于该孩子节点的下标 index 的整数设为 1。

```
for(int nodeIndex : list) {
    List<Integer> childrenList = DSG.get(nodeIndex).getChildren();
    for(int child : childrenList) {
        children[child] = 1;
    }
}
```

（3） 使用 int[] tailList 来统计删减过后的尾集合。下标 i 中的整数为 1 代表 index 为 i 的节点是没有删减的节点。具体实现如下：

```
for(int t = tailIndex ; t < Sk ; t++) {
    //g的tail Set从 tailIndex 开始
    DSGNode node = DSG.get(t);
    if(children[t] == 0 && !(node.getLayerIndex() == 1)) {
        //如果节点没有在子节点集合中并且不是skyline节点，则不加入
        continue;
    }
    else if(node.getLayerIndex() - maxLayer >= 2) {
        //如果节点的layer比最大layer大及2层以上，不加入
        break;
    }
    else
    {
        tailList[t] = 1;
    }
}
```

（4） 遍历 tailList，如果下标为 i 的位置整数为 1，将 index 为 i 的节点加入 G-skyline group，检查是否为新的 G-skyline group，这里使用了一个检查是否为 G-skyline group 的检查方法：检查 candidate group 的 unit group（自己的父母节点加上自身的集合）的大小是否等于 candidate group 自身的大小，这里使用 java 的数据结构 Set / HashSet 来去重检查。具体代码实现如下：

```
for(int candidate = 0 ; candidate < Sk ; candidate ++) {
    if(tailList[candidate] == 0)continue;
    set1.addAll(list);
    set1.addAll(DSG.get(candidate).getParents());
    set1.add(candidate);
    if(set1.size() == i) {
        List<Integer> gNew = new ArrayList<Integer>(k);
        gNew.addAll(list);
        gNew.add(candidate);
        if(i == k) {
            groupListNew.add(gNew);
        }
        else
        {
            groupList.add(gNew);
        }
        tmpCount ++;
        set1.clear();
    }
    else {
        set1.clear();
        continue;
    }
}
```

最后 groupListNew 中存储所有大小为 k 的 G-skyline groups，groupListNew 加上 DSG 预处理阶段得到的 unit 大小为 k 的节点集合 perfectNodeList 就是所有的解。

2.3 unit-wise+算法实现

从 DSG 出发，用 unit-wise 方法计算 group skyline。

2.3.1 基本思路

主要分为以下几个步骤：

1. 以上一步骤输出的 DSG 为输入，简化数据结构，构造 RefDSG，即 Reference-based DSG。这是因为上一步输出的 DSG 的节点 DSGNode 是通过 List 中的下标来存储其父母节点和孩子节点的，而在 unit-wise 方法中，在涉及 unit group 的操作时需要频繁查询和合并许多节点的父母节点和孩子节点。将 DSGNode 转换为基于引用的 RefNode，用 HashSet 保存节点的父母节点和孩子节点的引用，并抛弃无用属性，可以提高效率。另一方面，将排序后的 unit group 和 node 作为 RefDSG 的成员变量，可以提高抽象层次，提高可维护性。
2. 按照论文中的实现方式，将 nodes 和 unit groups 按照原本的索引反向排序。
3. 构造仅由单一 unit group 组成的初始 groups 列表，即 singleUnitGroups。
4. 对每一个初始 group：
   (1) 检查 G^{last}。
   (2) 以该 group 为初始节点，构建枚举树，并按照论文中的方法进行剪枝。在这里我们结合 java 的语言特性做了一些改进。

