Question 1: Implementation

Overview:

In this assignment, I implemented two multi-way natural join algorithms: sortmerge join and leapfrog join. They are both very fast. Because some intermediate results cannot be fit into the main memory, the sortmerge join cannot complete query3 on scale4, scale5 and scale6 of dataset2-zipf.

The sortmerge algorithm joins the relations in a pair-wise sequential manner. For example, in query3 we want to join R, S, T, U, V, W on A, B, C, X, Y, Z, the sortmerge will first join R and S on A, then join the previous result RS and T on B, C; then RST and U on X; then RSTU and V on Y, and finally RSTUV and W on Z.

The leapfrog join algorithm is implemented followed the provided paper. After some modification, especially the handling of duplicated results, it produces identical results with those of sortmerge join.

Highlights:

1. Handling duplicated records. There are two forms of duplication that make the result of leapfrog triejoin differ from that of sortmerge join.
    a. The first form is the complete duplication of an entire record, i.e., two or more lines of records are same. This is very common in relation P and Q. My solution is to add a unique attribute for each record (like the primary key), which is the line number (LineNo) of with record within its relation. This simple trick makes each record differ from all other records.
    b. The second form is the existence of the non-join attributes. For example, if we want to join R, S, T on A, B, C, as there are another three attributes X, Y, Z as well as three additional LineNo involved. The original triejoin proposed in the paper does not consider this situation, so we will only keep one of the several joined records that may have different X, Y, Z or LineNo values. The solution is simple: after joining on the required attributes, we continue to join on the non-join attributes. Therefore in the above example, we will join in the sequence of (A, B, C, RLineNo, X, SLineNo, Y, TLineNo, Z). It is worth mentioning that the order of non-join attributes will not affect the final result.

    After handling the two forms of duplication, the triejoin and sortmerge are now giving identical results.
2. Efficient array implementation of the binary search tree structure. Please refer to the Question3: Extension (b) part of this report for more details.
3. Flexible trie structure to overcome memory overflow issue. When performing leapfrog triejoin, the speed can be greatly improved if we pre-build the trie structure, i.e., the map between the trie paths of attribute values and LinearIterators. This can be seen from the benchmark section of this report. However, when the size of the input relation is too large,

say the scale6 version of P and Q, the map cannot be fitted inside the memory. To handle this situation, my implementation of the TrieIterator has the flexibility of generating LinearIterators on the fly (i.e. when performing TrieIterator::open() to explore the next layer of trie) or at the beginning.

4. Other efficient implementation. Many tricks such as bookkeeping or avoiding expensive operations are used throughout the implementation. And by default, just the counts but not the record of the final results are kept. Because for large scale databases, especially those following Zipf distribution, there are just too many records to fit in the memory or my disk. But for sortmerge join, intermediate results are kept in order to perform the following joins.

Questions:

(a) One RelationSpec object is defined for each relation, and all the record are stored as int* in the RelationSpec::memDB std::vector<int*>. So I am not implementing a SimpleIterator myself, but using the index to access std::vector<int*>. All key(), next() and at_end() function can be easily implemented using the interfaces of std::vector<int*>.

(b) Please refer to the sequential_sortmege_join and the sortmerge_join function defined in SortMerge.cpp. I sorted the RelationSpec::memDB according to the join attribute at each time, which can be regarded as using one SimpleIterator per input relation to be joined.

(c) Please refer to the LinearIterator class defined in the LinearIterator.cpp. It is designed to be used in a TrieIterator, so it contains some information such as current depth. Rather than using some existing balanced search tree library, I implemented my own version of an array based binary search structure in CompleteArrayBST.cpp. To find out more details about it and how it can meet the complexity requirement, please refer to the Question3: Extension (b) part of this report.

(d) Please refer to the TrieIterator class defined in TrieIterator.cpp. The TrieIterator class is implemented based on the pseudo-code provided by Todd L. Veldhuizen. As mentioned in the highlight section, either a map between the trie paths of attributes values and LinearIterators is maintained or LinearIterators for the next depth are calculated on the fly. For TrieIterator::open(), as there are at most N different paths for one relation, the cost of finding corresponding LinearIterator is O(logN) using std::map; for TrieIterator::up(), with the help of stack, the cost is just O(1); for TrieIterator::seek(), TrieIterator::next(), TrieIterator::key(), they just call the interface of corresponding LinearIterators, thus the complexity is carried over from LinearIterators.

