

Assignment #2
Big Data Management COSC2636/2632
Submission Deadline: May 3 2019 11:59pm
Student Name: Syed Hariz
Student ID: S3701799

Introduction

This report focuses on the implementation of various types of spatial keyword query relaxations that are proposed in INSPIRE (Incremental Spatial Prefix query Relaxation). The goal of this assignment is to demonstrate running the relaxation algorithms proposed and how it works on an interface. This report focuses on brief explanation of the type of queries used to achieve this task and illustrations to further understand how each relaxation of a spatial keyword query works and helps. The types of relaxations used are; substring range query and approximate prefix range query on top of the initial or traditional approach, the spatial prefix query. JAVA language is used to program the algorithm to achieve this task.

The outline of this report will include, explanation of relaxations used from the proposed framework, explanation of the algorithms, its drawbacks and explanation of the codes that were written to construct these relaxations together with the traditional approach.

Queries

Queries are used to retrieve objects that matches a certain keyword within a spatial region. Recent studies shows that to retrieve such data efficiently, the process of a spatial keyword can determine on how efficient, effective and user-friendly when finding its matching object in the spatial region. This study focuses on various types of relaxation offered to expand and retrieve more objects that matches the keyword entered than the traditional way of achieving it which is by Spatial Prefix Query. However, this report will be focused on explaining how the traditional way works on retrieving objects from the spatial region and 2 types of relaxation used to expand its search gap and returning more objects; Substring range query and approximate prefix range query.

Spatial Prefix query

Spatial Prefix query is the traditional way of returning objects in the spatial region. The logic that was applied on constructing this algorithm is when a keyword is searched, a POI (Point of interest) is returned if it is located in the query region. The query text is a prefix of its textual content of an object. Example of Spatial Prefix query is shown in Figure 1 without error and Figure 1.1 with error.

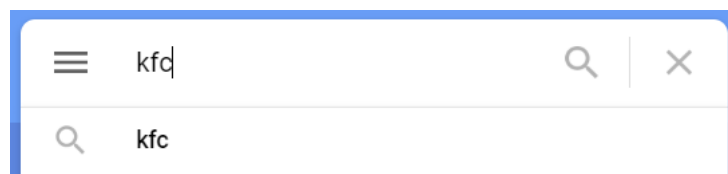


Figure 1: Exact match with searched and prefixed query

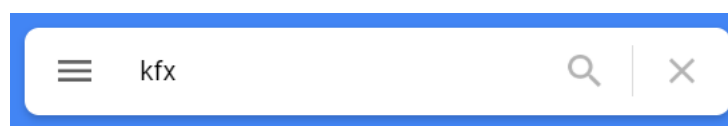


Figure 1.1: Typo in searched query, returns no results

Drawback

However, by depending on this singular query algorithm. There are some drawbacks that can be found. Since all the query texts are prefixed, it is possible that such query will not be returned due to human errors such as typos. For an exact query to be returned, the query has to be exactly the same as the textual content of the object or nothing will be returned. Secondly, the region that this query covers is small. Therefore, if searched query is outside the user's region, it will not be returned. Figure 2 shows how an object is not returned when it is outside the user's region.