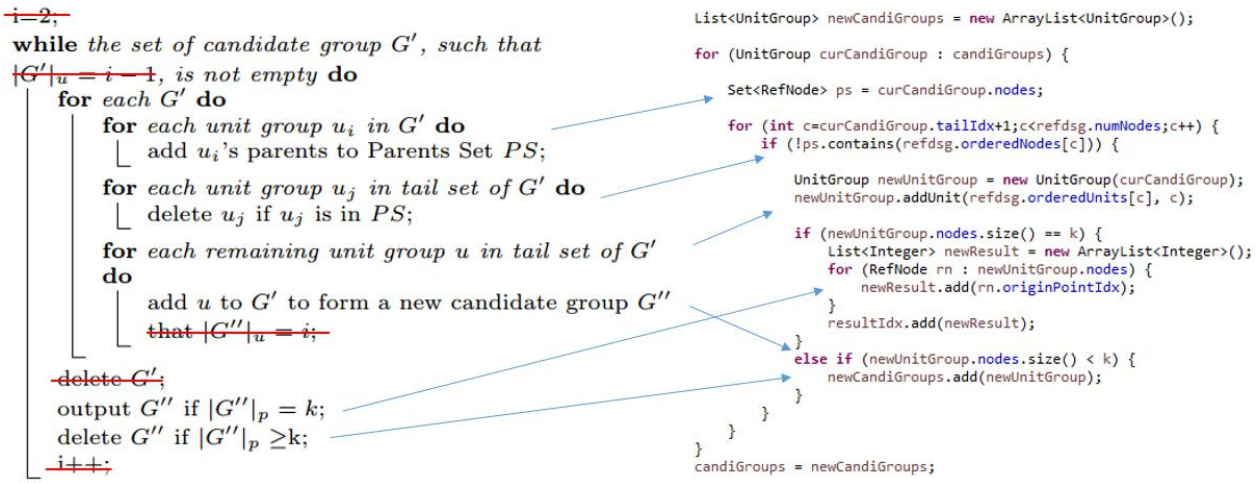
最终输出结果。

2.3.2 代码实现

1. **RefNode：**
   基于引用的 DSG Node。用 originPointIdx 表示原本的节点索引，parents 表示其所有的祖先节点，children 表示其所有的后代节点。这里沿用了论文中 parent 和 children 的称谓，虽然这个命名不是很精确。这里使用 HashSet 是为了提高查询和合并的效率。
2. **Unit 类**
   单个 unit group，记录其起始节点和所有长辈节点。
3. **UnitGroup 类**
   由 unit group 组合而成的 group。记录其在枚举树 tail list 中的位置和包含的节点。
4. **RefDSG**
   基于引用的 DSG。保存排序的 nodes 和 units。
5. **UnitWise 类**
   负责应用论文中描述的 unit-wise 方法进行计算并输出结果。

在具体的实现上，我们结合 java 的语言特性对论文中的方法进行了优化。以下面这段代码为例。

论文中所说的一个 node 的 parent set 其实可以等价替换为对应 unit group 的 nodes 集合。



```
~~i=2;~~
while the set of candidate group G', such that
~~|G'|_u = i-1~~, is not empty do
    for each G' do
        for each unit group u_i in G' do
            add u_i's parents to Parents Set PS;
        for each unit group u_j in tail set of G' do
            delete u_j if u_j is in PS;
        for each remaining unit group u in tail set of G'
        do
            add u to G' to form a new candidate group G''
            ~~that |G''|_u = i;~~
    ~~delete G';~~
    output G'' if |G''|_p = k;
    delete G'' if |G''|_p ≥k;
    ~~i++;~~
```

```
List<UnitGroup> newCandiGroups = new ArrayList<UnitGroup>();

for (UnitGroup curCandiGroup : candiGroups) {

    Set<RefNode> ps = curCandiGroup.nodes;

    for (int c=curCandiGroup.tailIdx+1;c<refdsg.numNodes;c++) {
        if (!ps.contains(refdsg.orderedNodes[c])) {

            UnitGroup newUnitGroup = new UnitGroup(curCandiGroup);
            newUnitGroup.addUnit(refdsg.orderedUnits[c], c);

            if (newUnitGroup.nodes.size() == k) {
                List<Integer> newResult = new ArrayList<Integer>();
                for (RefNode rn : newUnitGroup.nodes) {
                    newResult.add(rn.originPointIdx);
                }
                resultIdx.add(newResult);
            }
            else if (newUnitGroup.nodes.size() < k) {
                newCandiGroups.add(newUnitGroup);
            }
        }
    }
}
candiGroups = newCandiGroups;
```