(e) Please refer to the leapfrog_triejoin function and the TrieJoin class implemented in TrieJoin.cpp. I am using one TrieIterator per input relation to be joined.

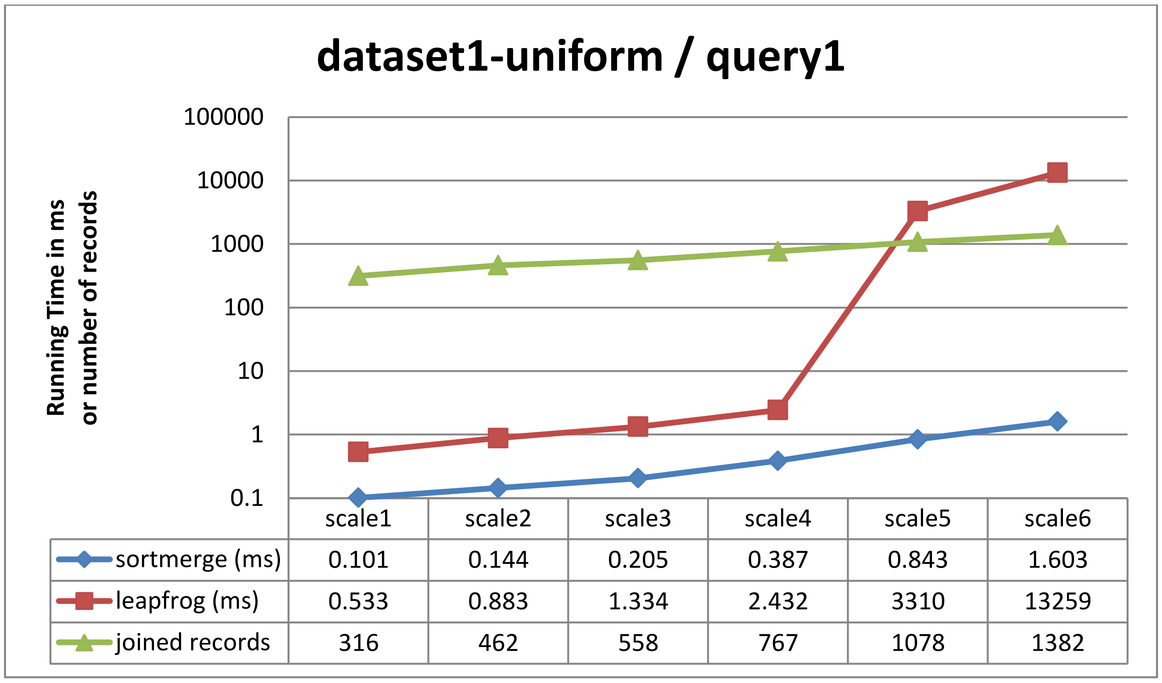
Question2: Benchmark

(a) Environment: VM-Fedora 17 running on Mac OSX 10.8.3 with 8GB memory and 4-core CPU at 2.6GHz. The running time for small numbers is averaged over 10 runs. The running time does not include the time for constructing trie structure unless the trie is built on the fly. The running time includes the sorting time for sortmerge algorithm. All the graphs are plotted in the ***logarithmic*** scale for Y-axis (running time in ms or number of records).

dataset1-uniform / query1

	sortmerge (ms)	leapfrog (ms)	joined records
scale1	0.101	0.533	316
scale2	0.144	0.883	462
scale3	0.205	1.334	558
scale4	0.387	2.432	767
scale5	0.843	3310*	1078
scale6	1.603	13259*	1382