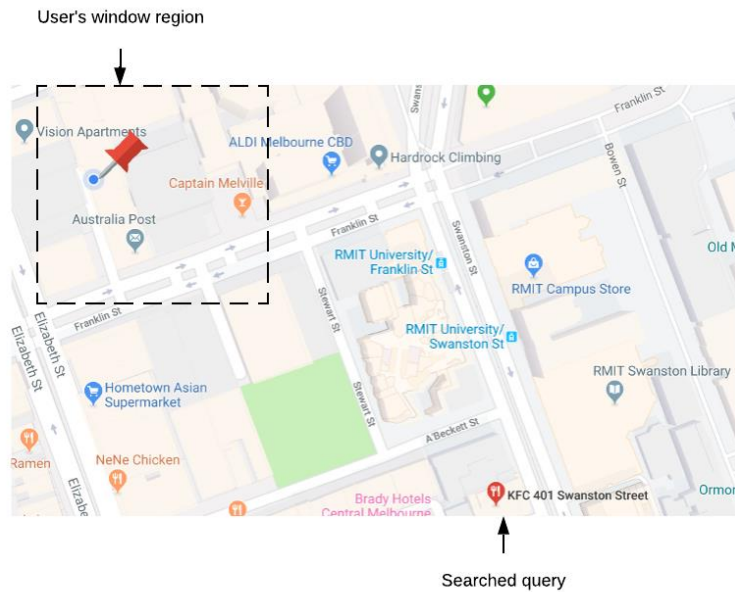


Figure 2: Searched object missed out from user's region

Substring range query

Substring range query is one of the relaxations that were presented by INSPIRE incremental framework. Spatial Substring (SS) query is matching the query searched from the user with the string of an object that contains the keyword. This algorithm is used on top of the Spatial Prefix to relax its condition of only returning an object with exact keywords in the query. The algorithm tackles this problem by matching a query text that contains in the textual content of an object in the spatial region. Figure 3 illustrates how a searched object is still returned with incomplete query text.

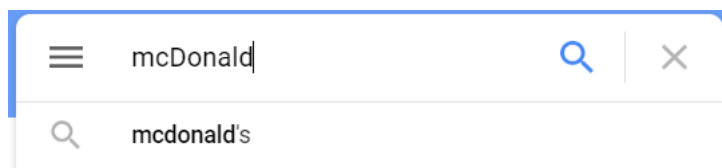


Figure 3: Object is returned with incomplete query

Drawback

In such cases, a drawback for this algorithm can be found when no results are returned when the searched query is different than objects that exists in the spatial region. For example, if a user searches for a *staples centre*, it only focuses on the textual content of an objects that contains a substring of it. Therefore, results that can possibly be returned are; *the staples centre*, *staples centre*, *the staples*. The goal of INSPIRE framework is to return as much objects with the closest match of searched query to enhance user-friendly towards the system. Therefore, multiple relaxations are included in this framework.

Approximate prefix range query

Spatial Approximate Prefix (SAP) query is returning an object/objects which has the closest match between the object's string and the user's query. How this algorithm enhances the INSPIRE framework is by allowing mismatches in the keyword search but limiting the matching with the start of an object's textual content. Therefore, by applying this algorithm it will help to return more objects with closest matches with the searched query. How a mismatch limit is determined on a textual content of an object is depending on how many q-grams length is proposed. Figure 4 shows how approximate prefix query expands the search gap of a query. A user searches for *staples center* in its query region point A, a q-gram length of 3 is used to match user's query with an object's textual content. Therefore, *stables center* point D is also returned since first 3 characters of point D matches the first 3 characters of the searched query.



Figure 4: Object with mismatch

Drawback

As proposed, the INSPIRE framework goal is to return accurate and varieties of results to the user. In such situation, this algorithm can also but rarely return less number of results to users. A more complex algorithm is proposed to tackle this problem which is the approximate substring query. A mixture of both substring query algorithm and approximate prefix query to ensure all relevant results are being returned that exists in the spatial region. Having said that, approximate substring query algorithm is not covered in this report.

Relaxation Hierarchy

Spatial Prefixed query in a Relaxed region ➡ Spatial Substring query ➡ Spatial Approximate Prefix query ➡ Spatial Approximate Substring query

Code explanation

Function 1 – Spatial Prefix Query

`this.readInfrequentInvertedIndex(query, memInfreqPosQgramInvertedMap, memInfreqQgramTokenInvertedMap, hasReadInfrequentToken, hasInfrequentPosQgram, hasInfrequentQgramToken);`

This part of the code in function 1 is to read the q-gram inverted index. The reason why we would want to read is to check if the inverted index contains infrequent positional q-grams. If it does not contain any infrequent positional q-grams, we then check if the inverted index contains infrequent q-grams token. We read if the inverted index contains infrequent positional q-grams/q-grams token is either to store both of the values or to create them.

```
if ( hasInfrequentPosQgram.isTrue() || hasInfrequentQgramToken.isTrue() )
{
    NavigableSet<Integer> infrequentJoinedResult = null;

    // if has infrequent positional q-gram,
    if(hasInfrequentPosQgram.isTrue())
    {
        infrequentJoinedResult =
this.infrequentQgramExactJoin(memInfreqPosQgramInvertedMap.values());
    }
    // else if has the infrequent q-gram token,
    else if(hasInfrequentQgramToken.isTrue())
    {
        infrequentJoinedResult =
this.infrequentQgramExactJoin(memInfreqQgramTokenInvertedMap.values());
    }
    // check the join result
    if(infrequentJoinedResult != null)
    {
        this.browsePrefixInfrequentResult(query, infrequentJoinedResult,
objectDatabase, hasReadInfrequentToken, prefixResult, visitedObjectMap,
prefixResultFromInfrequentQgram, substringCandidates);
    }
}
```