这两个集合的差集就是这个 node 本身，这一差别对剪枝没有影响。这个集合其实我们已经在之前求出来了，因而大大提高了效率。

在求 tail set 的时候，我们没有像论文中一样“删去在 parent set 中的 unit group”，而是“只添加不在 parent set 中的 unit group”。考虑到 java 中 ArrayList 的删除操作效率低下，LinkedList 的插入操作效率又低于 ArrayList，这样做可以使我们既使用 ArrayList 又避免删除操作，提高了效率。

文中使用变量 i 来索引不同深度的枚举集合。考虑到 java 的回收机制，我们直接用新的枚举集合替代旧的，以使得旧的集合占据的资源可以被释放和回收。

我们对论文提出算法的若干循环做了最大限度的合并，以提高效率。

3. 实验结果

实验在给定的三种数据分布（inde、corr、anti）的 4 种维度（2、4、6、8）下共 12 组数据上进行。实验的 group size k 取值分别为 2、4、6。在上面的两种算法实现描述中，均将所有的结果放入内存中，但是我们在实验中发现，当结果数目较大（上十亿）时，会发生内存溢出，而在实际的应用中，结果集也不是存在内存中的，所以我们在两个算法中在构造出每个结果后，没有将其加入结果集，而只是统计了结果数。

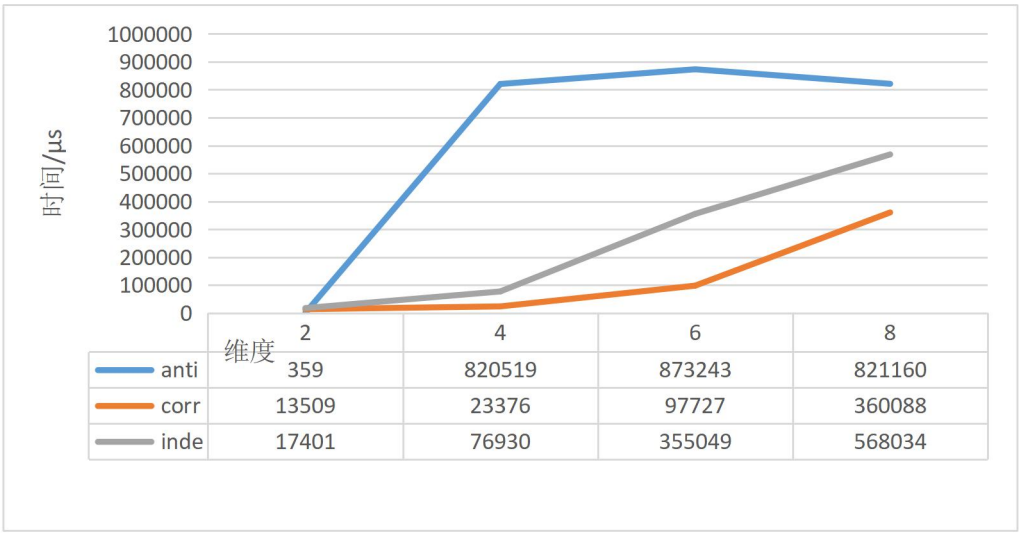
实验环境：

处理器:	Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz 3.60 GHz
已安装的内存(RAM):	16.0 GB (15.9 GB 可用)
系统类型:	64 位操作系统，基于 x64 的处理器

```
C:\Users\Administrator>java -version
java version "9.0.1"
Java(TM) SE Runtime Environment (build 9.0.1+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
```