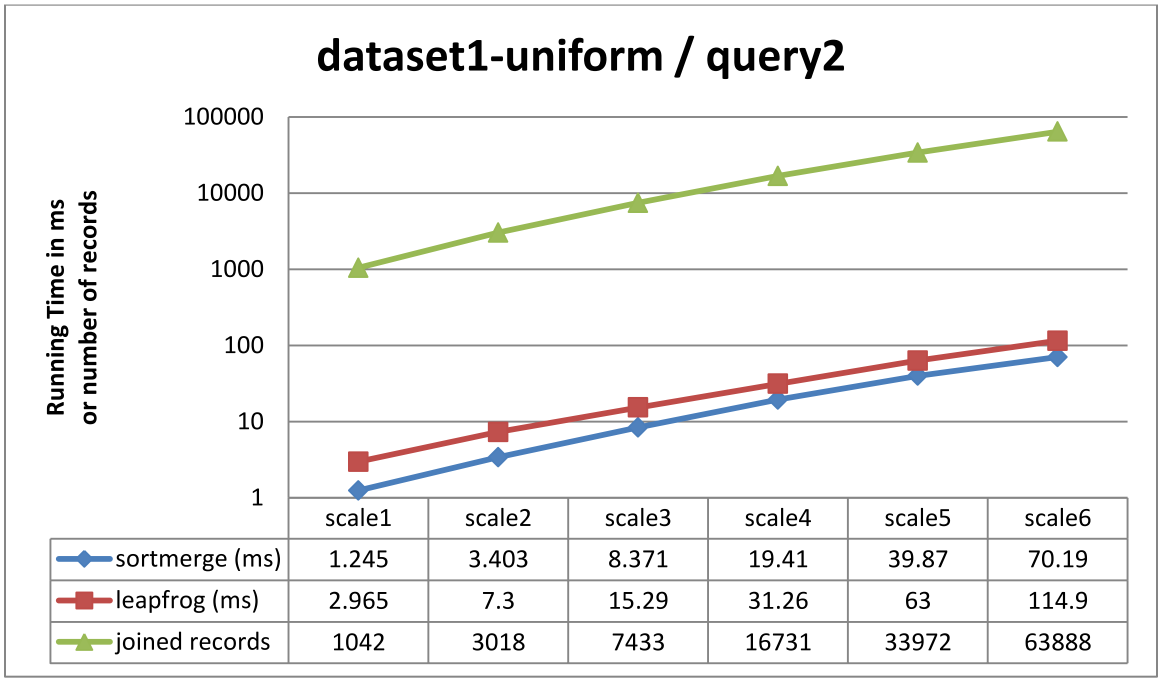
*** indicates building the trie structure on the fly**



	scale1	scale2	scale3	scale4	scale5	scale6
sortmerge (ms)	0.101	0.144	0.205	0.387	0.843	1.603
leapfrog (ms)	0.533	0.883	1.334	2.432	3310	13259
joined records	316	462	558	767	1078	1382

The values of records in dataset1 are generated following a uniform distribution. This means the joined results will not be very large. Query1 is perhaps the easiest for sortmerge, as it can finish the largest scale in less than 2ms and produce 1382 joined records. The leapfrog triejoin algorithm can solve scale1-4 with ease, but it encounters problem when facing scale5 and scale6 as the numbers of records in relation P and Q are so large. I cannot pre-build the trie structure (which has a space complexity of O(N)) in those cases as the path-LinearIterator map cannot fit inside the memory. Therefore the build-on-the-fly version of TrieIterator is adopted and this slows down the iteration operation dramatically.