This part of the code is to store the values of both infrequent positional q-grams and infrequent q-grams token into a NavigableSet collection which accepts type Integer. The code then does a Boolean check if both values exists before storing them, hence there are some conditional if statements of `hasInfrequentPosGram.isTrue()` and `hasInfrequentQgramToken.isTrue()`. If both values does not exist/null, we then pass this function's parameter of (query, infrequentJoinedResult, objectDatabase, hasReadInfrequentToken, prefixResult, visitedObjectMap, prefixResultFromInfrequentQgram, substringCandidates) into `browsePrefixInfrequentResult` function paramaters to visit the object database and search for a candidate.

```

else
{
    // get the intersecting node statistics
    //2 types of argument parameters
    //quadtree.getIntersectingNodeStatistic(readBefore, region,
intersectingNodeStatsMap, underLoadThreshold);
    quadtree.getIntersectingNodeStatistic(query._queryRegion,
intersectingNodeStatistic, sparseThreshold);

hasRetrievedIntersectingNodes.setTrue();

    // if the number of dense node is greater than the nodeNumberThreshold, do
the prefix node filter node first
    if ( intersectingNodeStatistic.denseNodeNumber >= visitingNodeSizeThreshold
)
    {
        // prefix node filter
        readIntersectingNodeStatsOfRelaxedRegion(query,
hasRetrievedIntersectingNodesOfRelaxedRegion,
intersectingNodeStatsMapOfRelaxedRegion);
        //2 Arguments

        prefixNodeFilter(query, intersectingNodeStatsMapOfRelaxedRegion,
substringNodeCandidateCountMap, substringNodeCandidateCountMap);

        if(prefixNodeCandidateCountMap.isEmpty())
        {
            return;
        }
        else
        {
            Iterator<Entry<String, ArrayList<Integer>>> prefixNodeItr =
prefixNodeCandidateCountMap.entrySet().iterator();

            while(prefixNodeItr.hasNext())
            {
                Entry<String, ArrayList<Integer>> entry =
prefixNodeItr.next();
                String nodeHilbertCode = entry.getKey();
                NodeStatistic nodeStats =
intersectingNodeStatistic.get(nodeHilbertCode);

                if(nodeStats != null)
                {
                    //2 types of argument parameters
                    this.prefixSingleNodeProcess(query, nodeHilbertCode,
nodeStats, prefixResult,
visitedObjectMap, prefixResultMap,
prefixResultFromNodeLevelJoin,
memHilbPosQgramInvertedMap, memHilbQgramTokenInvertedMap);
                }
            }
        }
    }
}

```

This else statement code will run if there are no infrequent q-grams. First we will need to get the NodeStatistics in the quad tree. Therefore, this part of the code:

```
quadtree.getIntersectingNodeStatistic(query._queryRegion,
intersectingNodeStatistic,
    sparseThreshold);
```

Will check if there are any intersecting nodes within the query region of the sparseThreshold value. If intersection is true, we recursively add the get the intersecting nodes and save the number of the objects in the quad tree node.

From this code, we proceed to check if the dense node is bigger or equal than the nodes of the sparsThreshold. The code of this explanation will run under if statement of:

```
if ( intersectingNodeStatistic.denseNodeNumber >= visitingNodeSizeThreshold )
```

Since the INSPIRE framework includes filters, we choose the type of filter according to which function it belongs to. In this case, it is the spatial prefix query and its filter is PoF. There are two types of filter levels included in the INSPIRE framework, Node-level and Object-level. This part of the code:

```
// prefix node filter
    readIntersectingNodeStatsOfRelaxedRegion(query,
hasRetrievedIntersectingNodesOfRelaxedRegion,
intersectingNodeStatsMapOfRelaxedRegion);
//2 Arguments

    prefixNodeFilter(query, intersectingNodeStatsMapOfRelaxedRegion,
substringNodeCandidateCountMap, substringNodeCandidateCountMap);
```

Will filter out the nodes where a node n must have atleast LBS, T, q_p matching positional qp-grams.