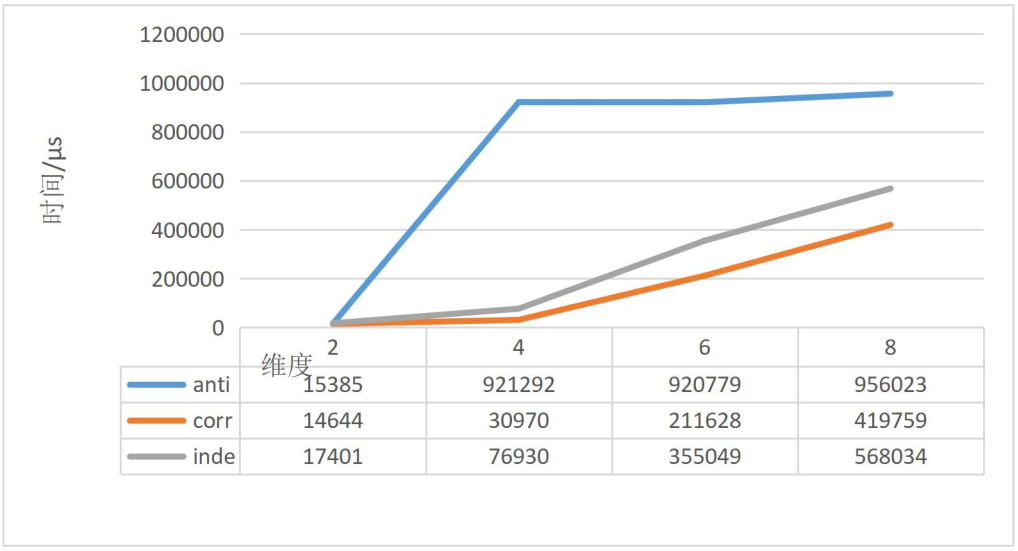
3.1 生成 skyline layers 实验结果

图表 1、图表 2、图表 3 分别展示了 k 取 2、4、6 时在上述 12 种数据集上生成 skyline layers 的耗时。在维度为 2 时采取二分查找的方式进行插入，而在高维度时采取简单的遍历方式。通过图表可以发现，在 k 取不同值、在具有不同分布的数据集上生成 skyline layers 时，维度为 2 时的耗时要明显小于高维情况，也就是说，论文中针对维度为 2 时的二分处理具有较好的性能优势。



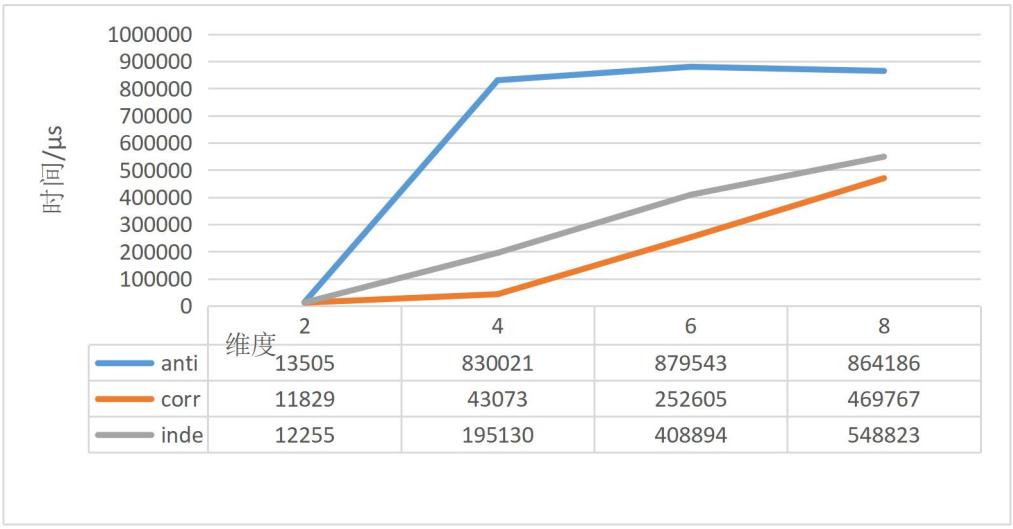
维度	2	4	6	8
anti	359	820519	873243	821160
corr	13509	23376	97727	360088
inde	17401	76930	355049	568034