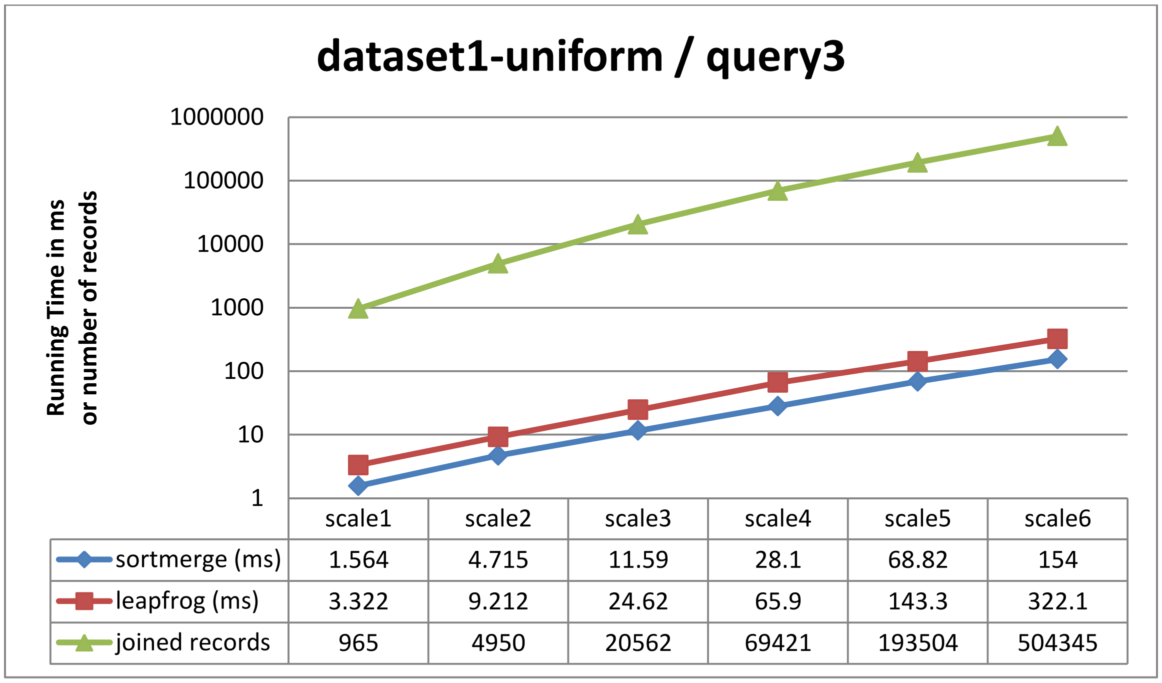
dataset1-uniform / query2			
	sortmerge (ms)	leapfrog (ms)	joined records
scale1	1.245	2.965	1042
scale2	3.403	7.300	3018
scale3	8.371	15.29	7433
scale4	19.41	31.26	16731
scale5	39.87	63.00	33972
scale6	70.19	114.9	63888



	scale1	scale2	scale3	scale4	scale5	scale6
sortmerge (ms)	1.245	3.403	8.371	19.41	39.87	70.19
leapfrog (ms)	2.965	7.3	15.29	31.26	63	114.9
joined records	1042	3018	7433	16731	33972	63888

Query2 of dataset1 is easy for both sortmerge and leapfrog. The largest scale that yields 63888 records can be solved within 120ms. For the logarithmic scale graph, we can see a roughly linear relationship between the scales (in terms of the number of final records) and the running time of two join algorithms. We can also see that sortmerge is faster than leapfrog in all 6 scales and the ratio more or less remains constant where sortmerge is roughly twice as fast.

dataset1-uniform / query3			
	sortmerge (ms)	leapfrog (ms)	joined records
scale1	1.564	3.322	965
scale2	4.715	9.212	4950
scale3	11.59	24.62	20562
scale4	28.10	65.90	69421
scale5	68.82	143.3	193504
scale6	154.0	322.1	504345