The final part of the code:

```
else
{
    Iterator<Entry<String, NodeStatistic>> intersectingNodeItr =
intersectingNodeStatistic.entrySet().iterator();

    while(intersectingNodeItr.hasNext())
    {
        Entry<String, NodeStatistic> entry = intersectingNodeItr.next();
        String nodeHilbertCode = entry.getKey();
        NodeStatistic nodeStats = entry.getValue();
        //2 types of parameters
        this.prefixSingleNodeProcess(query, nodeHilbertCode, nodeStats,
prefixResult, visitedObjectMap,
        prefixResultMap, prefixResultFromNodeLevelJoin,
memHilbPosQgramInvertedMap, memHilbQgramTokenInvertedMap);

    }
}
```

It will run when the number of dense node is not greater or equal than the number of threshold nodes. We iterate through the Entry collection to get its NodeStatistic value and its HilbertCode to obtain Object(s) that intersects.

Function 2 – Substring Query

```
if ( intersectingNodeStatsMap.denseNodeNumber < visitingNodeSizeThreshold )
{
    // if the node is never filtered before
    if ( doSubstringNodeFilterInPrefixQuery.isFalse() &&
doSubstringNodeFilter.isFalse() )
    {
        Iterator<Entry<String, NodeStatistic>> itr =
intersectingNodeStatsMap.entrySet().iterator();
        while(itr.hasNext())
        {
            Entry<String, NodeStatistic> entry = itr.next();
            String nodeHilbertCode = entry.getKey();
            NodeStatistic nodeStats = entry.getValue();

            //substring query in node
            //2 types of argument parameters
            //this.substringQueryInSingleNode(query, nodeHilbertCode,
nodeStats, resultMap);
            this.substringQueryInSingleNode(query, nodeHilbertCode,
nodeStats, substringResult,
                                memObjectMap, memHilbQgramTokenInvertedMap);

        }
    }
}
```

This part of the code runs similarly to the first spatial query. Since INSPIRE framework introduces a hierarchal relaxation, the codes written in each spatial query is similar to the previous one. However, the logic that was applied in this if statement is different because we want to find if the visiting node of given threshold is bigger than the dense node. Once we have find the nodes, we have to check if the nodes are already been filtered by the previous function (Spatial prefix). If not, the code in this condition will filter the nodes:

```
if ( doSubstringNodeFilterInPrefixQuery.isFalse() &&
doSubstringNodeFilter.isFalse() )
```

If the node is already filtered the else statement will run:

```
else
{
    Iterator<Entry<String, ArrayList<Integer>>> nodeItr =
substringNodeCandidateMap.entrySet().iterator();
    while(nodeItr.hasNext())
    {
        Entry<String, ArrayList<Integer>> entry = nodeItr.next();
        String nodeHilbertCode = entry.getKey();
        NodeStatistic nodeStats =
intersectingNodeStatsMap.get(nodeHilbertCode);

        if(nodeStats != null)
        {
            //2 types of argument parameters
            this.substringQueryInSingleNode(query, nodeHilbertCode,
nodeStats, substringResult, memObjectMap,
                                memHilbQgramTokenInvertedMap);

        }
    }
}
```

If the number of intersecting node is larger than the dense node, we have to filter them according to the function's filter category. In this case, the filter for substring query is CF where a node n must have at least $LBS_{T,q_c} = (|S| - q_c + 1) - q_c * T$ matching q_c -grams. This line of code will iterate through the candidate nodes to get its Hilbert code and node statistics:

```
Iterator<Entry<String, ArrayList<Integer>>> nodeItr =
substringNodeCandidateMap.entrySet().iterator();
```

If the node statistics of a node is null then we get its id in the database and update its statistics. This following function in the code will do that:

```
this.substringQueryInSingleNode(query, nodeHilbertCode, nodeStats,
substringResult, memObjectMap, memHilbQgramTokenInvertedMap);
```