图表 1



维度	2	4	6	8
anti	15385	921292	920779	956023
corr	14644	30970	211628	419759
inde	17401	76930	355049	568034

图表 2



图表 3

3.2 group size k=2 时实验结果

表 1 展示当 k=2 时在 12 组数据上使用 point-wise 和 unit-wise+算法计算 group skyline 的结果。图表 4、5、6 分别展示两种算法在三种不同分布的数据集上随着维度变化相应时间的变化。

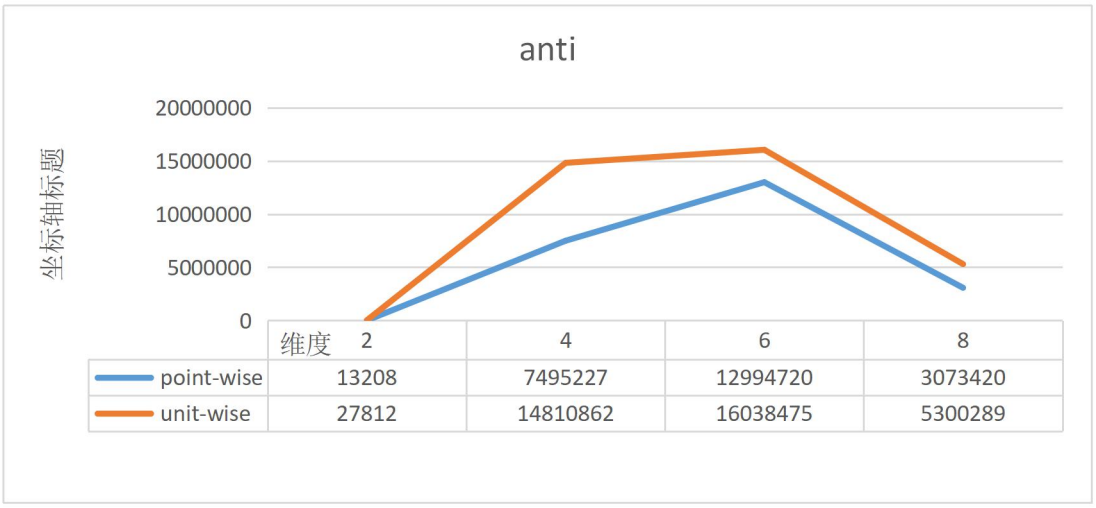
从表 1 中可以看出，g-skyline size 和算法运行时间与预处理之后的点数指数级相关。一般情况下，随着维度的增加，预处理之后的点数相应明显增加，因此最终体现为随着维度的增加，算法运行时间呈现指数级增长（见图表 5、图表 6）；而 anti 数据集在维度为 4、6、8 时，预处理之后的点数集合都非常大并且也与维度不呈现正相关，所以体现为 anti 数据集在维度为 4、6、8 时处理时间都非常长，甚至维度为 8 时算法运行时间比维度为 4 和 6 的情况下短（见图表 4）。