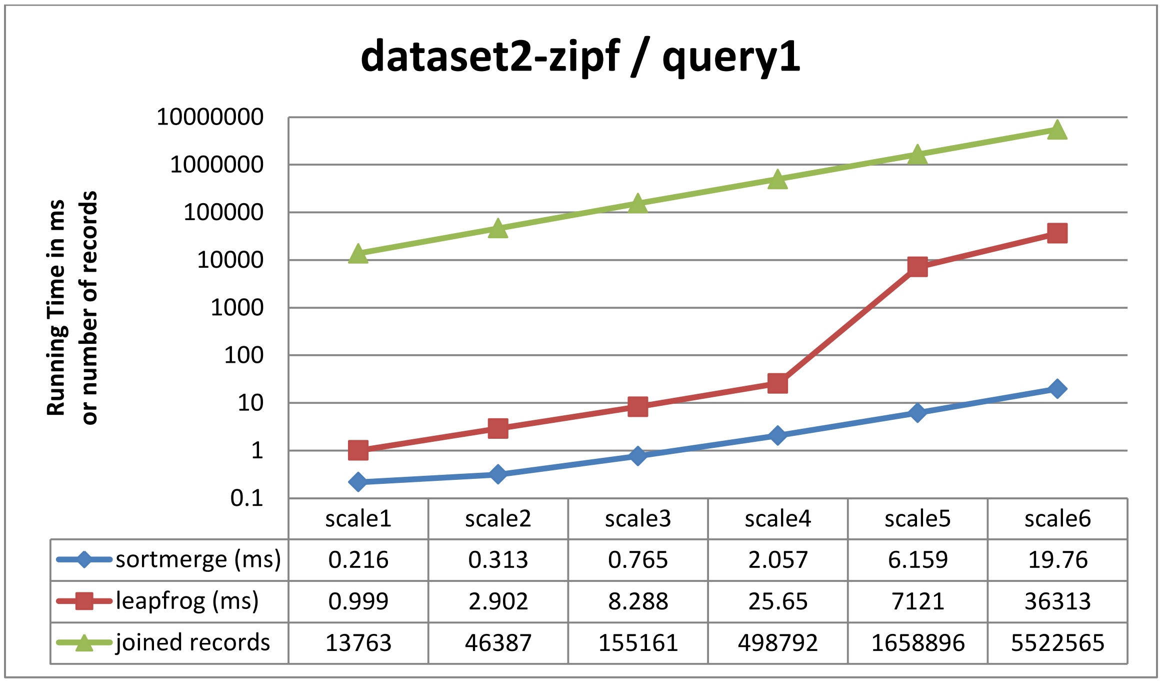


	scale1	scale2	scale3	scale4	scale5	scale6
sortmerge (ms)	1.564	4.715	11.59	28.1	68.82	154
leapfrog (ms)	3.322	9.212	24.62	65.9	143.3	322.1
joined records	965	4950	20562	69421	193504	504345

Both sortmerge and leapfrog perform well in this setting. The plot follows a similar pattern to that of dataset1-query2. Therefore the conclusion is similar.

dataset2-zipf / query1			
	sortmerge (ms)	leapfrog (ms)	joined records
scale1	0.216	0.999	13763
scale2	0.313	2.902	46387
scale3	0.765	8.288	155161
scale4	2.057	25.65	498792
scale5	6.159	7121*	1658896
scale6	19.76	36313*	5522565

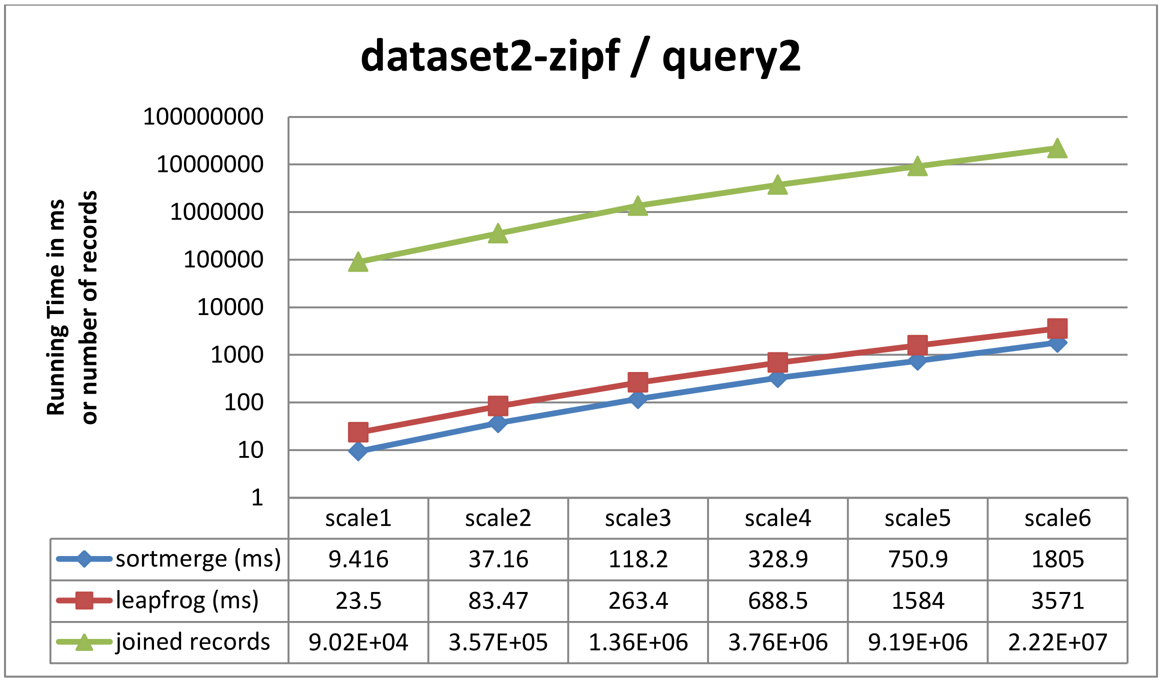
*** indicates building the trie structure on the fly**