Function 3 – Approximate Search

```
if ( intersectingNodeStatsMap.denseNodeNumber < visitingNodeSizeThreshold )
{
    Iterator< Entry< String, NodeStatistic >> itr =
intersectingNodeStatsMap.entrySet().iterator();

    while(itr.hasNext())
    {
        Entry<String, NodeStatistic> entry = itr.next();
        String nodeHilbertCode = entry.getKey();
        nodeStats = entry.getValue();

        //substring query in node
        //2 types of argument parameters
        if(nodeStats != null)
        {
            this.approximatePrefixQueryInSingleNode(query,
approximatePrefixResult, nodeHilbertCode, nodeStats, memObjectMap,
memInfreqPosQgramInvertedMap, memInfreqQgramTokenInvertedMap,
memHilbPosQgramInvertedMap,
memHilbQgramTokenInvertedMap, qgramFilter,
mergeSkipOperator, approximateSubstringCandidateMap);
        }
    }
    // approximate prefix query in node
}
```

This function of approximate query does not need any filters to be done when iterating through the nodes as it was already being filtered by the previous functions. Therefore, this part of the code iterates through the intersecting nodes to get its Hilbert code and node statistics. If the statistics of the node is null, the function `approximatePrefixQueryInSingleNode` will create its statistics.


```

else
{
    TreeSet< String > approximatePrefixNodeCandidates = new TreeSet<
String >();

    // node filter 2 filters
    //this.approximatePrefixNodeFilter(query, tokenMinMatchThreshold,
intersectingNodeStatsMap, approximatePrefixNodeCandidates);
    this.approximatePrefixNodeFilter(query, tokenMinMatchThreshold,
intersectingNodeStatsMap, doApproximateSubstringNodeFilter,
approximatePrefixNodeCandidates,
approximateSubstringNodeCandidates,
memPosQgramCountPairInvertedMap, memQgramTokenCountPairInvertedMap);
    // object process in node
    if ( ! approximatePrefixNodeCandidates.isEmpty() )
    {
        for ( String nodeHilbertCode : approximatePrefixNodeCandidates )
        {
            this.approximatePrefixQueryInSingleNode(query,
approximatePrefixResult, nodeHilbertCode, nodeStats, memObjectMap,
memInfreqPosQgramInvertedMap, memInfreqQgramTokenInvertedMap,
memHilbPosQgramInvertedMap,
memHilbQgramTokenInvertedMap, qgramFilter,
mergeSkipOperator, approximateSubstringCandidateMap);
        }
    }
}
}

```

As shown in the INSPIRE framework, each spatial query function will have their filters. However, for approximate search query we have to filter the nodes 2 times. The first one:

```

this.approximatePrefixNodeFilter(query, tokenMinMatchThreshold,
intersectingNodeStatsMap, doApproximateSubstringNodeFilter,
approximatePrefixNodeCandidates, approximateSubstringNodeCandidates,
memPosQgramCountPairInvertedMap, memQgramTokenCountPairInvertedMap);

```

Is to filter node with approximate search filter of PrF, PoF and CF. The second one:

```

this.approximatePrefixQueryInSingleNode(query, approximatePrefixResult,
nodeHilbertCode, nodeStats, memObjectMap, memInfreqPosQgramInvertedMap,
memInfreqQgramTokenInvertedMap, memHilbPosQgramInvertedMap,
memHilbQgramTokenInvertedMap, qgramFilter, mergeSkipOperator,
approximateSubstringCandidateMap);