同时，k=2 时，三个数据集 12 组数据中 point-wise 算法都比 unit-wise+表现好。

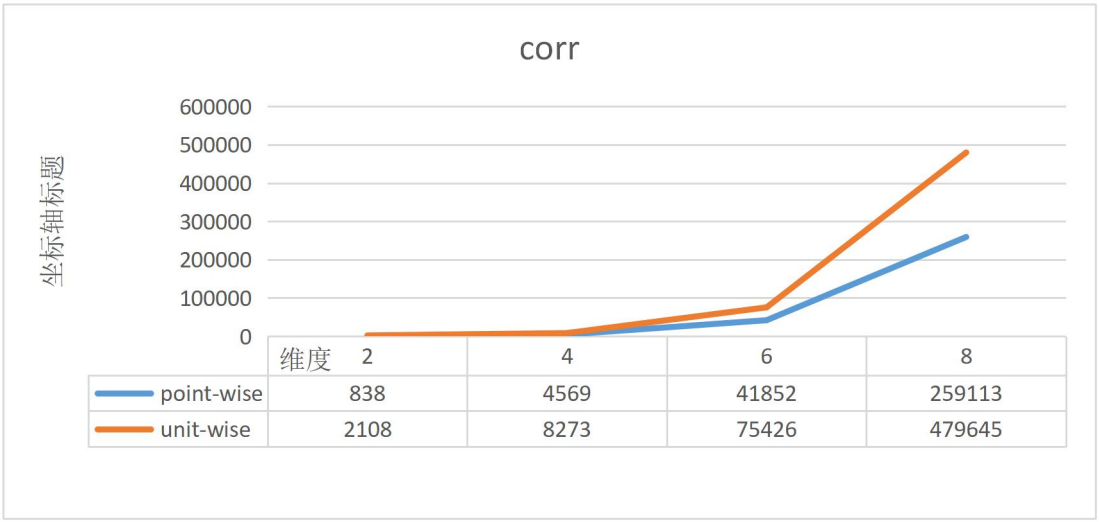
	point-wise/ μ s	unit-wise+/ μ s	前 k 层天际线点数	预处理之后的点数	g-skyline size
anti_2	13208	27812	359	129	9717
anti_4	7495227	14810862	10000	8960	40137105
anti_6	12994720	16038475	10000	9386	44044277
anti_8	3073420	5300289	10000	5378	14460073
corr_2	838	2108	3	1	2
corr_4	4569	8273	10000	35	632
corr_6	41852	75426	10000	404	81648
corr_8	259113	479645	10000	1520	1155104

inde_2	1012	2844	31	10	47
inde_4	21518	33213	10000	172	14828
inde_6	114242	246345	10000	887	393429
inde_8	679000	1578361	10000	2794	3902800