	scale1	scale2	scale3	scale4	scale5	scale6
sortmerge (ms)	0.216	0.313	0.765	2.057	6.159	19.76
leapfrog (ms)	0.999	2.902	8.288	25.65	7121	36313
joined records	13763	46387	155161	498792	1658896	5522565

The values of records in dataset2 are generated following a Zipf distribution. Although the numbers of records in each relation are the same as those in dataset1 respectively, because of the property of Zipf distribution, a small portion of value has many duplicates, the sizes of joined results are much larger. But for query1, sortmerge can still handle it easily as no intermediate results are needed and we only join once. For leapfrog triejoin, we have the same problem on large scale as dataset1. Therefore, the build-on-the-fly version is again adopted on scale5 and scale6. However, it is worth mentioning that when representing relations of the same size, tries use less memory when values are drawn from a Zipf distribution than a uniform distribution. As at each node, tries only need to store one value no matter how many duplicates there are.

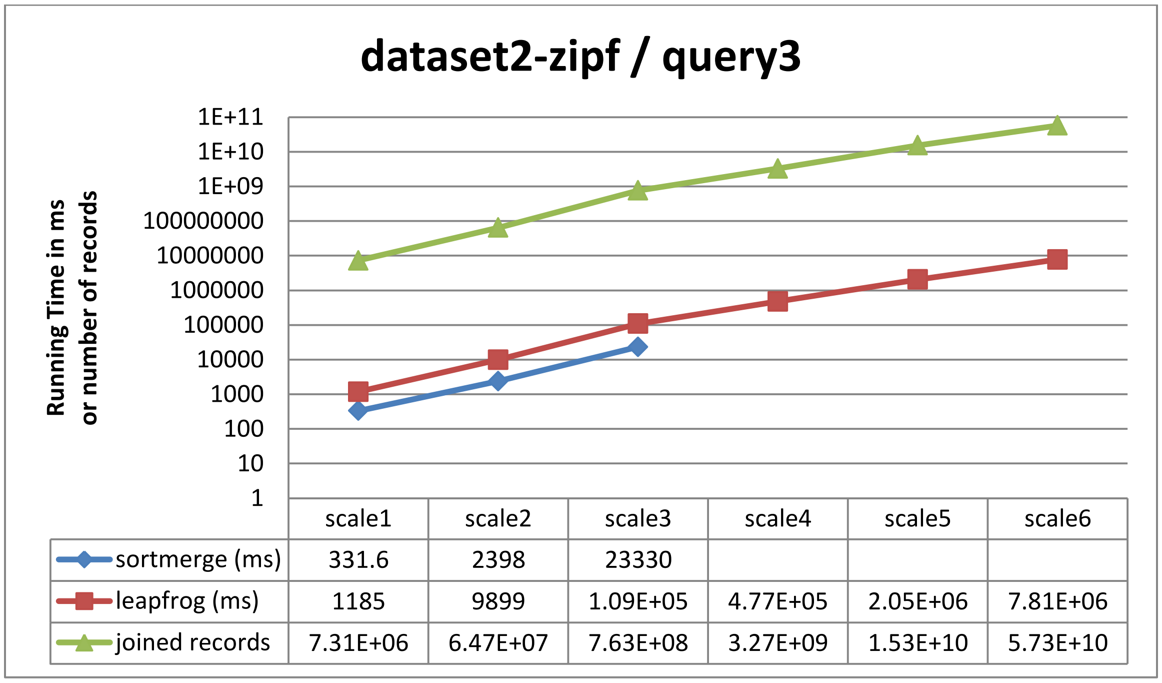
dataset2-zipf / query2			
	sortmerge (ms)	leapfrog (ms)	joined records
scale1	9.416	23.50	90211
scale2	37.16	83.47	357447
scale3	118.2	263.4	1356609
scale4	328.9	688.5	3759803
scale5	750.9	1584	9186514
scale6	1805	3571	22152989



	scale1	scale2	scale3	scale4	scale5	scale6
◆ sortmerge (ms)	9.416	37.16	118.2	328.9	750.9	1805
■ leapfrog (ms)	23.5	83.47	263.4	688.5	1584	3571
▲ joined records	9.02E+04	3.57E+05	1.36E+06	3.76E+06	9.19E+06	2.22E+07