```

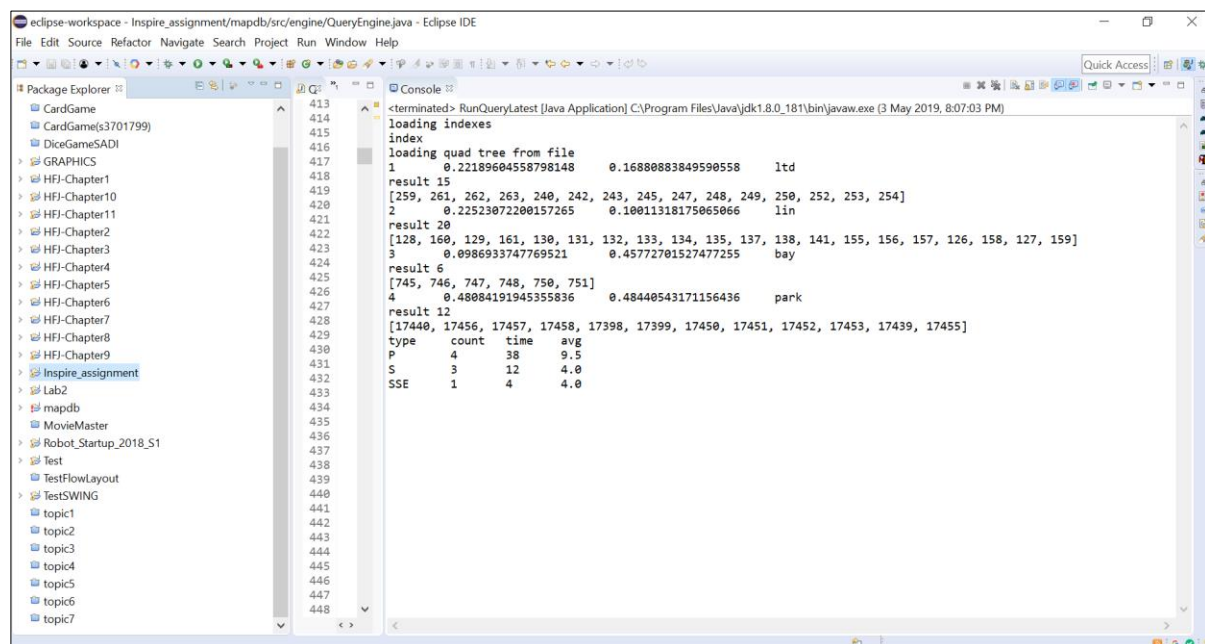
Is to apply the approximate search filter of the node and check its objects.

Program output

Expected

```
loading indexes
index
loading quad tree from file
1      0.22189604558798148 0.16880883849590558 ltd
result 15
[259, 261, 262, 263, 240, 242, 243, 245, 247, 248, 249, 250, 252, 253, 254]
2      0.22523072200157265 0.10011318175065066 lin
result 20
[128, 160, 129, 161, 130, 131, 132, 133, 134, 135, 137, 138, 141, 155, 156, 157,
126, 158, 127, 159]
3      0.0986933747769521 0.45772701527477255 bay
result 6
[745, 746, 747, 748, 750, 751]
4      0.48084191945355836 0.48440543171156436 park
result 12
[17440, 17456, 17457, 17458, 17398, 17399, 17450, 17451, 17452, 17453, 17439,
17455]
null
type    count    time    avg
P        4        25      6.25
S        3        10      3.3333333333333335
SSE      1         3       3.0
```

Output



```
eclipse-workspace - Inspire_assignment/mapdb/src/engine/QueryEngine.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer
CardGame
CardGame(s3701799)
DiceGameSADI
> GRAPHICS
> HFJ-Chapter1
> HFJ-Chapter10
> HFJ-Chapter11
> HFJ-Chapter2
> HFJ-Chapter3
> HFJ-Chapter4
> HFJ-Chapter5
> HFJ-Chapter6
> HFJ-Chapter7
> HFJ-Chapter8
> HFJ-Chapter9
> Inspire_assignment
> Lab2
> mapdb
> MovieMaster
> Robot_Startup_2018_S1
> Test
> TestFlowLayout
> TestSWING
> topic1
> topic2
> topic3
> topic4
> topic5
> topic6
> topic7

Console
<terminated> RunQueryLatest [Java Application] C:\Program Files\Java\jdk1.8.0_181\bin\javaw.exe (3 May 2019, 8:07:03 PM)
loading indexes
index
loading quad tree from file
1      0.22189604558798148 0.16880883849590558 ltd
result 15
[259, 261, 262, 263, 240, 242, 243, 245, 247, 248, 249, 250, 252, 253, 254]
2      0.22523072200157265 0.10011318175065066 lin
result 20
[128, 160, 129, 161, 130, 131, 132, 133, 134, 135, 137, 138, 141, 155, 156, 157, 126, 158, 127, 159]
3      0.0986933747769521 0.45772701527477255 bay
result 6
[745, 746, 747, 748, 750, 751]
4      0.48084191945355836 0.48440543171156436 park
result 12
[17440, 17456, 17457, 17458, 17398, 17399, 17450, 17451, 17452, 17453, 17439, 17455]
type    count    time    avg
P        4        25      6.25
S        3        10      3.3333333333333335
SSE      1         3       3.0
```