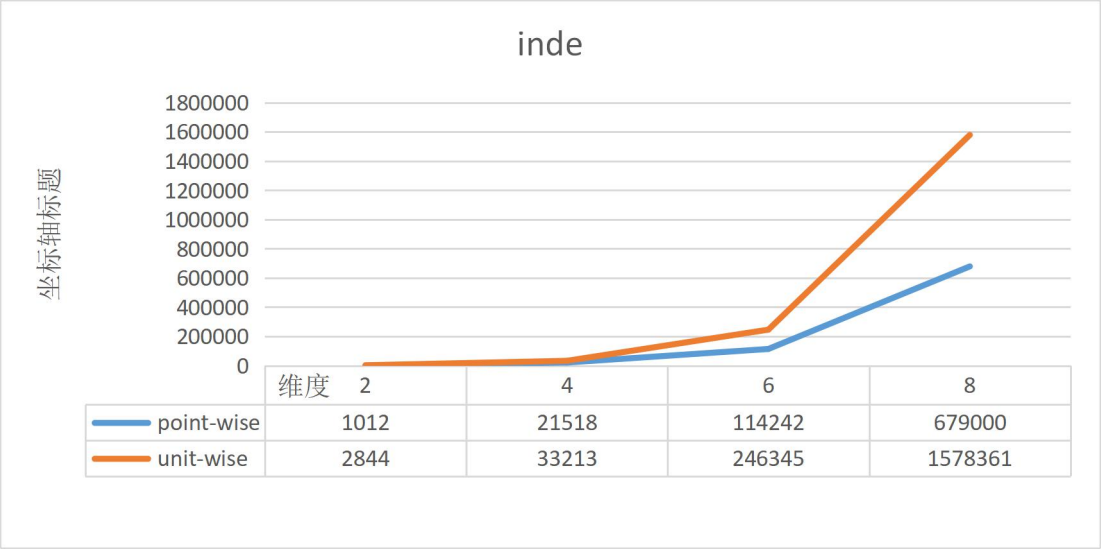
表 1



图表 4



图表 5



图表 6

3.3 group size k=4 和 6 时实验结果

表 2 和表 3 分别展示 k 取 4 和 6 时的实验结果。

当 k 取 4 时，如表 2 所示，预处理之后的点数较小（约在 400 以下）的数据集能够顺利地在较短的时间内得出结果；数据集 corr_6 预处理之后的点的数目到达 844 时，处理时间需要两分钟，结果集大小到达十亿，数据集 inde_6 预处理之后的点的数目到达 1682 时，处理时间则达到了将近一个半个小时，结果集数目达到 25808200629；而当预处理之后的点数更大的时候（超过 2000），结果数据集的大小只能更大，两种算法在有限的计算时间都无法计算出结果集，所以我们没有收集到运行时间。

我们以 point-wise 算法为例，point-wise 算法需要同时存储第 k 层之前的每一层的所有 G-groups，所以我们统计了 Sk 数目从 390 到 2651 的数据集在计算过程中产生的中间结果集的大小，如表 4 所示。

当 Sk 比较大（844），第三层的所有 G-groups 数目到千万级，point-wise 算法占用的内存特别大。

当 Sk 更大一点（1682），第二层的所有 G-groups 数目就有几十万，对应的第三层的 G-groups 数目上亿，最终结果集的大小到达 25808200629。

当 Sk 非常大（2000 以上）时，内存已经不足以能够分配出能够装下中间结果的大小，此时 point-wise 算法就失效了。我们可以看到 corr_8 的第三层的结果集就已经超过了内存大小。

我们也可以看到当 k=6 时，Sk 到达 200 以上 point-wise 算法需要存储的中间结果的数目就已经超过了内存可以分配的极限，point-wise 算法失效。此时 unit-wise+可以比较好的工作。

但是当 Sk 比较大时，即使利用 unit-wise+计算得到了一个包含着几十亿上百亿结果也是没有任何实际意义的，这是本文算法极大的局限性。

	point-wise/ μ s	unit-wise+/ μ s	前 k 层天际线点数	预处理之后的点数	g-skyline size
anti_2	3163941	1992135	918	390	16124810

anti_4			10000	9915	
anti_6			10000	9951	
anti_8			10000	7382	
corr_2	507	2067	9	6	7
corr_4	33849	42707	10000	90	71701
corr_6	140062003	125487122	10000	844	1112974784
corr_8			10000	2651	
inde_2	2845	3039	77	29	440
inde_4	5889611	4160674	10000	383	36861708
inde_6	2924445368	5152130543	10000	1682	25808200629
inde_8			10000	4374	

表 2

	point-wise/ μ s	**unit-wise+/ μ s**	**前 k 层天际线点数**	**预处理之后的点数**	**g-skyline size**
anti_2			1502	575	
anti_4			10000	9981	
anti_6			10000	9992	
anti_8			10000	8124	
corr_2	979	2334	17	8	8
corr_4	5708625	3570728	10000	162	3723233
corr_6			10000	1166	
corr_8			10000	3310	
inde_2	23603	13648	128	47	2930
inde_4			10000	550	
inde_6			10000	2210	
inde_8			10000	5144	

表 3

	预处理之后的点个数 **Sk**	第一层	第二层	第三层	第四层
anti_2	390	139	9717	455623	14513512
corr_6	844	404	81639	11003536	1112974784
inde_6	1682	887	393417	116343425	25808200629
corr_8	2651	1520	1155102	——	——

表 4