Although the sizes of final results start to get large, both sortmerge and leapfrog can still handle them within a reasonable time. We have no memory issue for both join algorithms. The plot follows a similar pattern to those of deteset1-query2 and deteset1-query3. Therefore the conclusion is similar.

dataset2-zipf / query3			
	sortmerge (ms)	leapfrog (ms)	joined records
scale1	331.6	1185	7313287
scale2	2398	9899	64691341
scale3	23330	1.09E+05	762545136
scale4	-	4.77E+05	3274331787
scale5	-	2.05E+06	15320032312
scale6	-	7.81E+06	57335153511



	scale1	scale2	scale3	scale4	scale5	scale6
sortmerge (ms)	331.6	2398	23330			
leapfrog (ms)	1185	9899	1.09E+05	4.77E+05	2.05E+06	7.81E+06
joined records	7.31E+06	6.47E+07	7.63E+08	3.27E+09	1.53E+10	5.73E+10

Dataset2-query3 is the hardest one. With the largest scale6, we have 57335153511 joined records, that is more than 57 billion and far more than a normal 32bit integer can hold. In fact, as my version of sortmerge doing its job in a pair-wise sequential manner, it cannot deal with scale 4, 5 and 6 because there is not enough memory space for the intermediate results. However, as leapfrog triejoin doing its job inherently multi-way, it can produce the results even for the largest scale in roughly two hours.

Question 3: Extensions

(a) In my implementation of the TrieIterator class, the trie representation or the proposed hash table representation of the relation is interchangeable. According to the paper, when TrieIterator::open() is invoked at some trie node n, the linear iterator methods LinearIterator::next(), LinearIterator::seek() and LinearIterator::atEnd() present the children of n. Therefore in my implementation, I just use the path from the trie root to the current node as the key to retrieve the corresponding LinearIterator, and it is the LinearIterator's responsibility to retrieve the desired value in order. Using a chain of hash tables can do the exact same task for me, i.e., by saving the path from the first attributes to the current attributes when going along the chain. And, of course, when constructing the hash table chain, its order should be compatible with the join order.

(b) My implementation of the LinearIterator class does not rely on a traditional binary search tree. To satisfy the complexity requirements of the LinearIterator class, a structure similar to balanced binary search tree is still needed. Keeping in mind that, in the LinearIterator, no insertion is needed after the initial construction of the "tree", i.e., the size and all the data are known at the beginning. Therefore I can use a fixed complete binary search tree for this task. Moreover, because of the property of the complete binary tree[1], it can be represented using an array. So my implementation is an array-based binary search structure. It has multiple advantages over a traditional tree:

1) Efficient to build. When building the CompleteArrayBST from an sorted array, it is recursively built using the following time cost: $\mathrm{T(n)} = 2 * \mathrm{T(n/2)} + \mathrm{O(1)}$. According to the master theorem. The complexity of T(n) is O(N). And because every O(1) step is just simple calculation of the array index, it saves time in memory allocation and pointer operation comparing to constructing a traditional binary search tree.
2) Efficient to find. Because the left/right/parent node can be easily calculated by simple index calculation. Supposing the iterator is at position cur, then:
    i. find left (if exist): cur -> cur*2+1
    ii. find right (if exist): cur -> cur*2+2
    iii. find parent (if exist): odd cur -> (cur-1)/2; even cur -> (cur-2)/2
3) Efficient to seek. When performing the seek() function, rather than returning to the tree root for each seek() operation, the iterator starts from the current position and just iterates far enough to find an upper bound for the key sought (refer to the implementation of CompleteArrayBST::seek()). This will satisfy the amortized complexity of O(1+log(N/m)) if m keys are visited in ascending order.
4) Space efficient. Unlike traditional tree structure, no additional space for left/right child pointers or parent pointer is needed. Exact N integer slots are needed for each value in the array.

[1] A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. (Wikipedia)